DESENVOLVIMENTO DE APLICAÇÕES WEB EM JAVA E OUTRAS TECNOLOGIAS

Atualizado para Java 17 e Jakarta EE 10

Terceira Edição

DAVID BUZATTO

IFSP - CÂMPUS SÃO JOÃO DA BOA VISTA

DESENVOLVIMENTO DE APLICAÇÕES WEB EM JAVA E OUTRAS TECNOLOGIAS

ATUALIZADO PARA JAVA 17 E JAKARTA EE 10

Terceira Edição

DAVID BUZATTO

Instituto Federal de Educação, Ciência e Tecnologia de São Paulo Câmpus São João da Boa Vista

Dados Internacionais de Catalogação na Publicação (CIP) (Câmara Brasileira do Livro, SP, Brasil)

Buzatto, David

Desenvolvimento de aplicações web em Java e outras tecnologias [livro eletrônico] : atualizado para Java 17 e Jakarta EE 10 / David Buzatto. --3. ed. -- Vargem Grande do Sul, SP : Ed. do Autor, 2025.

PDF

Bibliografia. ISBN 978-65-01-28800-0

1. Ciência da computação 2. Java (Linguagem de programação para computadores) 3. Tecnologia

4. WEB (Linguagem de programação) I. Título.

25-246676 CDD-005.133

Índices para catálogo sistemático:

1. Java : Lingagem de programação : Computadores : Processamento de dados 005.133

Aline Graziele Benitez - Bibliotecária - CRB-1/3129

À minha filha Aurora, luz da minha vida

> À minha esposa, Fernanda

À minha mãe, Selma

SUMÁRIO

Pr	efáci		V
I	O B	sásico do Básico	1
1	Java	para Web	3
	1.1	Introdução	3
	1.2	Contêineres de Servlets e Servidores de Aplicações	7
	1.3	Servlets e JSPs	8
	1.4	Preparação do Ambiente de Desenvolvimento	9
		1.4.1 Apache NetBeans	9
		1.4.2 Eclipse GlassFish	10
		1.4.3 Primeiro Projeto Java para Web	13
	1.5	Resumo	22
	1.6	Exercícios	23
	1.7	Projetos	23
2	Pro	cessamento de Formulários	25
	2.1	Introdução	25
	2.2	Formulários	26
	2.3	Métodos HTTP	35
		2.3.1 Método GET	35
		2.3.2 Método POST	37
		2.3.3 Tratando Métodos HTTP	38
	2.4	Resumo	39
	2.5	Exercícios	39
	2.6	Projetos	39
3	Ехр	ression Language e TagLibs	43
	3.1	Introdução	43
	3.2	Expression Language (EL)	44

ii SUMÁRIO

	3.3	Tags JSP
	3.4	JavaServer Pages Standard Tag Library - JSTL
	3.5	Resumo
	3.6	Exercícios
	3.7	Projetos
4	Pad	rões de Projeto: <i>Factory</i> , DAO e MVC 63
	4.1	Introdução
	4.2	Preparando o Ambiente
	4.3	Padrão de Projeto Factory
	4.4	Padrão de Projeto <i>Data Access Object</i> (DAO)
	4.5	Padrão de Projeto <i>Model-View-Controller</i> (MVC) 92
	4.6	Resumo
	4.7	Exercícios
	4.8	Projetos
5	Siste	ema para Controle de Clientes 95
	5.1	Introdução
	5.2	Analisando os Requisitos
	5.3	Projetando Banco de Dados
	5.4	Criando o Diagrama de Classes
	5.5	Construindo o Sistema
	5.6	Resumo
	5.7	Projetos
6	Prin	neiro Projeto: Sistema para Locação de DVDs v1.0 175
	6.1	Introdução
	6.2	Apresentação dos Requisitos
	6.3	Desenvolvimento do Projeto
	6.4	Resumo
TT	0.0	res Pales de Décises
11	υŲ	ue Falta do Básico?
7		odução à Linguagem JavaScript 179
	7.1	Introdução
	7.2	Funções de E/S e Operadores Aritméticos
	7.3	Declarações de Variáveis e Suas Implicações
	7.4	Estruturas Condicionais e Operadores
	7.5	Estruturas de Repetição e Arrays
	7.6	"Classes", Objetos e JSON

SUMÁRIO iii

	7.7	Manipulação do DOM	216
		7.7.1 JavaScript Puro	217
		7.7.2 Usando jQuery	218
	7.8	Manipulação de Formulários	219
	7.9	Eventos	223
		Simulação Usando a Canvas API	224
	7.11	Requisições Assíncronas e Intercâmbio de Dados	234
		7.11.1 AJAX com jQuery e com Fetch API	236
		7.11.2 AJAX com jQuery e com Fetch API Processando JSON	239
	7.12	Resumo	241
	7.13	Projetos	241
8	Siste	ma para Venda de Produtos	243
	8.1	Introdução	243
	8.2	Construindo o Sistema	245
		8.2.1 Entidades e Validações	245
		8.2.2 DAO	255
		8.2.3 Servlets	266
		8.2.4 Cadastro de Vendas	272
	8.3	Resumo	307
	8.4	Projetos	308
	8.5	Desafios	308
9	Segu	ndo Projeto: Sistema para Locação de Mídias	311
	9.1	Introdução	311
	9.2	Apresentação dos Requisitos	312
	9.3	Desenvolvimento do Projeto	313
	9.4	Resumo	313
III	Fina	alizando	315
10	Conc	clusão	317
Bib	oliogr	afia	319

PREFÁCIO

"Any fool can write code that a computer can understand. Good programmers write code that humans can understand".

Martin Fowler



REZADO leitor, seja bem-vindo! Os Capítulos deste livro foram organizados de forma a guiá-lo no processo de fixação dos conteúdos aprendidos durante a sua leitura por meio de exercícios, desafios e de projetos práticos aplicados no contexto de desenvolvimento de aplicações Web em

Java e outras tecnologias. A ordem dos Capítulos obedece a um caminho lógico que será empregado para auxiliá-lo no processo de aprendizagem das tecnologias e técnicas discutidas, ou seja, a organização dos Capítulos segue a ordem cronológica dos tópicos que serão apresentados, ensinados e treinados.

Antes de começar, eu gostaria de me apresentar. Meu nome é David Buzatto e sou Bacharel em Sistemas de Informação pela Fundação de Ensino Octávio Bastos (2007), Mestre em Ciência da Computação pela Universidade Federal de São Carlos (2010) e Doutor em Biotecnologia pela Universidade de Ribeirão Preto (2017). Atualmente meus principais interesses giram em torno de temas relacionados à construção de compiladores, teoria da computação, análise e projeto de algoritmos, estruturas de dados, algoritmos em bioinformática e desenvolvimento de jogos



eletrônicos. Atualmente sou professor efetivo do Instituto Federal de Educação, Ciência e Tecnologia de São Paulo (IFSP), câmpus São João da Boa Vista. A melhor forma de contatar é através do e-mail davidbuzatto@ifsp.edu.br.

Para que você possa aproveitar a leitura deste livro de forma plena, vale a pena entender alguns padrões que serão utilizados neste texto. As quatro caixas apresentadas

vi PREFÁCIO

abaixo serão empregadas para mostrar, a você leitor, respectivamente, boas práticas de programação, conteúdos complementares para melhorar e aprofundar seu aprendizado, observações pertinentes ao que está sendo apresentado e, por fim, itens que precisam ser tratados com cuidado ou que podem acarretar em erros comuns de programação.



Essa é uma caixa de "Boa Prática".

(i) | Saiba Mais

Essa é uma caixa de "Saiba Mais".

Ω| Observação

Essa é uma caixa de "Observação".

∧| Atenção!

Essa é uma caixa de "Atenção!".

Você notará que este este livro foi escrito de forma quase coloquial, com o objetivo de conversar com você e não com o objetivo de ser um material de pesquisa. É de suma importância que você procure implementar os exemplos apresentados e que resolva cada um dos exercícios básicos de cada Capítulo, visto que a utilização de uma linguagem de programação ou tecnologia, e mais importante ainda, a obtenção de maturidade no desenvolvimento de *software*, é ferramenta primordial para o seu sucesso profissional e intelectual.

Vale ressaltar que este livro e os projetos apresentados serão constantemente atualizados, sendo assim, sempre obtenha as últimas versões dos mesmos aqui. Espero que essa obra, de distribuição totalmente gratuita, lhe seja útil!

Parte I O Básico do Básico...

JAVA PARA WEB

"Uma longa viagem começa com um único passo".

Lao-Tsé



ESTE Capítulo teremos como objetivos entender o funcionamento e a arquitetura de aplicações Web desenvolvidas em Java, entender o funcionamento dos Servlets e dos JSPs e aprender a configurar e a utilizar a *Integrated Development Environment* (IDE) Apache NetBeans para o

apoio ao desenvolvimento de aplicações Web em Java.

1.1 Introdução

Para que você seja capaz de construir aplicações Web, primeiramente é preciso conhecer como esse serviço é estruturado. A *World Wide Web* (WWW), ou simplesmente Web, é um serviço executado em diversos computadores interligados em uma rede mundial, sendo que em alguns desses computadores são executados programas chamados de **servidores**, enquanto na maioria dos outros são executados programas chamados de **clientes**, que se comunicam com os servidores que, por sua vez, servem recursos para esses clientes. Na Figura 1.1 é ilustrado um recorte desta rede mundial.

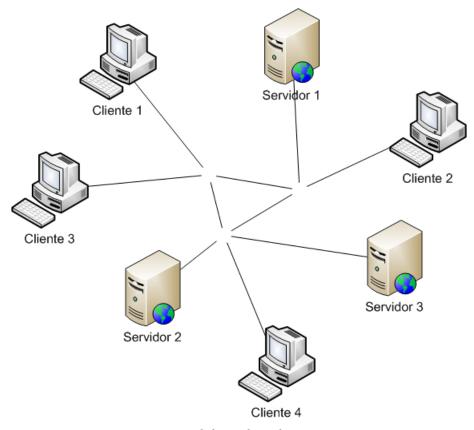
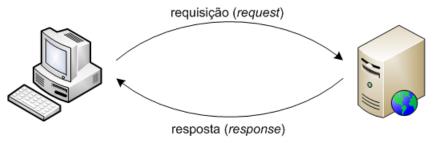


Figura 1.1: Recorte da estrutura da WWW

Perceba que no recorte apresentado na Figura 1.1 são mostrados sete computadores, sendo que quatro deles atuam como clientes e os outros três como servidores. É importante entender que o que faz um computador ser cliente ou servidor é o tipo de programa que está sendo usado/executado. No nosso exemplo, as máquinas que atuam como servidores executam um programa denominado Servidor Web, que tem a capacidade de servir (disponibilizar) aos outros computadores da rede, recursos que fazem parte de uma aplicação Web, por exemplo, arquivos *Hypertext Markup Language* (HTML), imagens em diversos formatos, arquivos de estilo, arquivos de *script* etc. Os clientes, por sua vez, são, na maioria das vezes, os conhecidos navegadores Web, ou *browsers*, que usamos no nosso dia a dia para acessar a Web e navegar em diversos *sites*.

Da mesma forma que existem diversos navegadores, existem também alguns Servidores Web, sendo o Apache o mais famoso e o mais utilizado. Como já foi dito, um Servidor Web tem a função de servir recursos requisitados pelos clientes. Vamos aprender como isso funciona. Veja a Figura 1.2.

Figura 1.2: Processo de requisição e resposta (request/response)



Na Figura 1.2 é mostrado o processo de requisição e resposta. Nesse processo, o cliente à esquerda (navegador), envia uma requisição a um recurso contido no Servidor Web (à direita) através de uma *Uniform Resource Locator* (URL), sendo que nessa requisição, o cliente especifica o protocolo a ser usado, o endereço do servidor e o caminho para o recurso. Assim, uma URL tem a seguinte forma:

protocolo://máquina/caminho_do_recurso

• Onde:

- protocolo: É a parte da URL que diz ao servidor qual o protocolo a ser utilizado. Quando acessamos páginas Web, por padrão, o protocolo utilizado é o Hypertext Transfer Protocol (HTTP);
- máquina: É o nome ou o endereço codificado pelo *Internet Protocol* (IP)
 da máquina que está executando o Servidor Web;
- caminho_do_recurso: É o caminho completo do recurso desejado que é disponibilizado pelo servidor.

Confuso? Nem tanto. Vamos a um exemplo! Imagine a seguinte situação: Queremos acessar o site do IFSP. Para isso, abra o seu navegador e preencha o campo endereço com http://www.ifsp.edu.br/ e tecle <ENTER>. Fazendo isso, o navegador envia uma requisição através de uma URL, usando o protocolo HTTP para a máquina www.ifsp.edu.br, que por sua vez retorna ao navegador uma página HTML que representa aquele endereço.

Perceba que não especificamos o caminho do recurso! Isso não foi necessário, pois os Servidores Web são normalmente configurados para ter um comportamento padrão para responder às requisições onde só seja especificado o nome da máquina e esse comportamento padrão é direcionar para o recurso index.html, que é um arquivo HTML. Portanto, usar o endereço http://www.ifsp.edu.br/é o mesmo que usar o endereço http://www.ifsp.edu.br/index.html. Faça um teste! Coloque o ende-

reço com o caminho do recurso (index.html) e tecle <ENTER>. O que aconteceu? A mesma página foi exibida não foi? Ótimo!

Vamos fazer mais um teste? Preencha novamente a barra de endereços no seu navegador com o endereço https://avatars.githubusercontent.com/u/313050 e tecle <ENTER>. Deve ter aparecido a minha foto, certo? Vamos analisar a URL: usamos o protocolo HTTP, para pedir para o Servidor Web que está executando na máquina avatars.githubusercontent.com o recurso 313050, que está armazenado no caminho /u/. Este processo é ilustrado na Figura 1.3.

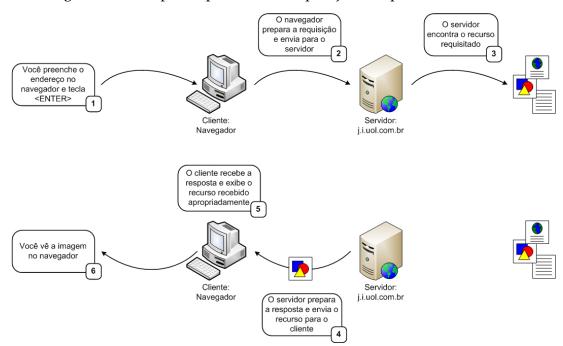


Figura 1.3: Exemplo do processo de requisição e resposta a um recurso

Fonte: Elaborada pelo autor

Perceba que os passos 1, 2 e 3 da Figura 1.3 representam o processo de requisição, enquanto os passos 4, 5 e 6 representam o processo de resposta. Note também que ao clicar em qualquer $link^1$ em uma página, todo esse processo é executado pelo navegador.

Muito bem! Agora que você já sabe como funciona o processo de requisição e resposta entre um navegador e um Servidor Web, podemos prosseguir com nossos estudos, agora focando em como a linguagem e a plataforma Java são utilizadas para trabalhar com aplicações Web.

¹O termo *link* será usado para designar os *hyperlinks* dos arquivos HTML

1.2 Contêineres de Servlets e Servidores de Aplicações

Na Seção anterior você aprendeu algumas coisas novas², mas onde que a linguagem Java se encaixa nisso tudo? Quando criamos aplicações Web, além de termos páginas com código HTML e outros recursos como imagens, por exemplo, precisamos ter também componentes que processem informações que queremos que nossa aplicação manipule. Imagine que você vai fazer *login* na sua rede social favorita (Facebook, Instagram etc.). Você preenche um campo com seu nome de usuário, sua senha e clica no botão para acessar o sistema e então eu lhe pergunto: O que acontece? Quem que vai validar seu usuário e sua senha? Como você deve saber uma página HTML não consegue fazer isso não é mesmo? Veja a tela de *login* do Facebook na Figura 1.4.

Figura 1.4: Tela de login da rede social Facebook



Fonte: Print screen de http://www.facebook.com, acessado em 21/11/2024

O responsável em processar os dados enviados é um componente de software que é executado no servidor em que a aplicação Web está implantada. Se a aplicação é feita em PHP, um script PHP vai fazer essa validação e retornar algum resultado com base no que foi verificado. Se a aplicação for feita em ASP.NET, Node.js etc. acontece o mesmo, ou seja, algum componente vai tratar a requisição -que enviou o usuário e a senha- e vai validá-la. Em Java é a mesma coisa!

Para que possamos utilizar código Java em nossas aplicações Web, recorremos a alguns componentes que podem ser criados dentro delas, sendo que o tipo principal desses componentes é o Servlet. Você se lembra do nosso exemplo do Facebook? Os desenvolvedores do Facebook, se usassem Java, poderiam ter criado um Servlet para receber os dados do formulário de *login*, processá-los e retornar alguma resposta para o cliente, que no caso, normalmente é um navegador.

Muito bem. Sabemos então que se usarmos Java para desenvolver nossas aplicações Web, os componentes que são capazes de processar dados nas nossas aplicações e retornar resultados são os Servlets. Os Servlets são executados e gerenciados pelos Contêineres de Servlets, que funcionam como um Servidor Web simples, mas com

²Provavelmente você já sabia disso!

alguns "poderes" a mais. Esses Contêineres também podem ser componentes de infraestruturas ainda mais robustas, que no caso são os Servidores de Aplicações. Um Servidor de Aplicações é como se fosse um Contêiner de Servlets com anabolizantes, pois além de implementar toda a especificação dos Servlets e as especificações ligadas a eles, esse tipo de servidor também implementa uma série de outras especificações da plataforma Jakarta EE (*Enterprise Edition*)³, antigamente denominada Java EE, que fogem do propósito deste livro.

Da mesma forma que existem navegadores e Servidores Web diferentes, adivinhe só, existem também Servidores de Aplicações e Contêineres de Servlets diferentes! Iremos utilizar a implementação de referência das especificações do Jakarta EE 10, que é feita pelo Servidor de Aplicações Eclipse GlassFish 7.x.x. Trabalharemos especificamente com a versão 7.0.15, pois será a ideal para o que precisaremos fazer e que tem um melhor suporte no NetBeans. Quando estamos no mundo "Java para Web", várias dúvidas surgem o tempo todo, visto que existe uma infinidade de termos diferentes e que normalmente causam confusão, além de existir um ecossistema absurdamente vasto. Na próxima seção vamos aprender mais alguns detalhes teóricos e na última seção vamos realizar algumas atividades!

1.3 Servlets e JSPs

Um carro serve para dirigir. Uma televisão para assistir. Uma sandália para andar. Cada um tem suas características e vão evoluindo com o tempo. Há muito tempo, quando foi inventada a televisão, a imagem que era gerada não era muito boa e era em preto e branco. Foram passando os anos e a indústria foi evoluindo o equipamento. Primeiro a imagem melhorou, depois colocaram cores, depois foram criando telas cada vez maiores, mais finas, com maior resolução, inventaram outras formas de emitir imagens, gastar menos energia elétrica e assim por diante. E daí?

Tudo evolui. O que não evolui é descartado e/ou substituído. A primeira especificação dos Servlets foi lançada em 1997 e de lá para cá, a especificação foi evoluindo, permitindo que os Servlets se tornassem componentes mais versáteis e mais fáceis de utilizar. Na versão 7.0.15 do Eclipse GlassFish (perfil *Web*) que implementa as especificações do Jakarta EE 10 (perfil *Web*), é implementada a especificação 6.0 dos Servlets, a especificação 3.1 das JSPs (JavaServer Pages), entre outras.

Já aprendemos que os Servlets são componentes de uma aplicação Web feita em Java e que têm a capacidade de processar dados enviados a eles e gerar respostas. Um Servlet pode gerar como resposta o código HTML de uma página, entretanto essa abordagem era utilizada nas versões mais antigas dos Servlets e é totalmente

³<https://jakarta.ee/>

desencorajada hoje em dia. Como eu disse agora há pouco, tudo evolui. Para que os desenvolvedores não precisassem mais gerar código HTML dentro do código Java de um Servlet, foram inventadas as páginas JSP. Uma página JSP é um arquivo que pode conter –e geralmente contém– código HTML e que pode interagir diretamente com algumas funcionalidades de uma aplicação Web.

A rigor, um JSP é processado pelo Servidor de Aplicações e todo o seu conteúdo é traduzido em um Servlet, que por sua vez é compilado e executado pelo Servidor. Todo esse processo é realizado nos bastidores, então não precisamos nos preocupar com esses detalhes, mas é sempre bom saber um pouquinho como as coisas funcionam não é mesmo?

Por causa desse comportamento de tradução das JSPs para Servlets, nós podemos inserir código Java dentro das JSPs, mas novamente não iremos usar essa abordagem, muito menos irei ensinar como fazer, visto que da mesma forma que gerar HTML dentro de um Servlet manualmente é desencorajado, essa abordagem de inserir código Java dentro das JSPs também não deve ser utilizada. Uma JSP deve ser usada para exibir dados, não para processá-los diretamente usando código Java. Note que não estou dizendo que não iremos manipular dados dentro das JSPs, mas sim que existem formas seguras e corretas para fazer isso. Aprenderemos esses detalhes só no Capítulo 3, pois até lá, já teremos aprendido outros detalhes que ainda não foram apresentados.

1.4 Preparação do Ambiente de Desenvolvimento

Nesta seção você aprenderá a instalar e configurar a IDE Apache NetBeans e o Servidor de Aplicações Eclipse GlassFish 7.0.15 (perfil *Web*).

1.4.1 Apache NetBeans

Você conhece a linguagem Java, aprendeu a teoria e a prática da programação orientada a objetos e provavelmente usou a IDE Apache NetBeans para escrever seus códigos. Neste livro trabalharemos com a versão 24 desta IDE e nos passos abaixo é explicado como deve ser feito o *download* e a instalação da mesma. Esses passos podem ser pulados caso você já tenha essa versão da ferramenta instalada em seu computador. Além disso, na playlist disponível no *link* https://www.youtube.com/playlist?list=PLqEuQ0dDknqVcfcBHGaYrET7IBfchVS-U você encontrará tutoriais de preparação de ambientes de desenvolvimento para trabalhar com Java para Web, que serão mantidos atualizados.

- 1. Acesse o endereço: https://dlcdn.apache.org/netbeans/netbeans-installers/24/Apache-NetBeans-24-bin-windows-x64.exe;
- 2. Ao acessar esse *link*, a versão 24 do Apache NetBeans será baixada;

3. Baixou? Execute o instalador. As versões atuais dos instaladores NetBeans farão a instalação completa da IDE. Basta seguir o assistente, escolher o JDK instalado (recomendo o 17 ou o 21) e aguardar o fim da instalação.

Com o NetBeans instalado, abra-o. Caso o esteja executando pela primeira vez, uma página de boas-vindas será exibida. Você pode fechá-la se quiser, além de marcar para que não seja exibida novamente. Antes de criarmos o nosso primeiro projeto Java Web, precisamos baixar, instalar e configurar o Servidor de Aplicações Eclipse GlassFish.

1.4.2 Eclipse GlassFish

Como já dito, iremos instalar a versão 7.0.15 do Eclipse GlassFish. Iremos realizar o download e a instalação dentro do NetBeans 24. Com o NetBeans aberto, do lado esquerdo, clique na aba *Services*. Nessa aba, há um nó chamado *Servers*. Clique com o botão direito do mouse nele e escolha *Add Server...*. Veja a Figura 1.5.

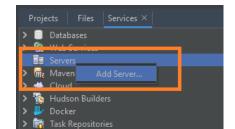


Figura 1.5: Registrando um servidor no NetBeans

Fonte: Elaborada pelo autor

Ao clicar em Add Server... o assistente para registrar o servidor no NetBeans aparecerá. No primeiro passo, mostrado na Figura 1.6, na lista de servidores suportados, escolha GlassFish Server. Na caixa de texto Name: edite o nome da instância do servidor. No meu caso, adicionei a versão do mesmo, ficando "GlassFish Server 7.0.15". Ao fazer isso, clique em Next > .

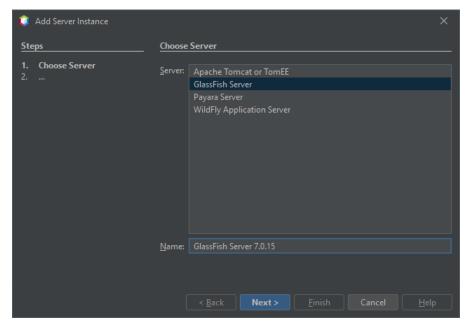


Figura 1.6: Registrando um servidor no NetBeans - Passo 1

O segundo passo do assistente precisamos definir onde os arquivos do servidor ficarão, como apresentado na Figura 1.7. Nesse passo, clique no botão *Browse...* e defina onde a instalação ficará. No meu caso, escolhi a pasta glassfish7015 dentro da pasta do meu usuário. Recomendo que você faça o mesmo. Deixa a opção *Local Domain* selecionada, escolha a versão do servidor para download (GlassFish Server 7.0.15), marque a opção "*I have read...*" e clique em *Download Now*. Ao finalizar o download, uma mensagem logo abaixo do assistente indicará que foi encontrada uma instalação do servidor no local escolhido, então basta clicar em *Next* > para realizarmos o terceiro e último passo do assistente.

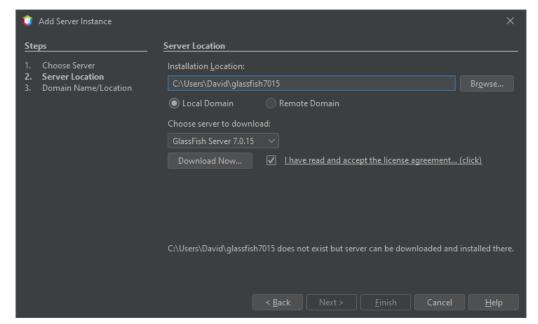


Figura 1.7: Registrando um servidor no NetBeans - Passo 2

Nesse passo, faremos a última configuração necessária, que consiste apenas em definir o nome do usuário administrador do servidor. Na Figura 1.8 isso pode ser visto na caixa de texto *User Name:* que está preenchida com "admin". Ao preencher o nome de usuário, clique em *Finish*.

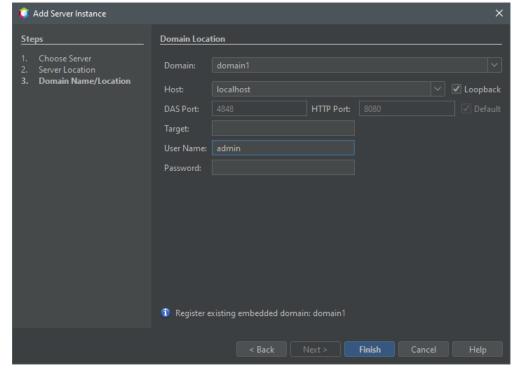


Figura 1.8: Registrando um servidor no NetBeans - Passo 3

Ao fazer isso, o GlassFish estará quase pronto para ser utilizado. A última coisa que precisaremos fazer é copiar o *Driver Java Database Conectivity* (JDBC) do Sistema Gerenciador de Banco de Dados (SGBD) MariaDB/MySQL que usaremos nos próximos Capítulos deste livro.

Para isso, baixe o arquivo acessível através do *link* https://repo1.maven.org/maven2/org/mariadb/jdbc/mariadb-java-client/3.5.0/mariadb-java-client-3.5.0.jar e, ao terminar de baixar, copie-o para o diretório C:\Users\SeuUsuario\glassfish7015\glassfish\domains\domain1\lib\. Agora sim, tudo pronto!

Caso haja alguma dúvida, visite o *link* mencionado que contém uma playlist de tutoriais para a configuração de ambientes de desenvolvimento.

1.4.3 Primeiro Projeto Java para Web

Agora que temos tudo configurado, iremos criar nosso primeiro projeto Java para Web! Siga os passos abaixo:

Passo 1: Clique no menu [File] e depois em [New Project...]. Fazendo isso, o
assistente para criação de projetos será aberto. Na lista de categorias, expanda o

- item $[Java\ with\ Ant]$ e escolha $[Java\ Web]$. Na lista de tipos de projeto, escolha $[Web\ Application]$ e clique no botão [Next>];
- **Passo 2:** Preencha o campo (*Project Name:*) com "OlaMundoWeb" (sem acentos, sem as aspas e tudo junto). Em (*Project Location:*), defina o diretório onde o projeto será salvo. Deixe a opção (*Use Dedicated Folder for Storing Libraries*) marcada e clique no botão (*Next* >);
- **Passo 3:** Na opção (*Server:*) escolha o "GlassFish Server 7.0.15", ou a opção com o nome que você definiu ao registrar o GlassFish. Em [*Java EE Version:*] escolha "Jakarta EE 10 Web". Em [*Context Path:*] deixe o valor padrão (/OlaMundoWeb), que é o mesmo nome que demos ao nosso projeto. Clique em [*Next* >];
- Passo 4: No último passo, o assistente perguntará quais frameworks nós queremos inserir no nosso projeto. Nós não vamos usar nenhum, então basta clicar em Finish. Fazendo isso, o novo projeto será criado e será aberto no NetBeans, sendo que por padrão será criado um arquivo HTML (index.html) que será a página inicial da nossa aplicação.

(i) | Saiba Mais

Existem diversas definições para *framework*, sendo que, informalmente, podemos definí-los como um conjunto de classes que incorporam uma abstração que tem como objetivo de solucionar problemas de um tipo ou domínio específico.

Muito bem, criamos nosso primeiro projeto. Vamos executá-lo para ver o que acontece? Na barra de ferramentas do NetBeans tem um botão com uma seta verde, igual a um botão de "play" de um reprodutor de mídias. Quando você clicar nesse botão, você vai ver que várias mensagens começarão a aparecer na janela de saída do NetBeans. Essas mensagens irão mostrar para nós o que está acontecendo no momento, como a inicialização do GlassFish (caso não esteja iniciado) etc. O que está esperando? Clique lá no "play". Assim que tudo estiver pronto, será aberta uma janela do seu navegador, onde será mostrado o conteúdo do index.html, que no nosso caso será uma página com "TODO write content" escrito.

Muito bem! Temos nossa primeira aplicação rodando no GlassFish! Fácil não é mesmo? Por enquanto não vamos nos preocupar com a estrutura do projeto, iremos aprender os detalhes aos poucos. Vamos colocar um pouco de código HTML no nosso index.html? Ele deve estar aberto no NetBeans. Se não estiver, procure o arquivo index.html na pasta *Web Pages* do seu projeto e clique duas vezes no arquivo para abri-lo no editor. Vamos mudar o título, escrevendo "Meu Primeiro Projeto Java para Web" no lugar de "TODO supply a title" e dentro da *tag body* do arquivo, inserir um

heading (h1>) e um parágrafo com um link para o site do IFSP. Veja na Listagem 1.1 como deve ficar seu código.

```
Listagem 1.1: Arquivo index.html
   <!DOCTYPE html>
2
  <html>
       <head>
4
           <title>Meu Primeiro Projeto Java para Web</title>
           <meta charset="UTF-8">
6
           <meta name="viewport" content="width=device-width, initial-scale=1.0">
7
       </head>
8
       <body>
9
           <h1>01á Mundo!</h1>
10
11
               <a href="http://www.ifsp.edu.br/">Site do IFSP</a>
12
           13
       </body>
14
  </html>
15
```

Salve o arquivo depois de editá-lo. Se o navegador ainda estiver aberto no index.html, volte a ele e aperte a tecla <F5> do seu teclado para mandar o navegador atualizar a página. Se não estiver, dê o "*play*" no projeto de novo. Você vai ver que a página vai exibir as alterações que fizemos. Teste o *link* para ver se está funcionando. A página deve ter ficado como mostrada na Figura 1.9.

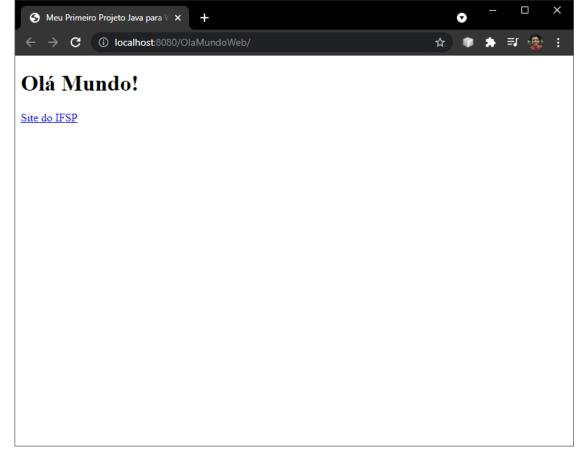


Figura 1.9: Arquivo index.html em exibição

Poderíamos ter usado um arquivo *JavaServer Pages* (JSP) ao invés de usar um arquivo HTML, permitindo a existência de outros tipos de estruturas que vamos aprender no decorrer do livro, mas por enquanto vamos manter o HTML. Vamos testar os Servlets agora? Como primeiro exemplo, nós vamos criar um Servlet manualmente, enquanto os outros que vamos desenvolver durante o nosso curso serão feitos usando um assistente do NetBeans, mas essa forma fácil nós só vamos aprender a partir do Capítulo 2.

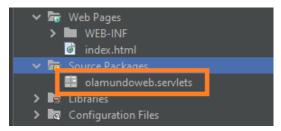
Aprenderemos como criar manualmente um Servlet, para que possamos aprender alguns detalhes importantes sobre o funcionamento de aplicações Web feitas em Java. Siga os passos abaixo:

• **Passo 1:** Na árvore que representa a estrutura do projeto, procure pela pasta *Source Packages*) e expanda-a (clique no sinal de "+" à esquerda). Dentro dela

haverá um pacote com o ícone cinza chamado (*default package*). Como vocês devem saber, é desencorajado que se trabalhe com pacotes padrão em Java, então vamos criar um pacote. Clique com o botão direito na pasta *Source Packages* e escolha *New*, procure pela opção *Java Package*... e clique nela. Se esta opção não estiver sendo exibida, clique na opção *Other*... (no final da lista), escolha *Source Packages* nas categorias, e *Java 'Package* nos tipos de arquivos e clique em *Next* > . Preencha o campo *Package Name*: com "olamundoweb" (sem as aspas) e clique em *Finish*). O pacote será criado;

• Passo 2: Repita o Passo 1, só que agora clicando com o botão direito no pacote que você criou e crie um pacote chamado "servlets" (sem as aspas). O nome do pacote deverá ser preenchido com "olamundoweb.servlets". Seu projeto agora terá um pacote chamado "olamundoweb.servlets". O resultado desses dois primeiros passos podem ser vistos na Figura 1.10;

Figura 1.10: Criação do pacote olamundoweb. servlets



Fonte: Elaborada pelo autor

• Passo 3: Clique com o botão direito no pacote olamundoweb.servlets, escolha New e clique na opção Java Class.... Novamente, se não a encontrar, clique em Other... e procure por Java Class... (está na categoria "Java") e clique em Next > . Preencha o campo Class Name: com "OlaServlet" (sem as aspas) e clique em Finish. A classe será criada dentro do pacote especificado e será aberta no editor. Você vai ter algo como apresentado na Listagem 1.2 (sem os comentários);

```
Listagem 1.2: olamundoweb/servlets/OlaMundoWeb.java

package olamundoweb.servlets;

public class OlaServlet {

}
```

• Passo 4: Para que uma classe seja um Servlet, precisamos estender a classe HttpServlet, que está contida no pacote jakarta.servlet.http4 e então implementar os métodos HTTP que queremos que nosso Servlet trate. Não se preocupe, ainda vamos aprender sobre os métodos HTTP, então o mais importante a saber, por enquanto, é que os métodos HTTP mais usados são o GET (doGet(...) de HttpServlet) e o POST (doPost(...) de HttpServlet). Então teremos que sobrescrever cada um desses métodos e ainda criaremos um terceiro que será invocado a partir dos outros dois. Confuso? Vamos ver como o código ficaria. Leia os comentários e copie o código da Listagem 1.3 para o seu editor.

```
Listagem 1.3: olamundoweb/servlets/OlaMundoWeb.java
  package olamundoweb.servlets;
2
  import java.io.IOException;
4 import jakarta.servlet.ServletException;
  import jakarta.servlet.annotation.WebServlet;
6 import jakarta.servlet.http.HttpServlet;
7 import jakarta.servlet.http.HttpServletRequest;
  import jakarta.servlet.http.HttpServletResponse;
9
10
11
   * Nosso primeiro Servlet!
12
13
   * A anotação @WebServlet é usada para indicar que esta classe
14
15
    * é um Servlet, configurando seu nome o padrão de URL que
   * será associado à esse componente.
16
17
   * @author Prof. Dr. David Buzatto
18
19
  @WebServlet( name = "OlaServlet", urlPatterns = { "/ola" } )
  public class OlaServlet extends HttpServlet {
22
       /**
23
        * Método para tratar requisições que usam o método
24
        * GET do protocolo HTTP. É um método herdado da
25
        * classe HttpServlet que precisa ser sobrescrito.
26
```

⁴O Jakarta EE 10 usa o novo *namespace* do Java EE, onde o pacote base é o jakarta. Antes da versão 9 o namespace base era javax.

```
27
        * A anotação @Override indica ao compilador que
28
        * estamos sobrescrevendo um método herdado da
        * classe que está sendo estendida, no caso
        * HttpServlet.
31
        * @param request Referência ao objeto que contém
        * os dados da requisição.
        * @param response Referência ao objeto que conterá
35
        * os dados da resposta.
36
37
38
        * @throws ServletException
        * @throws IOException
        */
       @Override
41
       protected void doGet(
42
               HttpServletRequest request,
43
44
               HttpServletResponse response )
               throws ServletException, IOException {
46
           // chama o método processRequest
47
           processRequest( request, response );
48
49
       }
50
       /**
        * Método para tratar requisições que usam o método
53
        * POST do protocolo HTTP.
54
55
        * @param request Referência ao objeto que contém
        * os dados da requisição.
        * @param response Referência ao objeto que conterá
        * os dados da resposta.
60
        * @throws ServletException
61
        * @throws IOException
        */
       @Override
       protected void doPost(
65
               HttpServletRequest request,
66
               HttpServletResponse response )
67
               throws ServletException, IOException {
68
```

```
69
           // chama o método processRequest
70
           processRequest( request, response );
71
72
       }
73
74
       protected void processRequest(
75
                HttpServletRequest request,
76
                HttpServletResponse response )
77
                throws ServletException, IOException {
78
79
80
             * Aqui vem o código que queremos que o
             * nosso Servlet execute.
82
83
           System.out.println( "Ola Mundo!" );
84
           System.out.println( "Meu Primeiro Servlet!" );
85
86
       }
88
  }
89
```

Até agora criamos uma classe chamada OlaServlet, que estende a classe HttpServlet. Sobrescrevemos os métodos doGet(...) e doPost(...) herdados de HttpServlet que tratam respectivamente os métodos GET e POST do protocolo HTTP e criamos um terceiro método, chamado processRequest(...), que tem a mesma assinatura dos métodos doGet(...) e doPost(...) e que é invocado dentro deles. É no processRequest que iremos colocar o código que queremos executar, sendo que no nosso exemplo, estamos mandando imprimir na saída duas Strings: "Olá Mundo!" e "Meu Primeiro Servlet!". Ou seja, se chamarmos o Servlet usando o método GET, o método doGet(...) será invocado e passará o controle para o método processRequest(...) que irá imprimir as mensagens na saída. O mesmo acontece para o método POST.

Muito bem, você tem um Servlet totalmente funcional, mas ai você se pergunta: "Como vou chamar esse Servlet através do navegador?". Então eu respondo: no código completo, você percebeu que há uma anotação chamada @WebServlet ? É

essa anotação que vai fornecer essa informação⁵, expecificamente no parâmetro urlPatterns. Perceba que configuramos esse parâmetro com um array de Strings com um elemento: "/ola". Com isso, podemos agora acessar o OlaServlet a partir de uma URL, que no nosso caso é "", ou seja, usamos o protocolo HTTP, para a máquina localhost (que é o endereço da nossa máquina), na porta 8080 (que é a porta que o GlassFish ouve por padrão), para acessar a aplicação chamada OlaMundoWeb (isso vem do contexto que criamos no Passo 3 da Subseção 1.4.3, volte lá para dar uma olhadinha), para por fim acessar o recurso mapeado sob o nome de "ola" que no caso é o nosso Servlet.

Sei que pode parecer um pouco confuso no começo, mas logo você vai pegar o jeito da coisa. Dê um "play" no projeto de novo. O navegador vai abrir no endereço da aplicação novamente. Insira o "ola" (sem as aspas) no final da URL e tecle <ENTER>. O que aconteceu? Apareceu uma página em branco não foi? É claro, afinal, nosso Servlet não gera HTML, mas apenas imprime duas mensagens na saída padrão não é mesmo? Mas como podemos ver essas mensagens? Volte no NetBeans e procure, logo abaixo, uma aba chamada (Output). Ela provavelmente vai estar selecionada. Dentro dela devem haver outras três abas: OlaMundoWeb (run) que deve estar selecionada e que é usada para mostrar o processo de construção do projeto da nossa aplicação, *Java DB Database Process*, que exibe o *status* do Java DB e, por fim, GlassFish Server 7.0.15, que exibe a saída padrão do GlassFish. Clique nessa última aba e veja o que está escrito lá embaixo: as duas mensagens que enviamos para a saída padrão através do método System.out.println(...) dentro do Servlet, sendo que o fim de cada linha conterá os caracteres [#]⁶! Veja o resultado na Figura 1.11. Volte ao navegador e tecle <ENTER> novamente no endereço do Servlet. Volte no NetBeans. Mais duas mensagens! Fácil não é mesmo?

⁵Antigamente precisávamos fazer o mapemanto em um arquivo *Extensible Markup Language* (XML) (http://pt.wikipedia.org/wiki/XML) chamado de Descritor de Implantação (DI, em inglês *Deployment Descriptor*), representado pelo arquivo web.xml, o que atualmente, com as versõs mais novas do Jakarta/Java EE, não é mais necessário para algumas situações.

⁶Esse sufixo é inserido automaticamente pelo servidor.

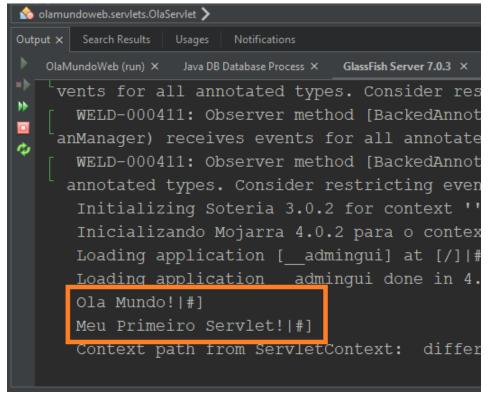


Figura 1.11: Saída do GlassFish 7.0.15

Por mais que nosso exemplo não tenha nenhuma utilidade aparente, ele foi importante para nós entendermos o funcionamento básico de uma aplicação Web feita em Java. Nos próximos Capítulos vamos colocar o que aprendemos em prática, além de aprender várias outras coisas com o objetivo de criarmos um sistema de cadastro na forma de uma aplicação Web. Não se esqueça de praticar o que aprendemos até agora.

1.5 Resumo

Neste Capítulo aprendemos o que é e como funciona uma aplicação Web em Java. Aprendemos a criar nosso primeiro projeto e alguns detalhes sobre a tecnologia que estamos utilizando. Executamos nossa aplicação e fizemos algumas modificações nela para vermos o que estava sendo feito. Criamos também –de forma manual– um Servlet, que como aprendemos é um dos componentes principais de uma aplicação Web em Java.

1.6. EXERCÍCIOS 23

1.6 Exercícios

Exercício 1.1: O que é um Servidor Web?

Exercício 1.2: Como são chamados os clientes que utilizamos para acessar aplicações servidas por um Servidor Web? Cite alguns exemplos.

Exercício 1.3: Diferencie um Servidor Web de um Contêiner de Servlets.

1.7 Projetos

Projeto 1.1: Crie um novo projeto Java Web no NetBeans, com o nome de "MinhaPagina", edite o index.html de modo a exibir seus dados pessoais, seus interesses etc. Tente inserir uma imagem também. Dica: a imagem deve estar dentro do projeto do NetBeans. Pense se você entende o motivo pelo qual o arquivo index.html é mostrado por padrão quando você acessa sua página através da URL <HTTP://localhost: 8080/MinhaPagina>.

Projeto 1.2: Crie um novo projeto Java Web no NetBeans, com o nome de "Contador". Nesse projeto você deve criar um Servlet manualmente e dentro do método processRequest(...) use uma estrutura de repetição para direcionar para a saída padrão os números de 1 a 30.

Projeto 1.3: Crie um novo projeto Java Web no NetBeans, com o nome de "Fibonacci". Nesse projeto, você deve criar um Servlet manualmente e dentro do método processRequest(...) use uma estrutura de repetição para exibir os 30 primeiros termos da série de Fibonacci. Crie um método chamado fibonacci dentro do seu Servlet, sendo que este método deve receber como parâmetro um inteiro e retornar um inteiro. O inteiro que é recebido como parâmetro é o número do termo desejado, enquanto o inteiro que é retornado é o termo correspondente ao parâmetro que foi recebido. A série de Fibonacci é formada inicialmente pelos números 1 e 1, sendo que os próximos números da série são gerados a partir da soma dos dois números anteriores. Os sete primeiros termos da série de Fibonacci são 1,1,2,3,5,8,13, onde: 2 = 1 + 1, 3 = 1 + 2, 5 = 2 + 3, 8 = 3 + 5, 13 = 5 + 8.

Exemplos de chamadas da função fibonacci:

- fibonacci(2): retorna 1
- fibonacci(5): retorna 5
- fibonacci(7): retorna 13

PROCESSAMENTO DE FORMULÁRIOS

"O modo como você reúne, administra e usa a informação determina se vencerá ou perderá".

Bill Gates



ESTE Capítulo teremos como objetivos entender o funcionamento de formulários HTML e conseguirmos diferenciar os métodos do protocolo HTTP e como tratá-los.

2.1 Introdução

Neste Capítulo vamos começar a aprender a criar algo útil! No Capítulo 1, aprendemos as bases do desenvolvimento Web em Java, criamos alguns programas de brinquedo para aplicar as técnicas que aprendemos e agora vamos aprender mais alguns detalhes, mas dessa vez nossos programas serão mais elaborados. Vamos começar?

¹Programa de brinquedo é todo programa criado para apresentar algum conceito e que, normalmente, não tem uma utilidade prática além da pedagógica.

2.2 Formulários

A forma tradicional de se desenvolver aplicações para Web que interajam com o servidor é utilizando os chamados formulários. Um formulário é composto normalmente por um conjunto de componentes que permitem que o usuário forneça dados para serem submetidos ao servidor. Quando esses dados são recebidos pelo servidor, algum componente da aplicação vai tratá-los, sendo que no nosso caso, esse componente vai ser implementado na forma de um Servlet.

Atualmente existem diversas técnicas para a criação de aplicações Web, sendo que, dependendo da técnica/tecnologia, a forma de submeter dados ao servidor é diferente. Uma dessas técnicas é o chamado *Asynchronous JavaScript and XML* (AJAX) que hoje em dia é implementado de inúmeras formas. Neste livro aprenderemos sobre AJAX no Capítulo 7.

Abra o NetBeans e crie um novo projeto Java Web. Dê o nome de "PrimeiroFormulario" (sem as aspas). Sempre que for criar um novo projeto, siga os passos descritos na Subseção 1.4.3 do Capítulo 1. Com o projeto criado, vamos editar o index.html. Nele vamos alterar o título da página e criar nosso primeiro formulário, que será usado para preenchermos dados pessoais de um cliente. Veja na Listagem 2.1 como ficou o código. Não se esqueça de copiá-lo no seu index.html.

```
Listagem 2.1: Protótipo do formulário de dados do cliente (index.html)
   <!DOCTYPE html>
2
  <html>
3
       <head>
4
           <title>Meu Primeiro Formulário</title>
5
           <meta charset="UTF-8">
6
           <meta name="viewport" content="width=device-width, initial-scale=1.0">
7
       </head>
8
       <body>
9
10
           <h1>Dados Pessoais do Cliente</h1>
11
12
           <form>
13
14
               <label>Nome: </label>
15
               <input type="text" size="20" name="nome"/>
16
               <br/>
17
18
```

2.2. FORMULÁRIOS 27

```
<label>Sobrenome: </label>
19
               <input type="text" size="20" name="sobrenome"/>
20
               <br/>
21
22
               <label>CPF: </label>
23
               <input type="text" size="15" name="cpf"/>
24
               <br/>
               <label>Data de Nascimento: </label>
27
               <input type="text" size="10" name="dataNasc"/>
28
               <br/>
29
30
               <label>Sexo: </label>
               <input type="radio" name="sexo" value="M"/> Masculino
               <input type="radio" name="sexo" value="F"/> Feminino
33
               <br/>
34
35
               <label>Observações: </label>
36
               <br/>
               <textarea cols="40" rows="10" name="observacoes"></textarea>
               <br/>
40
               <input type="submit" value="Enviar Dados"/>
41
42
           </form>
       </body>
45
46 </html>
```

Copiou? Salve o arquivo e execute a aplicação. Você vai ter algo como o mostrado na Figura 2.1.

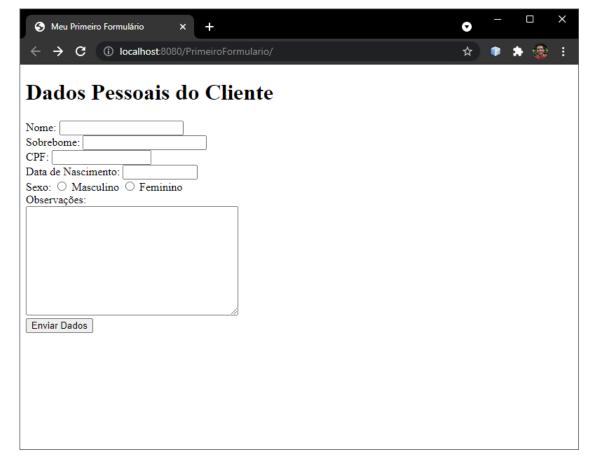


Figura 2.1: Visualização do protótipo do formulário de dados do cliente

Fonte: Elaborada pelo autor

Quanta coisa! O formulário não ficou uma obra prima, mas esse não é nosso objetivo agora. Precisamos entender o que cada *tag* faz. Vamos agora analisar o código da Listagem 2.1 e entender o protótipo que fizemos. Irei detalhar apenas as *tags* <form> e seus componentes, pois acredito que você já conheça as outras que foram utilizadas. Vamos lá então:

- **Linha 13:** Nesta linha abrimos a *tag* (form) que delimita um formulário HTML. Note que fechamos a *tag* (form) na linha 43. Todas as *tags* que forem inseridas entre (form) e (/form) farão parte do formulário;
- **Linha 15:** Criamos um *label* (tag <label>) com o conteúdo "Nome: ". A *tag* <label>) é usada para criar um rótulo. Ao invés de usar a *tag* <label>), poderíamos simplesmente ter inserido o texto que queremos mostrar no formulário, mas como o texto que estamos utilizando tem o propósito de ser um rótulo para

- um campo do formulário, iremos utilizar essa tag para deixar nosso código mais organizado e inserir uma certa carga semântica no nosso código;
- Linha 16: Criamos um *input* (campo de entrada) do tipo *text* (texto) com tamanho de 20 colunas e com o nome de "nome". Dentre as *tags* que representam componentes nos formulários, a <input> é uma delas. Existem vários tipos de *inputs*, diferenciados pela propriedade type, e que vamos aprender aos poucos. A propriedade size, como você deve ter percebido, é utilizada para configurar a largura do campo de texto. A propriedade name é utilizada pelo navegador para identificar os dados do componente em questão no momento de enviar os dados para o servidor. Não entendeu a utilidade da propriedade name? Não se preocupe, logo vai fazer sentido;
- **Linha 17:** Usamos a *tag* (br/>) para pular uma linha;
- Linhas 19 a 29: Os próximos três campos (sobrenome, CPF e data de nascimento) são bem parecidos com o primeiro;
- **Linha 32:** Criamos um *input* do tipo *radio* (botão de rádio), com nome configurado como "sexo" e com o valor (value) configurado com "M";
- Linha 33: Idem à linha anterior, com a diferença que o valor é "F". Note que a propriedade name de ambos os radios é a mesma, pois eles representam o mesmo campo (sexo). Perceba que no navegador, se você selecionar um deles e depois clicar no outro, o que estava selecionado previamente deixa de ser selecionado. Se a propriedade name for diferente, eles serão considerados campos diferentes e então esse comportamento da seleção não existirá. Note ainda que você pode "amarrar" quantos radios você precisar;
- Linha 38: Nessa linha definimos uma área de texto. Esse componente, representado pela tag <textarea> é utilizado, como o próprio nome já diz, para criar uma área de texto livre, onde o usuário poderá digitar uma quantidade arbitrária de texto. Note que para utilizar um <textarea> nós precisamos usar a tag de fechamento (</textarea>), ao invés de fazer da forma que estamos fazendo com os inputs. A novidade nesse componente são as propriedades cols e rows, que são usadas respectivamente para definir a quantidade de colunas e de linhas do componente;
- **Linha 41:** Por fim, nessa linha definimos um *input* do tipo *submit*, que é um botão que tem o comportamento padrão de, ao ser clicado, submeter (enviar) os dados do formulário para o servidor. Note que usamos a propriedade value para definir o texto do botão.

Agora que já conhecemos alguns dos componentes que podemos utilizar nos nossos formulários, mas você deve estar se perguntando: "Tudo bem, o *input* do tipo *submit* é usado para enviar os dados do formulário para o servidor, mas onde eu digo ao *submit* para onde os dados do formulário devem ser enviados?". Vamos à resposta!

Eu tenho dito várias vezes que o componente que vai tratar os dados de um formulário na nossa aplicação é o Servlet não é mesmo? Então precisamos criar um Servlet que vai receber esses dados e então configurar o formulário para direcionar os dados inseridos nele para o Servlet apropriado.

Preencha o campo *Class Name*: com "ProcessaDadosClienteServlet" (sem as aspas) e clique em *Next* >. Nesse passo, note que o assistente nos pede o nome do Servlet (*Servlet Name*:) e o(s) padrão(ões) de URL (*URL Pattern(s)*:). Além disso, há a opção de inserir esses dados no descritor de implantação (*deployment descriptor*), mas como estamos trabalhando com anotações para fazer o mapeamento do Servlet, essa opção deve ficar desmarcada. Falando sobre a anotação (*@WebServlet*), ao preenchermos esses campos, o NetBeans vai inserí-la para nós de forma automática! Deixe o campo *Servlet Name*: com o valor padrão (que é o nome da classe) e preencha o campo *URL Pattern(s)*: com "/processaDadosCliente" (sem as aspas). Não iremos aprender sobre os parâmetros de inicialização (*initialization parameters*) neste livro, mas nada impede que você aprenda para que eles servem, basta consultar a bibliografia recomendada nas referências bibliográficas do livro tudo bem? Tudo feito? Clique em *Finish*.

Ao fazer isso, o nosso Servlet será criado e aberto no editor. Veja que a anotação <code>@WebServlet</code> foi inserida apropriadamente no código da classe! Legal hein? Note também que o NetBeans já implementou o esqueleto do Servlet para nós. O primeiro método implementado é o <code>processRequest(...)</code>. Lembra-se dele? É nele que vamos inserir o código que queremos que o Servlet execute. Após o fechamento do bloco deste método, note que existe uma linha onde está escrito "HttpServlet methods. Click on the + sign on the left to Edit the code.". Siga a sugestão da frase e clique no "+". O que apareceu? A implementação dos métodos GET e POST, sendo que dentro delas é chamado o <code>processRequest(...)</code>! Viu só? Da mesma forma que fazíamos manualmente! Não iremos mexer ali, então você pode contrair novamente esta seção do código clicando no sinal de "-".

Note que além de implementar o esqueleto do nosso Servlet, o NetBeans também inseriu um trecho de código dentro do processRequest(...). O código que está inserido configura o que o Servlet gerará de resposta a quem o requisitou

2.2. FORMULÁRIOS 31

(response.setContentType(...)), obtém o fluxo de escrita do Servlet, escreve uma série de Strings que representam uma página HTML nesse fluxo e o fecha automaticamente, dado o uso do *try with resources*. Você se lembra que já falei algumas vezes que não iremos implementar Servlets que geram código HTML? Então, vamos limpar esse método, tirando todo o código que foi inserido dentro dele. Vá no editor e apague o conteúdo entre as linhas 34 (response.setContentType(...)) e 46 (}), incluindo elas. Seu processRequest(...) deve estar vazio agora.

Antes de escrevermos nosso código, vamos a mais um pouquinho de teoria. Você se lembra, lá no comecinho do Capítulo 1, que eu falei que o cliente manda uma requisição para o servidor e ele manda uma resposta? Nos Servlets, essa requisição e essa resposta são representadas respectivamente por objetos do tipo HttpServletRequest e HttpServletResponse. Note que os três métodos implementados nos nossos Servlets (processRequest(...), doGet(...) e doPost(...) recebem dois parâmetros, sendo eles dos tipos que mencionei. Qual a conclusão que você chega então? Os dados enviados pelo cliente ao servidor estão dentro do objeto do tipo HttpServletRequest, que chamamos de request no nosso código e os dados que enviamos de volta ao cliente, ou a quem invocou o Servlet, devem ser inseridos no objeto do tipo HttpServletResponse, que demos o nome de response.

Sabendo disso, agora ficou fácil! Os dados do formulário de clientes que estamos construindo serão recebidos pelo nosso Servlet através do objeto apontado por request! Legal, mas ainda falta um detalhe... Não informamos ao navegador qual o destino dos dados do formulário! Vamos fazer isso agora. Volte no index.html e procure pela tag <form> (a mesma que está na linha 13 da Listagem 2.1). Para informarmos ao navegador qual o destino do dados do formulário, utilizamos a propriedade action, sendo que nesta propriedade, colocamos a URL do componente que deve tratar os dados do formulário. Mapeamos nosso Servlet usando o padrão /processaDadosCliente não foi? Então, qual será a URL do nosso Servlet? Resposta: processaDadosCliente. Vamos editar nossa tag <form> então, configurando a propriedade action para a URL citada. Na Listagem 2.2 você pode ver como ficou o código da tag <form>. Note que não estou listando todo o arquivo.

Isso quer dizer que, quando clicarmos no botão "Enviar Dados" (que é um input

do tipo submit), os dados que estiverem nos componentes que estão dentro desse formulário serão enviados para o recurso processaDadosCliente, que é o endereço do nosso Servlet! Já alterou o arquivo? Salvou? Legal! Atualize a página, preencha os campos e clique no botão "Enviar Dados". O que aconteceu? Uma página em branco foi exibida não é? E na barra de endereços, o que apareceu? O endereço do Servlet mais um monte de "coisas"! Os detalhes sobre isso nós iremos aprender na próxima seção, então não se preocupe por enquanto.

Você se lembra dos nossos primeiros exemplos? Acessávamos o endereço do Servlet, uma página em branco era exibida e duas Strings eram direcionadas para a saída do servidor, lembra? Quem fazia esse direcionamento era o Servlet, não era? Vamos fazer a mesma coisa com o nosso Servlet de dados dos clientes, mas mostraremos os dados que foram preenchidos nos formulários. Vamos lá então?

Volte ao Servlet, vamos implementar o método (processRequest(...)). Qual é mesmo o nome do parâmetro do método (processRequest(...)) que armazena os dados enviados pelo cliente? É o request certo? Veja o código da Listagem 2.3. Leia todos os comentário que fiz.

```
Listagem 2.3: Implementação do método processRequest
  (ProcessaDadosClienteServlet.java)
  package primeiroformulario.servlets;
3 import java.io.IOException;
4 import jakarta.servlet.ServletException;
5 import jakarta.servlet.annotation.WebServlet;
6 import jakarta.servlet.http.HttpServlet;
7 import jakarta.servlet.http.HttpServletRequest;
  import jakarta.servlet.http.HttpServletResponse;
9
  /**
10
   * Servlet para processamento dos dados do formulário.
11
12
13
   * @author Prof. Dr. David Buzatto
14
  @WebServlet( name = "ProcessaDadosClienteServlet",
15
              urlPatterns = { "/processaDadosCliente" } )
16
  17
18
      protected void processRequest(
19
             HttpServletRequest request,
20
```

2.2. FORMULÁRIOS 33

```
HttpServletResponse response )
21
               throws ServletException, IOException {
22
23
           /* Precisamos manter de forma consistente
24
            * o mesmo encoding em todas as camadas da
25
            * aplicação, evitando assim problemas com
26
            * caracteres acentuados. Aqui informamos
            * que a requisição está chegando codificada
28
            * em UTF-8.
29
            */
30
           request.setCharacterEncoding( "UTF-8" );
31
32
           /* Obtém os dados do request.
            * O método getParameter de request obtém
            * um parâmetro enviado pelo formulário
35
            * que acessou o Servlet.
36
37
            * O parâmetro tem SEMPRE o mesmo nome configurado
38
            * na propriedade "name" do componente do formulário.
40
           String nome = request.getParameter( "nome" );
41
           String sobrenome = request.getParameter( "sobrenome" );
42
           String CPF = request.getParameter( "cpf" );
43
           String dataNascimento = request.getParameter( "dataNasc" );
44
           String sexo = request.getParameter( "sexo" );
           String observacoes = request.getParameter( "observacoes" );
47
           System.out.println( "Dados do Cliente:" );
48
           System.out.println( "Nome: " + nome );
49
           System.out.println( "Sobrenome: " + sobrenome );
           System.out.println( "CPF: " + CPF );
           System.out.println( "Data de Nascimento: " + dataNascimento );
52
53
           if ( sexo.equals( "M" ) ) {
54
               System.out.println( "Sexo: Masculino" );
           } else {
               System.out.println( "Sexo: Feminino" );
           }
59
           System.out.println( "Observacoes: " + observacoes );
60
61
       }
62
```

```
63
       @Override
64
       protected void doGet(
65
                HttpServletRequest request,
66
                HttpServletResponse response )
67
                throws ServletException, IOException {
68
           processRequest( request, response );
69
       }
70
71
       @Override
72
       protected void doPost(
73
74
                HttpServletRequest request,
                HttpServletResponse response )
75
                throws ServletException, IOException {
76
           processRequest( request, response );
77
       }
78
79
       @Override
80
       public String getServletInfo() {
           return "ProcessaDadosClienteServlet";
82
       }
83
84
  }
85
```

Copiou o código? Legal. Vamos entender o que está acontecendo. Primeiramente, na linha 31, para mantermos a consistência da codificação em que os dados da nossa aplicação estão trafegando entre as camadas, configuramos o request para processar seus dados usando o encoding UTF-8. Mais adiante no livro aprenderemos a criar um filtro que fará isso automaticamente para todos os nossos Servlets, mas por enquanto vamos fazer um a um, da forma que está no código.

Na linha 41 é declarada uma variável do tipo String com o nome de "nome". Essa variável é inicializada com o valor obtido ao se chamar o método getParameter(String param) de request. O parâmetro passado ao método getParameter(String param), que é uma String, é o nome que foi dado ao componente do formulário, que no caso foi "nome". Estude as linhas 42, 43, 44, 45 e 46 e tente fazer um paralelo com o formulário contido no index.html. Note que a String que é passada como parâmetro no método getParameter(...) sempre reflete o nome dado ao componente do formulário através da propriedade name. Veja a linha 44. Declaramos uma variável chamada dataNascimento que é inicializada com o valor do parâmetro "dataNasc" (configurado no formulário).

O que fizemos até a linha 46 foi criar uma variável que vai receber o valor de cada componente do formulário. A partir da linha 48, até o final do método, direcionamos para a saída os dados que foram obtidos. Vamos testar? Salve o Servlet e execute o projeto (botão de "play", lembra?). Na página, preencha o formulário, clique em "Enviar Dados" e volte no NetBeans para ver o que aconteceu. Olhe na janela de saída! Lá estão os dados que você preencheu no formulário! Muito bem! Imagine agora se esse fosse um sistema real. Esses dados recebidos dentro do Servlet poderiam alimentar uma *query* SQL para inserir esse cliente em um banco de dados! Legal não é?! A partir desse ponto, você já deve estar entendendo melhor o que está acontecendo, mas e aquele monte de coisas escritas no endereço do navegador depois de clicar em "Enviar Dados"? Esse comportamento está intrinsecamente ligado ao tipo de método HTTP que estamos usando. Vamos para a próxima Seção, vou explicar isso lá.

2.3 Métodos HTTP

O protocolo HTTP que usamos em nossas aplicações Web, define uma série de métodos que podem ser usados para tratar diversos tipos de requisições. Na nossa vida, como desenvolvedores Web, iremos nos importar com vários desses métodos. Neste Capítulo trataremos os métodos GET e POST, que são os únicos que podem ser usados em formulários HTML. Sendo assim vamos a eles.

(i) | Saiba Mais

Quer conhecer os outros métodos HTTP como HEAD, PUT, DELETE etc.? Acesse este link: https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Methods

2.3.1 Método GET

O método GET (to get = obter) é usado principalmente para pedir ao servidor algum recurso, sendo que ele retornará os dados requisitados, caso existam, claro. Quando nós fazemos uma pesquisa no Google ou clicamos em um link, estamos usando o método GET. Qualquer URL que colocamos na barra de endereço do nosso navegador é enviada ao servidor usando o método GET. Vamos fazer um teste. Entre na página do Google (<www.google.com>) e pesquise por "métodos http" (sem as aspas). Ao clicar em pesquisar, o resultado será mostrado no navegador. Note que no meu caso eu fiz a busca digitando diretamente na barra do navegador Chrome. Veja a barra de endereços. Haverá algo assim:

https://www.google.com/search?q=métodos+http&oq=métodos+http&aqs=chrome.....

Parece grego, mas não é! Vamos entender a URL. Estamos usando o protocolo HTTP, para acessar a máquina www.google.com e o recurso inicia em search e, a partir do ponto de interrogação, são codificados os parâmetros enviados na requisição do recurso:

```
\left( q = m etodos + http&oq = m etodos + http&aqs = chrome..... \right)
```

Dividindo esse resto da URL nos símbolos "&". Vamos obter isso aqui:

```
q=métodos+http
oq=métodos+http
aqs=chrome....
```

Note que a forma de cada pedaço da parte correspondente aos parâmetros enviados corresponde a é x=y, onde "x" é o nome de um parâmetro e "y" é o valor associado a ele. No nosso exemplo, o parâmetro "q" tem o valor "métodos+http" que no caso é a nossa consulta! Ou seja, o componente que trata as pesquisas do Google (search), entenderá que o parâmetro "q" vai conter o valor da pesquisa que estamos fazendo!

Execute novamente o nosso projeto, limpe todos os campos e preencha o campo "Nome" com "Juca" (sem as aspas) e o campo "Sexo" com Masculino e clique em "Enviar Dados". Veja a URL que foi obtida na barra de endereços:

```
http://localhost:8080/PrimeiroFormulario/processaDadosCliente?
```

Veja o caminho do recurso!

```
processaDadosCliente?nome=Juca&sobrenome=&cpf=&dataNasc=&)
sexo=M&observacoes=
```

O que isso quer dizer que, no nosso caso, o Servlet mapeado no endereço "proces-saDadosCliente" receberá os parâmetros nome, sobrenome, cpf, dataNasc, sexo e observacoes, receberá os valores associados a ele para serem usados e deverá, de alguma forma, retornar o recurso associado a eles. O ponto de interrogação após o mapeamento do Servlet (/processaDadosCliente) indica que o que vem depois dele (do ponto de interrogação) são parâmetros HTTP. Cada parâmetro, como já vimos, está na forma x=y, onde "x" é o parâmetro e "y" é o valor, sendo que eles são separadas por "&". Então temos: nome igual a "Juca", sobrenome igual a vazio, cpf igual a vazio, dataNasc igual a vazio, sexo igual a "M" e observacoes igual a vazio. A saída no NetBeans deve ter ficado assim:

Dados do Cliente: |#]

Nome: Juca|#]

Sobrenome: |#]

CPF: |#]

Data de Nascimento: |#]

Sexo: Masculino|#]
Observações: |#]

Vamos mandar a requisição novamente para o NetBeans, só que agora modificando a URL ao invés de usar o formulário. Dê o sobrenome de "Santos" ao Juca e defina o CPF como 123456789. Preencheu a URL na barra de endereços? Tecle <ENTER> e veja o que aconteceu no NetBeans. A saída deve ter sido essa aqui:

Dados do Cliente: |#]

Nome: Juca | #]

Sobrenome: Santos | #] CPF: 123456789 | #]

Data de Nascimento: |#]

Sexo: Masculino|#]
Observações: |#]

Então, basicamente, ao usarmos o método GET, indicamos que queremos algum recurso do servidor. Quando enviamos dados através do método GET, esses dados, na forma de parâmetros, são codificados na própria URL. O nosso formulário do index.html utiliza por padrão o método GET. Qualquer formulário usa por padrão o método GET, mas se quisermos mudar o método de envio do formulário, precisamos usar a propriedade method da tag <form>. Ai você me pergunta: Porque usaríamos outro método? O GET já não funciona? E eu respondo: Sim, o GET funciona, mas imagine a seguinte situação: você vai armazenar os dados de um usuário de um sistema. Você vai mandar vários dados para o servidor, inclusive uma senha. O que acontece? A senha enviada vai aparecer na URL! Afinal, a senha é um campo do formulário! O ideal seria ninguém a ver correto? Outro problema. O tamanho de uma URL é fixo! Então não podemos mandar conteúdos de tamanho arbitrário, visto que iremos perder dados caso usemos o método GET! Imagine mandar um vídeo para publicação no YouTube! Então como fazemos? Método POST, ao resgate!

2.3.2 Método POST

O método POST (*to post* = postar) é usado para enviar qualquer tipo de dados ao servidor o que, normalmente, acarretará em algum tipo de mudança no recurso requisitado ou mesmo no servidor. Ao contrário do método GET, ao usar o método POST, os parâmetros de um formulário não são inseridos na URL, mas sim no corpo da requisição. Sendo assim, a quantidade dos dados enviados usando o método POST pode ter qualquer tamanho, desde apenas um parâmetro, até arquivos de tamanhos

variados. Você já enviou uma foto para o Facebook não enviou? Saiba que ela foi enviada usando o método POST.

Como eu já disse na seção anterior, por padrão, o navegador envia os dados de um formulário usando o método GET. Caso queiramos mudar esse comportamento, basta usar a propriedade method da tag (form). Vamos fazer isso? Vá ao NetBeans, abra o arquivo index.html caso não esteja aberto, procure pela tag (form) e insira a propriedade method. Veja na Listagem 2.4 como deve ficar.

Salve o arquivo e execute o projeto novamente. Preencha o formulário e clique em "Enviar Dados". Verifique a URL, pois agora os parâmetros não serão mais codificados nela. Verifique a saída no NetBeans para constatar que os dados continuam a ser enviados. Edite novamente o index.html e mude o método para GET. Teste novamente. Os parâmetros devem estar aparecendo novamente na URL não é? De novo, edite o index.html, volte para o método POST e teste de novo.

Muito bem! Estamos quase acabando. Tenho certeza que você deve estar entendendo tudo. Se não estiver, releia o que está com dúvida tudo bem? Vamos à nossa última Seção, onde trataremos de mais um pouquinho de teoria.

2.3.3 Tratando Métodos HTTP

Você deve lembrar que quando criamos um Servlet manualmente, nós criávamos três métodos, o processRequest(...), o doGet(...) e o doPost(...). Quando criamos um Servlet usando o assistente do NetBeans, ele também cria uma estrutura parecida com a que a gente criava manualmente, além de já realizar o mapeamento do nosso Servlet usando a anotação @WebServlet. Na seção anterior falamos dos métodos GET e POST do protocolo HTTP, que são os que nós usamos como desenvolvedores Web. Você já deve ter notado, e eu também já falei, que um Servlet deve implementar o método HTTP que ele deve tratar. A implementação de um método HTTP em um Servlet deve ser feita dentro de métodos que por padrão são nomeados doXXX(...), onde XXX deve ser trocado pelo nome do método HTTP em questão. Sendo assim, requisições usando o método GET são tratadas dentro do método

2.4. RESUMO 39

doGet(...) do Servlet. Requisições usando o método POST são tratadas dentro do método doPost(...) do Servlet e assim por diante.

Note então que todos os Servlets que criamos até agora se comportam da mesma forma tanto para o método GET quanto para o método POST, pois sempre que a requisição chega no Servlet, o método apropriado é escolhido, entretanto, tanto o método doGet(...), quanto o método doPost(...), direcionam o fluxo de execução para o método processRequest(...)!

Muito legal não é mesmo? Com isso fechamos este Capítulo. No próximo iremos aprender a trabalhar com dois recursos importantes da especificação das JSPs: *Expression Language* (EL) e TagLibs. Após aprender essas duas funcionalidades, estaremos prontos para começar a criar nosso primeiro projeto que trabalha com banco de dados, mas antes disso ainda iremos formalizar e aprender algumas coisinhas. Ah, não se esqueça de praticar o que aprendemos até agora! Vamos ao resumo do Capítulo.

2.4 Resumo

Neste Capítulo demos um passo muito importante para a nossa vida como desenvolvedores Web, pois aprendemos a trabalhar com formulários e entendemos o funcionamento dos métodos GET e POST que fazem parte do protocolo HTTP. Como você já deve ter percebido, os formulários desempenham um papel importantíssimo nas aplicações Web. Tenho certeza que de agora em diante, sempre que você usar uma aplicação Web, você saberá como aquele formulário funciona. Para colocar em prática o que aprendemos, criamos um projeto Java Web no NetBeans e fizemos diversos testes.

2.5 Exercícios

Exercício 2.1: Qual a diferença entre os métodos GET e POST? Quando devemos utilizar um ou o outro?

2.6 Projetos

Projeto 2.1: Incremente o projeto que criamos durante o Capítulo inserindo mais alguns campos no formulário: email, logradouro, número, complemento, cidade, estado, CEP, se o cliente tem ou não filhos. Utilize apropriadamente os tipos de *input* que aprendemos até agora.

Projeto 2.2: Crie um novo projeto Java Web, com o nome de "FormularioDVD" (sem as aspas), que deve ter um formulário usado para enviar dados de um DVD. Um DVD, no nosso caso, deve ter: número, título, ator/atriz principal, ator/atriz coadjuvante, diretor/diretora e ano de lançamento. Para tratar o formulário, crie um Servlet usando o assistente do NetBeans. Esse Servlet deve obter os dados enviados através do formulário e imprimi-los na saída padrão usando System.out.println(...), como foi feito no exemplo construído durante este Capítulo. Esse formulário deve usar o método POST.

Projeto 2.3: Crie um novo projeto Java Web, com o nome de "Formulario Produto" (sem as aspas), que deve ter um formulário usado para enviar dados de um Produto. Um Produto, no nosso caso, deve ter: código de barras, descrição, unidade de medida (unidade ou kg), quantidade por embalagem, fabricante (nome). Para tratar o formulário, crie um Servlet usando o assistente do NetBeans. Esse Servlet deve obter os dados enviados através do formulário e imprimi-los na saída padrão usando System.out.println(...), como foi feito no exemplo construído durante este Capítulo. Esse formulário deve usar o método POST.

Projeto 2.4: Crie um novo projeto Java Web, com o nome de "CalculadoraWeb" (sem as aspas), que deve ter um formulário usado para atuar como uma calculadora. Nesse formulário, deve haver dois campos ("número 1" e "número 2") e um conjunto de radios para representar a operação a ser realizada (adição, subtração, multiplicação e divisão). Para tratar o formulário, crie um Servlet usando o assistente do NetBeans. Esse Servlet deve obter os dados enviados através do formulário, executar a operação escolhida pelo usuário e imprimir o resultado na saída padrão usando System.out.println(...), como foi feito no exemplo construído durante este Capítulo. Esse formulário deve usar o método GET. Faça testes de envio dos dados usando apenas a URL gerada depois da primeira submissão.

Projeto 2.5: Crie um novo projeto Java Web, com o nome de "TamanhoString" (sem as aspas), que deve ter um formulário com apenas um campo usado para enviar uma String de qualquer tamanho para um Servlet. Utilize uma <textarea> para o usuário poder inserir essa String no formulário. O Servlet deve obter a String enviada e imprimir a quantidade de caracteres da String na saída padrão usando System.out.println(...), como foi feito no exemplo construído durante este Capítulo. Qual método HTTP deve ser utilizado nessa situação? Justifique sua resposta.

Projeto 2.6: Crie um novo projeto Java Web, com o nome de "EhPrimo" (sem as aspas), que deve ter um formulário usado para enviar um número inteiro para um Servlet, que por sua vez deve verificar se este número é primo. O resultado do teste deve ser impresso na saída padrão. Esse formulário deve usar o método GET.

2.6. PROJETOS 41

Projeto 2.7: Crie um novo projeto Java Web, com o nome de "EquacaoSegundoGrau" (sem as aspas), que deve ter um formulário usado para enviar os coeficientes de uma equação de segundo grau para um Servlet, que por sua vez deve calcular as raízes da equação em questão e imprimir essas raízes na saída padrão. As raízes de uma equação do segundo grau podem ser determinadas usando a fórmula de Bhaskara http://pt.wikipedia.org/wiki/Bhaskara_II. O Servlet deve verificar também se os coeficientes passados representam uma equação do segundo grau válida. Esse formulário deve usar o método GET.

Expression Language E TagLibs

"A magia da linguagem é o mais perigoso dos encantos".

Edward Bulwer-Lytton



ESTE Capítulo teremos como objetivo entender a sintaxe, o propósito e como utilizar tanto a *Expression Language* quanto as *tags* JSP, além de aprendermos a lidar com as *tags* disponibilizadas na *JavaServer Pages Standard Tag Library* (JSTL).

3.1 Introdução

Neste Capítulo iremos aprender duas funcionalidades muito importantes e úteis do mundo Java Web: a *Expression Language* (EL) e as TagLibs. Essas duas funcionalidades nos ajudarão na tarefa de não misturar código Java nas nossas JSPs. Você se lembra quando falei que era possível inserir código Java dentro das JSPs, não lembra? Falei também que isso não deveria ser feito, pois deixa o código difícil de ler e de manter, além de fazer com que o trabalho do Web designer (que normalmente conhece mais HTML e CSS) se torne difícil, visto que ele teria que ter um bom conhecimento em Java e em como as JSPs funcionam. Usando a EL e as TagLibs, a manipulação de dados, provenientes dos Servlets, se torna muito mais fácil, pois utiliza uma sintaxe

simples e fácil de entender, ajudando no trabalho de quem não conhece muito bem o funcionamento de aplicações Web em Java. Vamos começar?

3.2 Expression Language (EL)

A EL é um recurso da especificação das JSPs que permite utilizarmos uma sintaxe especial para obter dados que gostaríamos de mostrar nas nossas páginas, além de permitir que façamos algumas outras coisas, como por exemplo, avaliar uma expressão lógica. Como de costume, iremos utilizar um projeto para aprendermos o recurso que estamos estudando. Crie um projeto Java Web com o nome "UsandoELeTagLibs" (sem as aspas). Nesse projeto teremos um formulário no index.html que terá seus dados tratados por um Servlet, que por sua vez irá fazer algum processamento e direcionar o resultado gerado para uma página JSP, chamada exibeDados.jsp.

Edite o index.html e insira um formulário. O meu ficou como mostrado na Listagem 3.1. Copie o código para o seu index.html e teste.

```
Listagem 3.1: Formulário para envio de dados de um produto (index.html)
   <!DOCTYPE html>
2
  <html>
3
       <head>
4
           <title>Usando EL e TagLibs</title>
5
           <meta charset="UTF-8">
6
           <meta name="viewport"</pre>
7
                  content="width=device-width, initial-scale=1.0">
8
9
           <style>
10
                .alinharDireita {
11
                    text-align: right;
12
                }
13
           </style>
14
15
       </head>
16
       <body>
17
           <div>
18
19
                <h1>Dados do Produto</h1>
20
21
                <form method="post" action="processaDadosProduto">
22
```

```
23
             24
               25
                  Código:
26
                  <input type="text" name="codigo"/>
27
               28
               29
                  Descrição:
30
                  <input type="text" name="descricao"/>
31
               32
                33
                  Unidade de Medida:
34
                  >
                     <select name="unidade">
36
                        <option value="kg">Quilograma</option>
37
                        <option value="l">Litro</option>
38
                        <option value="un">Unidade</option>
39
                     </select>
40
                  42
                43
                  Quant. em Estoque:
44
                  <input type="text" name="quantidade"/>
45
               46
                47
                  48
                     <input type="submit" value="Enviar Dados"/>
49
50
               51
             52
          </form>
54
55
       </div>
56
    </body>
57
  </html>
```

Note que nesse arquivo temos algumas coisas novas. A primeira novidade é a *tag* <style> na linha 10. Usamos essa *tag* para criar regras de estilo para formatar/estilizar a visualização do nosso nosso documento. Tudo que usarmos de formatação como alinhamento, cor de texto etc., será definido usando estilos. Esses estilos são codificados usando as folhas de estilo *Cascading Style Sheets* (CSS). A sintaxe é muito

simples. As definições em CSS são chamadas de seletores. Sendo assim, a definição alinharDireita (linha 11) indica que estamos definindo um seletor que é uma classe de formatação (denotada pelo ponto (.)) que tem como nome alinharDireita. Todas as propriedades de formatação de um seletor são inseridas entre chaves. Note que usei a propriedade text-align com o valor de "right". Ou seja, todas as tags que usarem a classe (atributo class) alinharDireita vão ser formatadas de forma a alinhar seu texto à direita.

Você já deve conhecer as tabelas do HTML não é mesmo? Note que organizei todo o formulário dentro de uma tabela e que a primeira coluna da tabela usa a classe <code>.alinharDireita</code>, que definimos no começo do arquivo. Verifique a linha 26 para ver um exemplo. Outra modificação que fiz foi em relação à *action* da *tag* <code><form></code>. Note que o caminho expresso na *action* é o mapeamento do Servlet que tratará a requisição, assim como estamos fazendo nos Capítulos anteriores. Esse tipo de caminho se chama "caminho relativo", pois o mapeamento do Servlet¹ está no mesmo diretório em relação ao index . html², sendo assim, não precisamos colocar o caminho completo, pois os dois recursos estão no mesmo diretório. Ao prosseguirmos com o conteúdo, irei ensinar uma técnica muito útil para não termos problema com os caminhos dos recursos.

A última novidade no nosso formulário é o uso da tag (select) (linha 36) que é usada para criar uma caixa de seleção ($combo\ box$). Veja que a propriedade name é definida na tag (select) e que dentro dessa tag existem três tags do tipo (opção). A tag (option) é usada para criar um item da caixa de seleção. Cada option tem um valor associado que será enviado para o servidor com base na seleção feita. Por exemplo, se a opção "Unidade" for selecionada, será enviado o valor "un" no parâmetro "unidade", definido na propriedade name da tag (select).

Com o formulário pronto, vamos criar uma classe que vai representar o nosso produto. Na pasta *Source Packages*, crie um novo pacote chamado "entidades" (sem as aspas). Nesse pacote, crie uma classe com o nome de "Produto" (sem as aspas). Nosso produto contém um código, uma descrição, uma unidade de medida e uma quantidade em estoque. Sendo assim, nossa classe também terá esses quatro campos, que devem ser implementados como membros privados. Você deve ter aprendido que quando criamos uma classe, devemos tornar seus campos privados e então criar métodos públicos para configurar e obter esses dados. Em Java, nós usamos um padrão chamado JavaBeans, que define algumas regras para se nomear os métodos que serão usados. Por exemplo, nosso produto terá um membro privado chamado quantidade, então teremos dois métodos públicos para acessar esse campo. O mé-

¹http://localhost:8088/UsandoELeTagLibs/processaDadosProduto

²http://localhost:8088/UsandoELeTagLibs/index.html

todo setQuantidade(...) (configure a quantidade) será usado para configurar a quantidade, enquanto o método getQuantidade() (obtenha a quantidade) será usado para obter o valor da quantidade. Utilizando essa abordagem de usar métodos para obter e configurar certos campos, nós obtemos o que chamamos de propriedades de uma determinada classe, pois independente de como os métodos são implementados, os usuários da classe só enxergam os métodos públicos. Lembra-se da propriedade do encapsulamento da programação orientada a objetos? Olha ela ai! Note que usamos os prefixos set e get para nomear métodos que respectivamente alteram e obtenham uma determinada propriedade do objeto. O uso desses prefixos, entre outros detalhes, é descrito no padrão JavaBeans.

Quantos detalhes hein? Veja na Listagem 3.2 como ficou a implementação da classe Produto.

Listagem 3.2: Implementação da classe Produto (entidades/Produto.java) package entidades; 2 3 * Entidade Produto. 4 5 * @author Prof. Dr. David Buzatto 6 7 public class Produto { 10 private int codigo; private String descricao; 11 private String unidadeMedida; 12 private int quantidade; 13 14 public int getCodigo() { 15 return codigo; 16 } 17 18 19 public void setCodigo(int codigo) { this.codigo = codigo; 20 } 21 22 public String getDescricao() { 23 return descricao; 24 } 25 26

```
public void setDescricao( String descricao ) {
27
           this.descricao = descricao;
28
       }
29
30
       public String getUnidadeMedida() {
31
           return unidadeMedida;
32
33
34
       public void setUnidadeMedida( String unidadeMedida ) {
35
           this.unidadeMedida = unidadeMedida;
36
       }
37
38
       public int getQuantidade() {
39
           return quantidade;
40
       }
41
42
       public void setQuantidade( int quantidade ) {
43
44
           this.quantidade = quantidade;
       }
45
46
  }
47
```

Com a classe Produto implementada, agora podemos criar objetos do tipo Produto que vão conter os dados obtidos no formulário. Quem obtém e processa os dados enviados pelo formulário são os Servlets, então vamos criar um. Crie o pacote "servlets" -no mesmo nível do pacote "entidades" que já foi criado- para conter o Servlet que será criado. A classe do nosso Servlet, que deverá estar dentro do pacote recém-criado, terá o nome de "ProcessaDadosProdutoServlet" e deverá ser mapeada para "/processaDadosProduto" em *URL Pattern(s):*. A implementação do método processRequest do Servlet criado pode ser vista na Listagem 3.3.

```
Listagem 3.3: Implementação do Servlet que cria um novo Produto (servlets/ProcessaDadosProdutoServlet.java)

1 package servlets;
2 import entidades.Produto;
4 import java.io.IOException;
5 import jakarta.servlet.RequestDispatcher;
6 import jakarta.servlet.ServletException;
7 import jakarta.servlet.annotation.WebServlet;
```

```
import jakarta.servlet.http.HttpServlet;
  import jakarta.servlet.http.HttpServletRequest;
  import jakarta.servlet.http.HttpServletResponse;
11
   /**
12
    * Servlet para processamento de dados de produtos.
13
    * @author Prof. Dr. David Buzatto
15
16
   @WebServlet( name = "ProcessaDadosProdutoServlet",
17
                urlPatterns = { "/processaDadosProduto" } )
18
   public class ProcessaDadosProdutoServlet extends HttpServlet {
19
20
       protected void processRequest(
21
               HttpServletRequest request,
22
               HttpServletResponse response )
23
               throws ServletException, IOException {
24
25
           request.setCharacterEncoding( "UTF-8" );
27
           // obtém os dados do formulário
           int codigo = 0;
29
           int quantidade = 0;
30
           String descricao = request.getParameter( "descricao" );
31
           String unidadeMedida = request.getParameter( "unidade" );
           try {
34
               codigo = Integer.parseInt( request.getParameter( "codigo" ));
35
           } catch ( NumberFormatException exc ) {
36
               System.out.println( "Erro ao converter o codigo." );
           }
39
           try {
40
               quantidade = Integer.parseInt(request.getParameter("quantidade"));
41
           } catch ( NumberFormatException exc ) {
42
               System.out.println( "Erro ao converter a quantidade." );
           }
45
           // cria um novo produto e configura suas propriedades
46
           // usando os dados obtidos do formulário
47
           Produto prod = new Produto();
48
           prod.setCodigo( codigo );
49
```

```
prod.setDescricao( descricao );
50
           prod.setUnidadeMedida( unidadeMedida );
51
           prod.setQuantidade( quantidade );
52
53
           // configura um atributo no request chamado "produtoObtido"
54
           // sendo que o valor do atributo é o objeto "prod"
55
           request.setAttribute( "produtoObtido", prod );
56
57
           // prepara um RequestDispatcher para direcionar para a página
58
           // "exibeDados.jsp" que está no mesmo diretório em relação
59
           // ao mapeamento deste Servlet
60
           RequestDispatcher disp
61
            → =request.getRequestDispatcher("exibeDados.jsp");
62
           // faz o direcionamento, chamando o método forward.
63
           disp.forward( request, response );
64
65
       }
66
67
       @Override
68
       protected void doGet(
69
               HttpServletRequest request,
70
               HttpServletResponse response )
71
               throws ServletException, IOException {
72
           processRequest( request, response );
73
       }
74
75
       @Override
76
       protected void doPost(
77
               HttpServletRequest request,
78
               HttpServletResponse response )
79
               throws ServletException, IOException {
80
           processRequest( request, response );
81
       }
82
83
       @Override
84
       public String getServletInfo() {
85
           return "ProcessaDadosProdutoServlet";
86
       }
87
88
89 }
```

Vamos às novidades apresentadas nesse Servlet. Entre as linhas 29 e 32 declaramos as variáveis que vão conter o valor dos campos do formulário, sendo que só obtemos os valores da descrição e da unidade de medida, pois o método getParameter retorna Strings.

Para as variáveis inteiras, nós precisamos converter o valor retornado de "codigo" e "quantidade" para inteiro. Você já deve conhecer esse tipo de conversão não é mesmo? Entre as linhas 34 e 44 usamos dois blocos try para verificar se a conversão de cada valor que deve ser inteiro foi bem sucedida. Caso você não conheça essa construção da linguagem, segue então uma explicação bem rápida.

O bloco try (tentar) é usado na linguagem Java para englobar trechos de código que, ao serem executados, podem emitir certos tipos de "erros". Esses "erros" são chamados de exceções. O método parseInt(...) da classe Integer pode gerar um tipo desses erros quando é passado para ele uma String que não representa um número. Exemplo: imagine que no formulário dos dados do produto, você preencheu "um" ao invés de "1" no campo código. Esse valor ("um") vai para o Servlet e quando o método parseInt tenta convertê-lo para um inteiro, ele verifica que "um" não representa um número, então ele dá um tipo de "grito", que avisa quem está usando o método que alguma coisa errada aconteceu. Para ouvir esse "grito", precisamos usar o bloco [try] e, logo em seguida, usar um catch, que é como se fosse um tipo de "ouvido" que só ouve um tipo de "grito". O [parseInt(...)] tentou converter "um", não conseguiu e então gritou: "NumberFormatException!!!" Como temos um catch ("ouvido") configurado para ouvir esse tipo de "grito", quando o [parseInt(...)] "gritar", o catch vai entender o "grito" e vai fazer alguma coisa, que no caso é mostrar na saída "Erro ao converter o código.". A mesma coisa é feita para o valor da quantidade. Resumindo –o try é usado para englobar uma ou mais linhas de código que potencialmente podem lançar algum tipo de exceção, sendo que a exceção que é lançada dentro do try deve ser capturada em um catch correspondente. Essa explicação é uma forma bem simples de entender o mecanismo de tratamento de exceções do Java, visto que existem muitos outros detalhes, como exceções que obrigatoriamente precisam ser tratadas ou lançadas ou não precisam ser verificadas, assim como a NumberFormatException no nosso caso.

Voltando ao Servlet... Na linha 48 da Listagem 3.3 é instanciado um novo Produto e este é atribuído a uma referência do tipo Produto chamada prod. Entre as linhas 49 e 52 são configuradas as propriedades do produto a partir dos dados obtidos através do formulário. Na linha 56, inserimos um atributo no request. Damos o nome de "produtoObtido" a esse atributo e configuramos seu valor como sendo o produto que criamos e configuramos entre as linhas 48 a 52. Isso quer dizer que a próxima página ou Servlet que receber a requisição a partir deste Servlet, vai receber um objeto

request com esse atributo, ou seja, o objeto "prod", que é um Produto, vai ficar acessível a outro componente da nossa aplicação! Confuso? Já você vai entender, fique calmo.

Na linha 61 criamos um RequestDispatcher, que é usado para direcionar o fluxo de execução do Servlet que está sendo executado para um outro recurso. No caso, esse recurso foi definido como exibeDados. jsp, uma página JSP que ainda vamos criar e que vai usar o atributo "produtoObtido" configurado no request para exibir os dados do produto.

Por fim, na linha 64, o método forward de disp, que é o nosso RequestDispatcher para o recurso exibeDados. jsp é invocado, passando como parâmetro o request e o response do Servlet. Quando o método forward é invocado, o servidor direciona o fluxo para o recurso configurado e devolve o controle para o navegador caso o recurso deva ser exibido por ele. Uma JSP, por padrão, é usada para isso não é mesmo?

O que a página exibeDados . jsp vai fazer é pegar o atributo "produtoObtido" configurado no request e mostrar seus dados. Vamos criar então essa página. No NetBeans, procure pela pasta chamada *Web Pages*. Clique com o botão direito nela, escolha *New* e procure por *JSP...*. Se não achar essa opção, você já deve saber como proceder não é mesmo? Preencha o campo *File Name*: com "exibeDados" (sem as aspas) e clique em *Finish*. O arquivo será criado e exibido no editor. Veja na Listagem 3.4 como ficou o código depois de ser editado.

```
Listagem 3.4: Código do arquivo (exibeDados.jsp)
   <%@page contentType="text/html" pageEncoding="UTF-8"%>
  <!DOCTYPE html>
2
3
  <html>
4
       <head>
5
           <title>Produto Obtido</title>
6
           <meta charset="UTF-8">
7
           <meta name="viewport" content="width=device-width, initial-scale=1.0">
8
9
10
           <style>
               .alinharDireita {
11
                   text-align: right;
12
               }
13
           </style>
14
15
       </head>
16
```

```
<body>
17
       <div>
18
19
         <h1>Produto Obtido</h1>
20
21
         22
            23
               Código:
24
               ${requestScope.produtoObtido.codigo}
25
            26
            27
               Descrição:
28
               ${requestScope.produtoObtido.descricao}
            30
            31
               Unidade de Medida:
32
               ${requestScope.produtoObtido.unidadeMedida}
33
            34
            Quant. em Estoque:
36
               ${requestScope.produtoObtido.quantidade}
37
            38
            39
               40
                  <a href="index.html">Voltar</a>
41
               42
            43
          44
45
       </div>
46
    </body>
 </html>
48
```

Copiou? Salvou? Faça um teste então! Execute o projeto, preencha o formulário e clique em "Enviar Dados". O Servlet será invocado, processará os dados e vai redirecionar para a página exibeDados. jsp, que por sua vez vai mostrar os dados do produto. Legal não é? Mágica? Não! Vamos entender o que está acontecendo no código do arquivo exibeDados. jsp. Veja as linhas 25, 29, 33 e 37. A construção \${...} é a EL! Usando a EL, nós podemos acessar valores que estão configurados no request e em outros escopos também, que vamos aprender depois. No caso, o objeto requestScope da EL faz referência ao objeto request do Servlet que é gerado a partir da JSP. Lembrese que uma JSP é convertida em um Servlet automaticamente pelo servidor!

Usar a expressão \${requestScope.produtoObtido.codigo} em EL quer dizer: obtenha o código do objeto configurado no atributo produtoObtido do request. Lembrese, nós configuramos no atributo "produtoObtido" no request (linha 56 da Listagem 3.3) um produto, que por sua vez tem um código (acessado pelo método getCodigo).

O propósito da EL é obter objetos que estão ativos nos diversos escopos da aplicação e poder obter suas propriedades, sem precisar lidar diretamente com código Java. Sei que esse foi um exemplo bem simples, mas tenho certeza que você deve ter entendido. Veja que o codigo usado na EL é relativo ao método getCodigo(...) e não ao membro privado da classe Produto chamado codigo. Essa mágica se dá pelo uso do padrão JavaBeans! Como exercício mental, analise as linhas 29, 33 e 37 e tente imaginar o que está acontecendo. Agora que já sabemos o que é a EL e como utilizá-la, vamos às *tags* JSP.

3.3 Tags JSP

Na especificação das JSPs, existem uma série de *tags* especiais que devem ser implementadas por quem implementa a especificação. Essas *tags* são nomeadas usando o prefixo "jsp". Na verdade, nós quase não iremos utilizar essas *tags* e as que utilizarmos, explicarei no momento oportuno, mas para você ter uma ideia de como elas funcionam, crie uma página JSP chamada "testesTags" na pasta *Web Pages* do projeto que estamos trabalhando. Veja na Listagem 3.5 o código que você deve copiar para o arquivo testesTags. jsp.

```
Listagem 3.5: Exemplo das tags <jsp:useBean> <jsp:setProperty>
   (testesTags.jsp)
  <%@page contentType="text/html" pageEncoding="UTF-8"%>
   <!DOCTYPE html>
2
3
  <html>
4
5
       <head>
           <title>Testes Tags JSP</title>
6
7
           <meta charset="UTF-8">
           <meta name="viewport"</pre>
8
                 content="width=device-width, initial-scale=1.0">
9
       </head>
10
       <body>
11
12
```

3.3. TAGS JSP 55

```
<%--
13
                  Criando um objeto do tipo produto
14
                  usando a tag <jsp:useBean>
15
16
            <jsp:useBean id="meuProduto"</pre>
17
                           class="entidades.Produto"
18
                           scope="page"/>
19
            <jsp:setProperty name="meuProduto"</pre>
20
                               property="codigo"
21
                               value="4"/>
22
            <jsp:setProperty name="meuProduto"</pre>
23
24
                               property="descricao"
                               value="Arroz"/>
25
            <jsp:setProperty name="meuProduto"</pre>
26
                               property="unidadeMedida"
27
                               value="kg"/>
28
            <jsp:setProperty name="meuProduto"</pre>
29
                               property="quantidade"
30
                               value="100"/>
31
32
            <h1>Produto Criado:</h1>
33
            ${pageScope.meuProduto.codigo},
34
            ${pageScope.meuProduto.descricao},
35
            ${pageScope.meuProduto.unidadeMedida},
36
            ${pageScope.meuProduto.quantidade}
37
38
       </body>
39
   </html>
40
```

Copie código arquivo, salve endereço o no seu e acesse http://localhost:8080/UsandoELeTagLibs/testesTags.jsp no seu navegador para testar a página. O que aconteceu? O produto criado foi exibido assim "4, Arroz, kg, 100" não foi? Vamos analisar o código no arquivo testesTags. jsp. Entre as linhas 13 e 16 definimos um comentário, que nas JSPs é delimitado entre <%− e −%>. Esse comentário não pode ser visto no código-fonte da página HTML gerada! Nas linhas 17 e 19 usamos a tag (<jsp:useBean>) para criar um objeto com nome de "meuProduto", configurado pelo atributo id, do tipo entidades. Produto, configurado pelo atributo class, que vai existir no escopo da página, ou seja, esse objeto só existe nesta página. Note que precisamos colocar o caminho completo da classe no atributo class para informarmos qual o tipo de objetos que queremos instanciar. A partir da linha 20, usamos a tag < jsp: setProperty> para configurar as propriedades do

objeto chamado "meuProduto" que foi criado usando a tag (jsp:useBean). Entre as linhas 20 e 22, referenciamos o objeto "meuProduto" e configuramos a propriedade "codigo" ((property="codigo") com o valor "4". A instrução em Java equivalente a estas duas linhas é meuProduto.setCodigo(4). A partir da linha 33, mostramos então os dados do objeto "meuProduto" que foi criado usando EL. Note que desta vez, usamos pageScope ao invés de requestScope, visto que o objeto existe apenas no escopo da página (veja na linha 19).

Da mesma forma que existem as *tags* JSP padrão, você pode criar suas próprias *tags* que podem ter comportamentos dos mais variados possíveis, entretanto nós não iremos aprender a fazer isso. Como é possível criar *tags* personalizadas, nós podemos usar conjuntos de *tags* que são implementadas por terceiros em nossos projetos. Um desses conjuntos é a *JavaServer Pages Standard Tag Library* (JSTL), que vamos aprender na próxima Seção. Vamos lá então!

3.4 JavaServer Pages Standard Tag Library - JSTL

A JSTL é uma biblioteca formada por um conjunto de *tags* (TagLib = *tag Library* = Biblioteca de *tags*) que visam apoiar o desenvolvedor na tarefa de construir suas páginas JSP, permitindo que várias coisas possam ser feitas sem o uso direto de código Java, por exemplo, iterar por uma lista de objetos, executar testes lógicos, formatar dados, entre muitos outros. Quando formos implementar nosso primeiro projeto no Capítulo 6, iremos utilizar muitos recursos da JSTL, mas por agora vamos aprender apenas como inseri-la no nosso projeto e fazer um pequeno exemplo.

Vamos implementar um exemplo bem simples. Iremos criar um for usando *tags* da JSTL. Crie mais uma página JSP, com o nome de "testesJSTL". O NetBeans vai abrir o arquivo quando este for criado. Vamos editá-lo? Copie então o código da Listagem 3.6.

```
Listagem 3.6: Exemplo de uso da JSTL (testesJSTL.jsp)
  <%@page contentType="text/html" pageEncoding="UTF-8"%>
  %@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
  <!DOCTYPE html>
3
4
5
  <html>
      <head>
           <title>Testes Tags JSTL</title>
7
           <meta charset="UTF-8">
8
           <meta name="viewport"</pre>
9
                 content="width=device-width, initial-scale=1.0">
10
```

```
11
          <style>
12
             .linhaPar {
13
                 background: #00bbee;
14
15
16
             .linhaImpar {
17
                 background: #eeeeee;
18
             }
19
          </style>
20
21
      </head>
22
      <body>
23
          <div>
24
             25
                 <c:forEach begin="1" end="10" varStatus="i">
26
                     <c:choose>
27
                        <c:when test="${i.count % 2 == 0}">
28
                            29
                               Linha ${i.count} JSTL é animal!
30
                            31
                        </c:when>
32
                        <c:otherwise>
33
                            34
                                Linha ${i.count} JSTL é show!
                            36
                        </c:otherwise>
37
                     </c:choose>
38
                 </c:forEach>
39
             40
         </div>
      </body>
42
43 </html>
```

Copiou? Testou? Se tudo deu certo, você deve ter visto uma tabela zebrada (cor sim/cor não). Veja a Figura 3.1.



Figura 3.1: Visualização da página testes JSTL. jsp

Fonte: Elaborada pelo autor

Vamos analisar o código da Listagem 3.6. Talvez você tenha se assustado, mas não se preocupe, estou aqui para te explicar. Vamos começar pela primeira linha. Nessa linha, como em todos os JSPs que criamos até agora, usamos a diretiva page. As diretivas nos JSPs são delimitadas por <%@ e %> e são usadas para realizar algumas configurações no Servlet que será gerado a partir do JSPs. A diretiva page, no nosso caso, é usada para configurar o response do Servlet, informando que ele vai conter um documento do tipo "text/html" (atributo contentType) e que o encoding utilizado (como os caracteres são codificados) é o UTF-8 (atributo pageEncoding). Veja que isso é análogo ao que estamos fazendo manualmente na primeira linha do método processRequest(...) dos nossos Servlets.

Na linha 2 utilizamos a diretiva taglib. Essa diretiva vai permitir que nós digamos qual TagLib queremos utilizar. O atributo uri é usado para informarmos qual biblioteca de *tags* queremos utilizar. No nosso caso, queremos utilizar as funcionalidades principais da JSTL, que são chamadas de "core". A *Uniform Resource Identifier* (URI) para definir isso é a http://java.sun.com/jsp/jstl/core. O outro atributo, prefix (prefixo), nos permite definir um prefixo para usar as *tags*. O prefixo padrão para a parte "core" da JSTL é "c", mas podemos usar o prefixo que quisermos. Para

manter o padrão, iremos usar o "c" mesmo. Assim, quando qualquer desenvolvedor Web que conheça a JSTL bater o olho no código e ver alguma *tag* que inicie com "c:" vai saber que a *tag* que está sendo utilizada faz parte do core da JSTL.

Entre as linhas 12 e 20 definimos duas classes CSS. Sendo que uma usaremos para colorir o fundo das linhas pares de uma tabela, enquanto a outra será usada para colorir o fundo das linhas ímpares. A propriedade usada em ambas as classes é a background (fundo), sendo que em cada uma usamos uma cor diferente usando a notação *Red Green Blue* (RGB) em hexadecimal. Nas linhas 25 e 40 delimitamos uma tabela.

(i) | Saiba Mais

Nunca ouviu falar de cores na notação em hexadecimal? De uma olhada nesses links: http://paletton.com/, https://pt.wikipedia.org/wiki/Tripleto_hexadecimal e http://en.wikipedia.org/wiki/Web colors>

Na linha 26, usamos a tag (c:forEach) (olhe o prefixo!), usada para iterar um determinado número de vezes ou sobre alguma lista de objetos. No nosso caso, queremos que o que está entre (c:forEach) e (/c:forEach) seja executado dez vezes, pois definimos que a iteração deve iniciar em 1, usando o atributo begin, e ir até 10, usando o atributo end. Queremos também que o status da iteração seja armazenado na variável i, definida no atributo varStatus. Então temos um for que vai executar dez vezes. Durante estas dez iterações, vamos construir nossa tabela, inserindo linhas nela com apenas uma coluna, mas queremos que as linhas pares sejam coloridas usando a classe linhaPar, enquanto que as linhas ímpares sejam coloridas usando a classe linhaImpar. Sabemos que todo número par tem resto igual à zero numa divisão por dois, correto? Então precisamos saber em qual iteração estamos, calcular o resto e verificar se é zero. Se for, cria uma linha da tabela usando a classe linhaPar, caso contrário, usa linhaImpar.

Para criar séries de testes lógicos como numa estrutura if/else, nós usamos a *tag* <c:choose> (*to choose* = escolher) e dentro dela colocamos as condições que queremos testar usando a *tag* <c:when> que é equivalente aos if's e else if's e, por fim, se necessário, usamos a *tag* <c:otherwise> que é equivalente ao else. Vamos analisar o código então: entre as linhas 27 e 38 nós definimos nossa estrutura condicional usando a *tag* <c:choose>. Dentro dela, definimos na linha 28 uma *tag* <c:when>, que testa (atributo test) se a divisão da propriedade count da variável i por dois é igual a zero (par). Note o uso da EL e que podemos executar operações

aritméticas dentro dela! Se o resultado for true, o número é par e o conteúdo deste c:when é gerado, ou seja, uma linha da tabela usando a classe linhaPar. Caso contrário, como não temos mais nenhum c:when, é gerado o código dentro do c:otherwise, que por sua vez também gera uma linha da tabela, só que usando a classe linhaImpar. Fique à vontade para mudar o conteúdo gerado dentro de cada linha, bem como as classes.

Viu como não é tão complicado? Na verdade é bem simples e fácil de usar, mas precisamos praticar para ficarmos craques. Depois dessa introdução à EL e a JSTL, nós já estamos quase prontos para começarmos o nosso primeiro projeto Web de verdade, mas ainda temos que formalizar algumas coisas que serão vistas no próximo Capítulo 4. Novamente, não se esqueça de praticar o que fizemos até agora.

3.5 Resumo

Neste Capítulo nós aprendemos a criar um formulário que teve seus dados tratados por um Servlet, que por sua vez redirecionou o fluxo da aplicação para outra página JSP, utilizada para mostrar os dados informados no formulário. Com isso, tivemos uma noção do que é a EL e como ela funciona. Depois aprendemos um pouco sobre as *tags* padrão da especificação dos JSPs e, por fim, aprendemos utilizar a JSTL em nosso projeto, além de fazermos alguns testes. No Capítulo 4 vamos dar uma paradinha com os JSPs e Servlets para podermos aprender como estruturar um projeto Web que trabalha com banco de dados. Tenho certeza que você vai gostar bastante.

3.6 Exercícios

Exercício 3.1: Explique, com suas palavras, qual a importância da EL.

Exercício 3.2: Justifique porque é melhor usar a JSTL ou qualquer outra TagLib ao invés de usar código Java diretamente nas JSPs.

3.7 Projetos

Projeto 3.1: Modifique o Projeto 2.1 do Capítulo 2 para executar da mesma forma que o projeto criado neste Capítulo, ou seja, o formulário deve submeter os dados para um Servlet, que por sua vez deve criar e enviar um objeto através do request para um JSP, que deve exibir ao usuário usando EL.

3.7. PROJETOS 61

Projeto 3.2: Modifique o Projeto 2.2 do Capítulo 2 para executar da mesma forma que o projeto criado neste Capítulo, ou seja, o formulário deve submeter os dados para um Servlet, que por sua vez deve criar e enviar um objeto através do request para um JSP, que deve exibir ao usuário usando EL.

Projeto 3.3: Você já sabe que uma JSP na verdade é um Servlet não é mesmo? Será então que podemos definir na action de um formulário o endereço de um arquivo JSP ao invés de um Servlet? Crie um novo projeto Java Web, chamado "JSPTrataFormulario", onde você deve ter um formulário no index.html que contenha dois campos: nome e idade. A action deste formulário deve apontar para um arquivo JSP chamado "exibeDadosForm.jsp" (sem as aspas). Nesse arquivo, você deve mostrar os dados recebidos do formulário do index.html usando EL. Dica: para acessar os parâmetros do request usando EL, usa-se \${param.nomeDoParametro}. Por exemplo, o parâmetro idade é acessado usando \${param.idade}.

Projeto 3.4: Crie um novo projeto Java Web, com o nome de "TabelaArbitraria", onde no index.html você deve ter um formulário que pede ao usuário que seja digita a quantidade de linhas e de colunas que ele deseja que uma tabela seja gerada. Aponte a action deste formulário para outro arquivo JSP, chamado "montadorTabela.jsp", que obtém os dados enviados pelo formulário do index.html usando EL e usa dois <c:forEach> aninhados para construir a tabela de dimensões arbitrárias. Monte a tabela somente se o tamanho de linhas e de colunas for maior que zero. Se não for, exiba uma mensagem ao usuário. Dica: para testar se duas condições são verdadeiras, ou seja, se param.colunas > 0 e param.linhas > 0, use o operador and (e) da EL, enquanto o operador "maior que" é o gt (greater than). Ou seja, test="\${(param.linhas gt 0) and (param.colunas gt 0)}".

PADRÕES DE PROJETO: Factory, DAO E MVC

"É longo o caminho que vai do projeto à coisa".

Molière



ESTE Capítulo teremos como objetivo entender e aplicar os Padrões de Projeto *Factory*, DAO e MVC. Além disso iremos aprender a integrar o acesso à banco de dados em uma aplicação Java.

4.1 Introdução

A partir de agora iremos dar um passo importantíssimo em nossos estudos sobre desenvolvimento de software, pois iremos aprender a lidar com um banco de dados em um sistema de testes que iremos construir. Isso nos dará o embasamento necessário para que no Capítulo 5 nós consigamos fazer essa mesma integração em aplicações Web. Além de aprendermos a conectar nossa aplicação com uma base de dados, iremos aprender também alguns padrões de projeto que nos ajudarão a organizar nossa aplicação de forma a melhorar sua manutenção.

Antes de começarmos a discussão e a implementação desses padrões, nós precisamos preparar nosso ambiente de desenvolvimento. O NetBeans já temos insta-

lado. O Sistema Gerenciador de Banco de Dados (SGBD) que iremos utilizar, o MariaDB¹/MySQL, você provavelmente já deve ter instalado também. Com isso pronto, precisamos instalar uma ferramenta para nos ajudar a criar nossa base de dados. Iremos utilizar o MySQL Workbench, uma ferramenta gratuita para gerenciamento do MariaDB/MySQL.

4.2 Preparando o Ambiente

Para começar, vamos fazer o download da ferramenta. Acesse o endereço https://dev.mysql.com/downloads/workbench/, escolha Microsoft Windows como plataforma, que provavelmente é o sistema operacional que você está utilizando e clique no botão "Download". Fazendo isso, você será direcionado para uma página onde é requisitado um nome de usuário e senha. Se você não quiser se cadastrar (não precisa), clique no link embaixo do formulário de login onde está escrito "No thanks, just start my download". O download do instalador será iniciado.

Baixou? Legal! Execute o instalador. Na época da elaboração desse livro, a última versão disponível era a 8.0.40. Basta seguir os passos, fazendo a instalação completa. Ao terminar, deixe marcada a opção *Launch MySQL Workbench now* e clique em *Finish*. Aguarde o MySQL Workbench abrir. A interface principal da versão 8.0.40 do Workbench pode ser vista na Figura 4.1.

 $^{^1\}mathrm{Provavelmente}$ você deve ter instalado na sua máquina o Maria
DB que é distribuído junto ao XAMPP

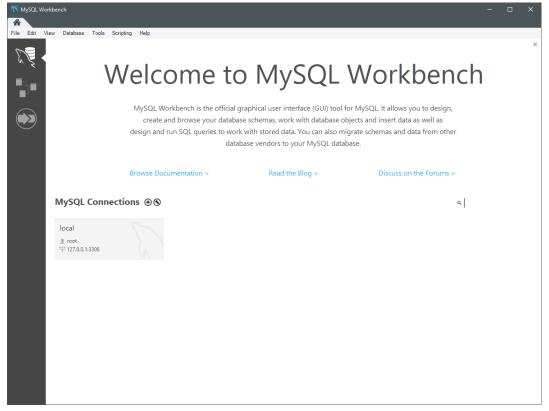


Figura 4.1: Interface principal do MySQL Workbench

Fonte: Elaborada pelo autor

Verifique se existe alguma instância configurada abaixo de [MySQL Connections], se não houver, clique em +. Um assistente aparecerá. Em [Connection Name:] dê um nome para a conexão, pode ser qual você desejar. No meu caso, é "localhost".

[Connection Method:] deve ficar com a opção "Standard (TCP/IP)" selecionada. [Hostname:] deve estar preenchido com o IP 127.0.0.1 que é o endereço de loopback. Como o servidor está instalado na máquina que você está trabalhando, deixe como está. Em [Username:] mantenha "root" e não será necessário configurar uma senha. Em [Port:], mantenha 3306, que é a porta padrão que o MariaDB/MySQL ouve. Deixe [Default Schema:] vazio. Note que estou assumindo que você está usando o MariaDB distribuído no XAMPP e que não realizou nenhuma mudança na instalação e no usuário administrador. Caso tenha realizado alguma alteração, você precisará replicá-las na configuração do MySQL Workbench. Clique no botão [Test Connection]. A ferramenta vai avisar que há uma incompatibilidade de protocolos, pois ela está tentando conectar numa instância do MySQL, mas estamos rodando o MariaDB. Esse aviso pode ser ignorado, clicando em [Continue Anyway]. Se tudo estiver correto, a

conexão será bem sucedida, sendo avisada através de um diálogo com a mensagem "Successfully made the MySQL connection". Lembre-se que o MariaDB/MySQL deve estar em execução! Por fim, clique no botão OK, o diálogo será fechado e a conexão aparecerá.

Clique duas vezes na conexão criada. Novamente, será mostrado um aviso, dizendo sobre a incompatibilidade de protocolos. Não marque a opção "Don't show this message again" e clique em Continue Anyway. Pronto? Muito bem! Sempre que abrirmos o Workbench, essas configurações já estarão feitas, não se preocupe. Agora nós vamos criar uma base de dados para trabalharmos nos exemplos deste Capítulo.

Na interface que se abriu, do lado esquerdo, em *Navigator* há duas abas que ficam abaixo. Uma é chamada *Administration*, que é a que está selecionada por padrão, e outra chamada *Schemas*. Clique em *Schemas*. No aba de esquemas, na parte em branco, clique com o botão direito e escolha *Create Schema*. Preencha o campo *Name*: com "testes_padroes" (sem as aspas) e deixe o campo *Charset/Collation*: como "Default Charset" e "Default Collation". Clique em *Apply*. Um diálogo será aberto para mostrar o código SQL que será executado. Clique em *Apply* e se der tudo certo, clique em *Finish*. A aba de criação de esquemas continuará aberta, podendo ser fechada. Ao terminar esse processo, o novo esquema (vamos chamar os esquemas de base de dados a partir de agora) será criado e estará listado na lista *SCHEMAS*. Vamos configurar a base de dados que acabamos de criar como padrão, ou seja, as instruções SQL que realizarmos serão aplicados nessa base. Para isso, clique com o botão direito em testes_padroes e escolha a opção *Set as Default Schema*. Veja a Figura 4.2. Após fazer isso, o nome da base de dados ficará em negrito.

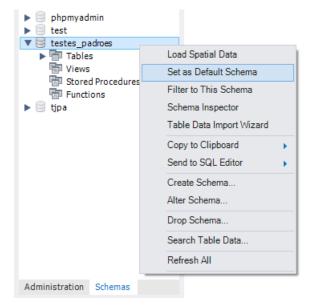


Figura 4.2: Selecionando a base de dados testes_padroes

Fonte: Elaborada pelo autor

Com a base testes_padroes configurada como padrão, expanda-a e em | *Tables* | clique com o botão direito e escolha Create Table... Vamos criar agora uma tabela que vai armazenar os dados de teste que iremos mexer durante este Capítulo. Preencha [Table Name:] com "pais" (sem as aspas). Pais é país, mas vamos omitir o acento tudo bem? Deixe os valores padrão nos campos [Charset/Collation:], Engine: e Comments: Logo abaixo, clique duas vezes na primeira linha da coluna Column Name . A ferramenta vai preencher o nome da primeira coluna automaticamente com o valor "idpais". Edite esse nome e deixe apenas "id". O tipo (Datatype) deve ficar como INT (inteiro) e as colunas PK (primary key/chave primária), NN (not null/não nulo) e AI (auto-increment/auto-incremento) devem ficar marcadas. Essa coluna da tabela será nossa chave primária. Agora vamos às outras colunas. Nossos países terão um nome e uma sigla. Então precisamos criar mais duas colunas, uma com nome de "nome" e a outra com nome de "sigla". A coluna "nome" terá o tipo (VARCHAR (100)), ou seja, um (VARCHAR) de 100 posições e a coluna "sigla" terá o tipo VARCHAR (2). Marque essas duas colunas como NN, ou seja, elas terão dados obrigatoriamente quando um país for ser inserido na tabela. Veja como ficou na Figura 4.3.

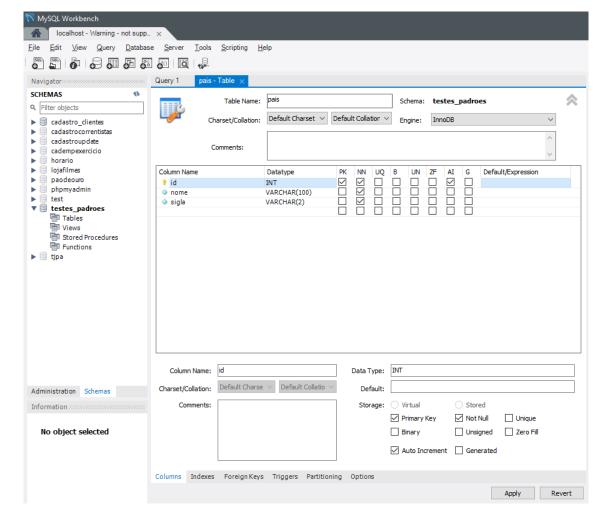


Figura 4.3: Criando a tabela "pais"

Fonte: Elaborada pelo autor

Feito isso, clique em Apply. Um diálogo será aberto para exibir o código SQL que será executado (Listagem 4.1). Clique em Apply e depois em Finish. Pronto, criamos a nossa tabela para armazenar dados dos países, ela poderá ser vista do lado esquerdo, na aba Schemas, dentro da base testes_padroes. Deixe o Workbench aberto e abra o NetBeans.

```
Listagem 4.1: Instrução CREATE TABLE para a tabela "pais"

CREATE TABLE `pais` (

id` int(11) NOT NULL AUTO_INCREMENT,
```

```
3   `nome` varchar(100) NOT NULL,
4   `sigla` varchar(2) NOT NULL,
5   PRIMARY KEY (`id`)
6 ) ENGINE=InnoDB;
```

4.3 Padrão de Projeto Factory

Primeiramente, vamos criar um novo projeto no NetBeans, só que agora ele não será um projeto Web, pois o que vamos fazer não precisa ser em um projeto desse tipo. Siga os passos que você está acostumado a fazer ao criar um projeto Java Web, só que desta vez escolha *Java with Ant* na categoria e *Java Application* nos tipos de projeto. Dê o nome de "PadroesEmPratica" para o projeto. Não se esqueça de marcar a opção *Use Dedicated Folder for Storing Libraries*).

Como iremos conectar no banco de dados MariaDB, precisamos adicionar no nosso projeto a biblioteca que implementa esse conector. Infelizmente o NetBeans não vem com essa biblioteca pronta para ser usada, então teremos que baixá-la e inserila no projeto. Para isso, primeiro baixe o arquivo . jar da bibliteca no endereço https://repol.maven.org/maven2/org/mariadb/jdbc/mariadb-java-client/ 3.5.0/mariadb-java-client-3.5.0.jar>. Com o arquivo baixado, clique com o botão direito no nó [Libraries] e escolha a opção [Add Library...]. No diálogo que foi aberto, clique em [Create...]. No diálogo que vai se abrir, preencha [Library Name:] com "MariaDB-ConnectorJ-3.5.0" (ou o nome que você quiser) e em [Library Type] deixe "Class Libraries" e clique em [OK]. Mais um diálogo será aberto. Clique no botão [Add JAR/Folder...] e procure pelo arquivo . jar da biblioteca que acabou de baixar. Antes de selecioná-lo, certifique-se que do lado direito do diálogo de abertura de arquivo a opção *Copy to Libraries Folder*: esteja marcada para que o arquivo seja copiado para dentro do nosso projeto. Após essa conferência, selecione o arquivo e clique em Add JAR/Folder. Ao clicar nesse botão, um alerta perguntará se você que criar um diretório na pasta de bibliotecas do projeto, clique em Yes. Agora clique no botão [OK] do diálogo [Customize Library] e, agora que a biblioteca foi criada, selecione-a no diálogo [Add Library] e clique no botão [Add Library]. Perceba que ao fazer isso, um aparecerá um novo nó dentro da pasta [Libraries] do projeto com o nome da bibliteca que foi criada e adicionada.

A partir de agora, podemos conectar ao banco de dados a partir do nosso código. Como você deve saber, para que possamos usar um banco de dados em Java, nós usamos uma especificação chamada *Java Database Connectivity* (JDBC). Essa especificação deve

ser implementada pelos fabricantes dos SGBDs, permitindo assim que os programas feitos em Java possam se comunicar com estes SGBDs. Normalmente, esses pacotes que implementam o JDBC são chamados de "Drivers JDBC" e são obtidos nos sites dos fabricantes dos SGBDs.

A questão agora é: "Como conectar no banco de dados?". Para que possamos conectar no MariaDB, existem uma série de "comandos" que precisamos fazer cada vez que queremos estabelecer uma conexão. Você, como desenvolvedor, sabe que uma boa prática de programação é encapsular trechos de código que fazem uma determinada tarefa em funções e que as funções em Java são chamadas de métodos.

Legal, mas e o que o tal do "padrão de projeto" tem haver com isso? Ou melhor, o que é um padrão de projeto? O termo padrão de projeto (*design pattern* em inglês) foi criado por Christopher Alexander (ALEXANDER; ISHIKAWA; SILVERSTEIN, 1977) na década de 1970 para designar soluções de sucesso para problemas recorrentes na área da arquitetura. Alexander definiu nessa época uma série de padrões que apresentavam soluções padrão para problemas que aconteciam de forma corriqueira, sendo que uma série de padrões correlatos foram o que podemos chamar de linguagem de padrões. A partir do termo cunhado por Alexander, alguns programadores começaram a criar padrões de projeto para a computação, iniciando esse trabalho do domínio da programação orientada a objetos.

O livro "Padrões de Projeto: Soluções Reutilizáveis de Software Orientado a Objetos" (GAMMA et al., 2000) é a referência básica para os padrões de projeto criados para resolver problemas relacionados ao desenvolvimento orientado a objetos. A partir de agora, quando você ler "padrão de projeto", entenda que estarei falando de padrões relacionados ao desenvolvimento de software orientado a objetos, tudo bem? Sendo assim, o primeiro padrão que iremos aprender se chama *Factory* (fábrica).

Vamos entender o contexto do padrão. Imagine que no seu programa você precisa criar e utilizar um determinado tipo de objeto muitas e muitas vezes e que este objeto precisa ser inicializado com uma série de valores, sendo que esses valores normalmente são sempre os mesmos. Assim, cada vez que você instância esse objeto, você precisa executar uma determinada quantidade de código. Como resolver isso? Criar um método que faça essa tarefa é uma boa solução não é mesmo? Mas em qual classe eu vou escrever esse método? No padrão *Factory*, nós criamos classes especializadas que serão fábricas de objetos e que terão um método que executará a tarefa da fábrica, ou seja, criar um determinado tipo de objeto.

No nosso projeto, um tipo de objeto que usaremos muito, é um objeto que representa a conexão entre nosso programa escrito em Java e o SGBD. Sendo assim, nós precisamos de uma fábrica de conexões! Vamos implementar a fábrica? No projeto que você criou agora há pouco, o NetBeans gerou por padrão um pacote chamado "padroesempratica"

33 34 dentro da pasta [Source Packages]. Crie dentro deste pacote outro com o nome de "jdbc" (sem as aspas) e dentro do pacote "padroesempratica.jdbc", crie uma classe Java com o nome de "ConnectionFactory" (sem as aspas). Essa classe conterá o método que vai fabricar a conexão. Veja o código dela na ListagemListagem 4.2.

Listagem 4.2: Uma fábrica de conexões (padroesempratica/jdbc/ ConnectionFacotory.java) package padroesempratica.jdbc; 3 import java.sql.Connection; 4 import java.sql.DriverManager; 5 import java.sql.SQLException; 7 * Uma fábrica de conexões. 8 9 * @author Prof. Dr. David Buzatto 11 public class ConnectionFactory { 13 14 * O método getConnection retorna uma conexão com a base de dados 15 * testes_padroes. 17 * @return Uma conexão com o banco de dados testes_padroes. 18 * Othrows SQLException Caso ocorra algum problema durante a conexão. 19 */ 20 public static Connection getConnection() throws SQLException { /* O método getConnection de DriverManagaer recebe como parâmetro 23 * a URL da base de dados, o usuário usado para conectar na base 24 * e a senha deste usuário. O Driver JDBC apropriado será 25 * carregado com base na biblioteca configurada. 26 return DriverManager.getConnection("jdbc:mariadb://localhost/testes_padroes", "root", 30 "root"); 31 32 }

```
35 }
```

Copiou o código? Ótimo! Esta classe tem um método estático chamado getConnection() que vai fabricar a conexão para nós e que caso ocorra algum problema, vai lançar uma exceção do tipo SQLException. Toda vez que chamarmos esse método, ele vai retornar uma nova conexão para nós e o Driver JDBC apropriado será carregado automaticamente pelo DriverManager.

Agora precisamos testar esse método para ver se não está havendo nenhum erro. Clique novamente com o botão direito no pacote "padroesempratica" e crie um novo pacote chamado "testes". Dentro do pacote "padroesempratica.testes", crie uma classe chamada "TesteConnectionFactory". Como você deve saber, para que uma classe em Java possa ser executada, nós precisamos implementar o método $\mathtt{main}(\ldots)$ com uma determinada assinatura. O que vamos fazer no método $\mathtt{main}(\ldots)$ da classe "TesteConnectionFactory" é tentar criar uma conexão e ver se nenhum erro é retornado. Na Listagem 4.3 você pode ver o código desta classe.

Listagem 4.3: Classe para teste de conexão (padroesempratica/testes/ TesteConnectionFactory.java) package padroesempratica.testes; 2 import java.sql.Connection; import java.sql.SQLException; import padroesempratica.jdbc.ConnectionFactory; 6 7 * Teste de conexão. 8 9 * @author Prof. Dr. David Buzatto 10 11 public class TesteConnectionFactory { 12 13 public static void main(String[] args) { 14 15 // tenta criar uma conexão 16 try { 17 18 Connection conexao = ConnectionFactory.getConnection(); 19 System.out.println("Conexão criada com sucesso!"); 20 21

```
} catch ( SQLException exc ) {
22
23
                System.err.println( "Erro ao tentar criar a conexão!" );
24
                exc.printStackTrace();
25
26
            }
27
28
       }
29
30
   }
31
```

Copie o código para a classe e salve o arquivo. Para executarmos apenas uma classe que tem o método [main(...)], basta clicar com o botão direito no editor e escolhe a opção Run File, ou então, com o arquivo aberto no editor, usar o atalho <Shift+F6>. Fazendo isso, a classe vai ser compilada e executada pelo NetBeans. Se tudo estiver correto, você verá na saída a mensagem "Conexão criada com sucesso!". Deu erro? Verifique se o código da classe ConnectionFactory está correto e se a senha do usuário root, definida dentro do método [getConnection()] está correta. No nosso caso, ela é root². Agora que está tudo certo, vamos simular um erro. Entre na classe ConnectionFactory e coloque uma senha inválida (terceiro parâmetro do método [getConnection(...)] de DriverManagaer) para o usuário root, por exemplo, "123". Volte na classe de testes e execute-a novamente (<Shift+F6>). Gerou um erro não foi? A mensagem "Erro ao tentar criar a conexão!" deve ter sido exibida, seguida de várias linhas que explicam o erro ocorrido. A primeira linha dos erros diz que o acesso foi negado para o usuário "root@localhost" não foi? Por que aconteceu isso? Porque a senha está errada! Volte na fábrica de conexões e coloque a senha correta (no meu caso, root). Teste novamente. Agora deve estar tudo certo.

Legal, temos uma fábrica de conexões, mas do que adianta uma fábrica de alguma coisa se a gente não usar o que é fabricado? Vamos para o próximo padrão, onde iremos organizar uma camada de persistência para nossa aplicação e usaremos a fábrica de conexões para viabilizar a comunicação entre nossa aplicação e o SGBD.

4.4 Padrão de Projeto Data Access Object (DAO)

Quando temos o nosso primeiro contato com JDBC, normalmente nos é ensinado a colocar todo o código SQL que vai executar uma determinada operação em um método que trata o "clique" de um botão. Sendo assim, imagine uma interface gráfica

²Note que a senha do usuário root no meu ambiente de desenvolvimento é root também, mas no seu ela pode ser outra ou mesmo estar vazia. Certifique-se de que essa informação esteja correta.

de uma aplicação *desktop* (em Swing) onde poderíamos criar, alterar e/ou excluir países. Quando implementamos o botão responsável por criar um novo país, somos ensinados a inserir todo o código SQL para fazer isso, sendo que esse código é implementado usando uma instrução INSERT. O mesmo aconteceria para os botões alterar e excluir. Será que essa é a melhor solução? Será que a classe que implementa nossa interface gráfica tem que ter essa responsabilidade, ou seja, lidar com SQL? E se por algum motivo nós quiséssemos usar o código para criar um novo país em alguma outra janela? Teríamos que copiar o código de inserção novamente? Tenho certeza que para essa última pergunta você já deve ter respondido mentalmente que "NÃO!", pois podemos criar métodos que executariam essa operação, mas então te pergunto: Como fazer isso?

Para resolver esse problema, existe um padrão de projeto onde a ideia é isolar todo acesso ao banco de dados em classes que seriam responsáveis em fazer a comunicação entre a aplicação Java, ou qualquer aplicação escrita usando uma linguagem orientada a objetos, e o banco de dados. Esse padrão se chama *Data Access Object* (DAO), sendo este um dos mais famosos. Vamos aprender como implementá-lo?

Um objeto do tipo DAO deve ter a capacidade de executar as operações básicas sobre uma determinada tabela de um banco de dados. Essas operações são comumente chamadas de "CRUD" que vem de "*Create, Read, Update e Delete*" (Criar, Ler, Atualizar e Excluir). Praticamente cada tabela do nosso banco de dados terá no lado da aplicação uma classe implementada que representará um registro da tabela (tabela "pais", classe Pais) por meio de um objeto do tipo em questão, além de ter uma classe DAO que vai manipular esses objetos. Como precisamos definir essas quatro operações básicas que cada DAO vai conter, vamos criar uma classe abstrata que servirá de modelo.

No pacote "padroesempratica", crie um novo pacote chamado "dao". Dentro do pacote "padroesempratica.dao", crie uma classe chamada "DAO" e copie o código apresentado na Listagem 4.4.

```
Listagem 4.4: Implementação da classe abstrata DAO (padroesempratica/dao/DAO.java)

1 package padroesempratica.dao;
2 import java.sql.Connection;
4 import java.sql.SQLException;
5 import java.util.List;
6 import padroesempratica.jdbc.ConnectionFactory;
7
8 /**
```

```
* DAO genérico.
10
    * @author Prof. Dr. David Buzatto
11
12
public abstract class DAO<Tipo> {
       // cada DAO terá uma conexão.
       private Connection conexao;
16
17
       /**
18
        * Construtor do DAO.
19
        * É nesse construtor que a conexão é criada.
20
21
        * @throws SQLException
22
        */
23
       public DAO() throws SQLException {
24
25
           /*
26
            * Usa-se o método getConnection() da fábrica de conexões,
27
            * para criar uma conexão para o DAO.
28
           conexao = ConnectionFactory.getConnection();
30
31
       }
32
       /**
        * Método para obter a conexão criada.
35
36
        * @return Retorna a conexão.
37
       public Connection getConnection() {
           return conexao;
40
       }
41
42
       /**
43
        * Método para fechar a conexão aberta.
        * Othrows SQLException Caso ocorra algum erro
        * durante o fechamento da conexão.
47
48
       public void fecharConexao() throws SQLException {
49
           conexao.close();
50
```

```
}
51
52
       /**
53
        st Método abstrato para salvar uma instância de uma
54
        * entidade da base de dados.
55
56
        * É o "C" do CRUD.
57
58
        * Oparam obj Instância do objeto da entidade a ser salvo.
59
        * Othrows SQLException Caso ocorra algum erro durante a gravação.
60
61
       public abstract void salvar( Tipo obj ) throws SQLException;
62
       /**
64
65
        * Método abstrato para atualizar uma instância de uma
        * entidade da base de dados.
66
67
        * É o "U" do CRUD.
68
        * @param obj Instância do objeto da entidade a ser atualizado.
70
71
        st Othrows SQLException Caso ocorra algum erro durante a atualização.
72.
       public abstract void atualizar( Tipo obj ) throws SQLException;
73
74
75
        * Método abstrato para excluir uma instância de uma
76
        * entidade da base de dados.
77
78
        * \acute{E} o "D" do CRUD.
79
80
        * Oparam obj Instância do objeto da entidade a ser salvo.
81
        * Othrows SQLException Caso ocorra algum erro durante a exclusão.
82
83
       public abstract void excluir( Tipo obj ) throws SQLException;
84
85
       /**
86
        * Método abstrato para obter todas as instâncias de uma
87
        * entidade da base de dados.
88
89
        * É o "R" do CRUD.
90
91
        * Oreturn Lista de todas as instâncias da entidade.
92
```

```
* Othrows SQLException Caso ocorra algum erro durante a consulta.
93
94
       public abstract List<Tipo> listarTodos() throws SQLException;
95
96
97
         * Método abstrato para obter uma instância de uma
98
         * entidade pesquisando pelo seu atributo identificador.
100
         * É o "R" do CRUD.
101
102
         * @param id Identificador da instância a serv obtida.
103
         * @return Instância relacionada ao id passado, ou null caso não seja
104
         * encontrada.
105
         * Othrows SQLException Caso ocorra algum erro durante a consulta.
106
107
       public abstract Tipo obterPorId( int id ) throws SQLException;
108
109
110
  | }
```

Copiou o código? Ótimo! Vamos entendê-lo. Na linha 13 definimos uma classe abstrata³ chamada DAO que diz que toda classe que for implementá-la deve fornecer um tipo genérico chamado de Tipo. As construções entre < e > são chamadas de "Tipos Genéricos" em Java. Note que o tipo Tipo é usado em todo o corpo da classe. Está confuso? Acalme-se, logo você vai entender.

Na linha 16 é declarada uma variável de instância que referenciará uma conexão, que sempre será obtida usando o método <code>getConnection()</code> definido na linha 39. Observe que poderíamos ter declarado essa conexão como <code>protected</code>, mas vamos deixá-la como <code>private</code> e usar o método <code>getConnection()</code> para obtê-la. Na linha 49 é definido o método para fechar a conexão, afinal, sempre depois de usarmos uma conexão, precisamos fechá-la.

Veja que no construtor da classe a conexão é obtida usando a fábrica que criamos na Seção anterior! Esse construtor será executado quando instanciarmos os objetos das classes que estenderem esse DAO, então, quando criarmos nossos objetos DAO, uma conexão com o banco de dados será estabelecida.

A partir da linha 62 são definidos todos os métodos CRUD deste DAO genérico. Note que temos dois métodos que correspondem à parte "R" do CRUD, onde um obtém todas as entidades cadastradas e outro obtém apenas uma usando como base seu identificador.

³Classes abstratas não podem ser instanciadas, são usadas como modelos.

Com o DAO genérico pronto, vamos implementar a classe que vai representar a tabela "pais". Crie um novo pacote chamado "entidades" dentro do pacote "padroesempratica". Dentro do pacote "padroesempratica.entidades", crie uma classe chamada "Pais" (sem as aspas). Os objetos dessa classe representarão registros da tabela "pais", sendo assim, precisamos que essa classe tenha os mesmos atributos da tabela correspondente. Lembre-se que na tabela "pais" nós definimos três colunas: id (INT), nome (VARCHAR) e sigla (VARCHAR). Na nossa classe, iremos usar o tipo int como tipo correspondente ao INT da tabela. Para o tipo VARCHAR, usaremos String. Veja na Listagem 4.5 como deve ficar a implementação parcial da classe Pais.

```
Listagem 4.5: Implementação parcial da classe Pais
   (padroesempratica/entidades/Pais.java)
  package padroesempratica.entidades;
2
3
   * Classe Pais.
4
5
    * @author Prof. Dr. David Buzatto
6
7
  public class Pais {
10
      private int id;
      private String nome;
11
      private String sigla;
12
13
  }
14
```

Tenho certeza que você se lembra da discussão sobre o padrão JavaBeans não é mesmo? Onde foi dito que devemos expor ao mundo "fora da classe" os atributos que nós queremos que possam ser configurados e obtidos por seus utilizadores? Da primeira vez que falamos sobre isso, nós implementamos manualmente cada método set e get correspondente a um campo privado não foi? Como essa tarefa é muito corriqueira, o NetBeans tem uma funcionalidade que faz isso automaticamente para nós. Com a classe implementada, como exibido na Listagem 4.5, clique com o botão direito no editor e escolha *Insert Code...*. Ao clicar nesta opção, uma pequena lista com o nome de *Generate* será exibida no editor. Nessa lista, escolha a opção *Getter and Setter...*. Adivinhe o que vamos fazer? Gerar os métodos get e set para cada campo da classe! Fazendo isso, um diálogo será exibido, mostrando todos os campos privados da classe. Marque cada um dos campos clicando na caixa de seleção

correspondente, ou então, a classe inteira e clique no botão *Generate*. Veja o que aconteceu! A IDE gerou o código dos gets e sets para nós! Segue na Listagem 4.6 o código completo da classe Pais.

Listagem 4.6: Implementação da classe Pais (padroesempratica/entidades/Pais.java)

```
package padroesempratica.entidades;
  /**
3
   * Classe Pais.
5
    * @author Prof. Dr. David Buzatto
  public class Pais {
       private int id;
10
       private String nome;
11
       private String sigla;
12
13
       public int getId() {
14
15
           return id;
       }
16
17
       public void setId( int id ) {
18
19
           this.id = id;
       }
20
21
       public String getNome() {
22
           return nome;
23
       }
24
       public void setNome( String nome ) {
26
           this.nome = nome;
27
       }
28
29
       public String getSigla() {
30
           return sigla;
31
       }
32
33
       public void setSigla( String sigla ) {
34
           this.sigla = sigla;
35
```

```
36 }
37
38 }
```

Muito legal não é mesmo? Agora que temos a classe que representa a estrutura da tabela "pais" da nossa base de dados, vamos implementar nosso primeiro DAO concreto, ou seja, uma classe que vai estender a classe abstrata DAO. Novamente, no pacote "padroesempratica.dao", crie uma classe chamada "PaisDAO" (sem as aspas). Essa classe irá lidar com os objetos do tipo Pais, fazendo a ponte entre nossos objetos e o banco de dados. Com a classe criada, copie o código apresentado na Listagem 4.7.

```
Listagem 4.7: Implementação parcial da classe PaisDAO
   (padroesempratica/dao/PaisDAO.java)
  package padroesempratica.dao;
2
  import padroesempratica.entidades.Pais;
4
5
   * DAO para a classe Pais.
6
7
   * @author Prof. Dr. David Buzatto
8
  public class PaisDAO extends DAO<Pais> {
10
11
12 }
```

Veja que nosso PaisDAO vai estender DAO, informando como tipo a classe Pais (DAO<Pais>). Ao copiar o código, você perceberá que o NetBeans vai reclamar, dizendo que tem um erro na classe. O nome da classe ficará destacado em vermelho. Passe o mouse por cima do nome e aguarde. Será exibida a causa do erro. No erro, é dito que a classe PaisDAO não implementa todos os métodos abstratos da classe DAO e isso é verdade, visto que como estamos estendendo a classe DAO, precisamos implementar todos os métodos abstratos que foram definidos nela e ainda não fizemos isso. Teríamos então que implementar manualmente todos os métodos marcados como abstratos na classe DAO. Ao invés de fazermos isso manualmente, o NetBeans pode nos ajudar novamente. Veja que na linha do erro, à esquerda, é mostrada uma pequena lâmpada com uma bolinha vermelha. Clique nela. Ao clicar, o NetBeans vai listar as alternativas que ele pode executar para resolver o erro. Veja a Figura 4.4.

Figura 4.4: Implementando automaticamente os métodos abstratos de DAO

Fonte: Elaborada pelo autor

Clique na opção [Implement all abstract methods] e, novamente, como num passe de mágica, o NetBeans gera todo o esqueleto da classe para nós, criando uma implementação padrão para cada método abstrato da classe DAO. Mesmo ao fazer isso, o NetBeans continua reclamando que existe um erro. Nesse caso, é dito que é lançada uma exceção no construtor padrão⁴. Você se lembra que lá no DAO genérico nós temos um construtor que cria a conexão e que ele lança uma SQLException? Pois bem, quando criamos um objeto de uma determinada classe, o construtor da superclasse da classe em questão a primeira coisa que será executada. Como estendemos DAO em PaisDAO, ao tentarmos instanciar um objeto do tipo PaisDAO, o construtor de PaisDAO será executado, além do construtor de DAO, que é sua superclasse. Como o construtor de DAO lança uma exceção caso ocorra algum problema, nós precisamos ou tratar ou dizer que o construtor de PaisDAO também lança esse tipo de exceção. Nós iremos usar a segunda abordagem. Para isso, basta implementar o construtor padrão de PaisDAO e dizer que ele lança esse tipo de exceção. Sendo assim, segue na Listagem 4.8 a implementação que temos até agora da classe PaisDAO.

⁴O construtor padrão é o construtor que não tem nenhum parâmetro.

```
Listagem 4.8: Implementação parcial da classe PaisDAO com o construtor
   padrão (padroesempratica/dao/PaisDAO.java)
package padroesempratica.dao;
2
3 import java.sql.SQLException;
4 import java.util.List;
5 import padroesempratica.entidades.Pais;
  /**
7
   * DAO para a classe Pais.
   * @author Prof. Dr. David Buzatto
10
11
  public class PaisDAO extends DAO<Pais> {
13
       public PaisDAO() throws SQLException {
14
           super();
15
       }
16
17
       @Override
18
       public void salvar( Pais obj ) throws SQLException {
19
           throw new UnsupportedOperationException( "Not supported yet." );
20
       }
21
22
       @Override
23
       public void atualizar( Pais obj ) throws SQLException {
24
           throw new UnsupportedOperationException( "Not supported yet." );
25
       }
26
27
       @Override
28
       public void excluir( Pais obj ) throws SQLException {
29
           throw new UnsupportedOperationException( "Not supported yet." );
30
       }
31
32
       @Override
33
       public List<Pais> listarTodos() throws SQLException {
34
           throw new UnsupportedOperationException( "Not supported yet." );
35
       }
36
37
       @Override
38
       public Pais obterPorId( int id ) throws SQLException {
39
```

```
throw new UnsupportedOperationException( "Not supported yet." );

throw new UnsupportedOperationException( "Not supported yet." );

throw new UnsupportedOperationException( "Not supported yet." );

throw new UnsupportedOperationException( "Not supported yet." );
```

Muito bom! Até agora preparamos toda o esqueleto do nosso PaisDAO, mas SQL que é bom, nada. Vamos agora implementar nosso primeiro método do CRUD, o "salvar". Antes de implementar o método "salvar", importe a classe java.sql. PreparedStatement. Na Listagem 4.9 pode ser vista a implementação do método salvar da classe PaisDAO.

```
Listagem 4.9: Implementação do método "salvar" da classe PaisDAO
   (padroesempratica/dao/PaisDAO.java)
  package padroesempratica.dao;
3 import java.sql.PreparedStatement;
4 import java.sql.SQLException;
5 import java.util.List;
  import padroesempratica.entidades.Pais;
7
8
   * DAO para a classe Pais.
9
10
    * @author Prof. Dr. David Buzatto
11
12
  public class PaisDAO extends DAO<Pais> {
13
14
       public PaisDAO() throws SQLException {
15
           super();
16
17
       }
18
       @Override
19
       public void salvar( Pais obj ) throws SQLException {
20
21
           String sql = """
22
                        INSERT INTO pais( nome, sigla )
                        VALUES( ?, ? );
24
                        0.0\,0 .
25
26
           PreparedStatement stmt = getConnection().prepareStatement( sql );
27
           stmt.setString( 1, obj.getNome() );
28
```

```
stmt.setString( 2, obj.getSigla() );
29
30
           stmt.executeUpdate();
31
           stmt.close();
32
33
       }
34
35
       @Override
36
       public void atualizar( Pais obj ) throws SQLException {
37
           throw new UnsupportedOperationException( "Not supported yet." );
38
       }
39
40
       @Override
41
       public void excluir( Pais obj ) throws SQLException {
42
           throw new UnsupportedOperationException( "Not supported yet." );
43
       }
44
45
       @Override
46
       public List<Pais> listarTodos() throws SQLException {
47
           throw new UnsupportedOperationException( "Not supported yet." );
48
       }
49
50
       @Override
51
       public Pais obterPorId( int id ) throws SQLException {
52
           throw new UnsupportedOperationException( "Not supported yet." );
       }
54
55
  }
56
```

Vamos analisar o código para ver o que está acontecendo. Na linha 20 está definida a assinatura do método. O método salvar recebe um parâmetro do tipo Pais, sendo que os dados do objeto passado nesse parâmetro serão persistidos no banco de dados. Entre as linhas 22 e 25 é definida a instrução INSERT do banco de dados. Note que a String que define o código SQL está delimitada entre um par de três aspas duplas. Essa construção é chamada de bloco de texto (*text block*) e foi adicionada na linguagem Java a partir da versão 15. A vantagem dessa sintaxe é permitir que escrevamos Strings quebrando linhas, sem a necessidade de ficar concatenando vários pedaços de uma String comprida linha a linha, auxiliando principalmente na legibilidade e manutenção do código. Como a coluna id da tabela pais foi definida como autoincremento, nós não precisamos fornecê-la. Ao invés de definirmos manualmente os valores das colunas nome e sigla, perceba que utilizamos sinais de interrogação,

indicando que no lugar de cada ponto de interrogação será trocado pelo valor correto. Na linha 25 é criado um PreparedStatement a partir da conexão do DAO usando o código SQL que foi definido entre as linhas 22 e 25. Na linha 28, o primeiro parâmetro do código SQL (primeiro ponto de interrogação) é "trocado" pelo valor retornado pelo método getNome() do objeto referenciado por obj que é do tipo Pais. A mesma coisa acontece na linha 29, onde o segundo parâmetro do código SQL (segundo ponto de interrogação) é "trocado" pelo valor retornado pelo método getSigla(). Com o PreparedStatement configurado, na linha 31 mandamos que seja executado o PreparedStatement. Por fim, na linha 32, fechamos o PreparedStatement. Fácil não é?

Vamos testar? No pacote "padroesempratica.testes", crie uma classe chamada TestePaisDAO e copie o código da Listagem 4.10.

```
Listagem 4.10: Código de teste para o PaisDAO (padroesempratica/
  testes/TestePaisDAO.java)
  package padroesempratica.testes;
3 import java.sql.SQLException;
  import padroesempratica.dao.PaisDAO;
  import padroesempratica.entidades.Pais;
7
    * Testes da classe PaisDAO.
8
9
    * @author Prof. Dr. David Buzatto
10
  public class TestePaisDAO {
12
13
       public static void main( String[] args ) {
14
15
           Pais pais = new Pais();
16
           pais.setNome( "Brasil" );
17
           pais.setSigla( "BR" );
18
19
           PaisDAO dao = null;
20
21
           try {
23
               dao = new PaisDAO();
24
               dao.salvar( pais );
25
```

```
26
            } catch ( SQLException exc ) {
27
                exc.printStackTrace();
28
            } finally {
29
30
                if ( dao != null ) {
31
32
                     try {
33
                         dao.fecharConexao();
34
                     } catch ( SQLException exc ) {
35
                         System.err.println( "Erro ao fechar a conexão!" );
36
                         exc.printStackTrace();
37
                     }
38
39
                }
40
41
            }
42
43
       }
45
46 }
```

Copiou o código? Execute a classe (botão direito no arquivo, [Run File] ou <Shift+F6>). Se tudo estiver correto, a classe será compilada e executada e nenhum erro será emitido. Fazendo isso, um novo registro na tabela "pais" será inserido. Vamos confirmar isso? No MySQL Workbench deve haver um editor SQL já aberto. Se não houver, abra um clicando no ícone com uma folha com a sigla SQL e um sinal de +. Confirme também se a base de dados "testes_padroes" está configurado como padrão ou está ativa. No editor, digite SELECT * FROM pais; e clique no botão que tem um desenho de um raio na barra de ferramentas da aba do editor. O código SQL digitado será executado e o resultado será exibido logo abaixo. Veja a Figura 4.5.

Query 1 ×

1 • SELECT * FROM pais;

Figura 4.5: Fazendo uma consulta do MySQL Workbench



Fonte: Elaborada pelo autor

Muito bem! Nosso método salvar de PaisDAO está funcionando corretamente. Vamos analisar o código da Listagem 4.10. Entre as linhas 16 e 18 instanciamos e configuramos um objeto do tipo Pais. O nome desse país é Brasil e a sigla é BR. Na linha 20, declaramos uma referência do tipo PaisDAO que foi inicializada com null. Como todo o código dos métodos do DAO podem lançar uma exceção do tipo SQLException, temos que usar um bloco try Na linha 24 instanciamos o PaisDAO e na linha 25 passamos o objeto do tipo Pais que criamos para o método "salvar" do DAO, que por sua vez vai executar o código SQL que definimos. Caso ocorra algum problema durante a execução de uma dessas duas linhas, o catch que ouve exceções do tipo SQLException captura o erro e manda mostrar esses problemas dando um printStackTrace() no objeto que representa a exceção. Por fim, temos um bloco finally que trata do fechamento da conexão. Sempre quando usarmos um DAO, precisamos fechar sua conexão quando terminarmos de usar. Na linha 31 verifica-se se o dao aponta para [null]. Se não apontar, tenta fechar a conexão, que também pode lançar uma exceção do tipo SQLException e que é tratada dentro do bloco try que está aninhado no [finally].

Agora que já criamos e testamos nosso primeiro método do PaisDAO, vamos implementar o restante dos métodos. Os métodos de pesquisa ("R" do CRUD) serão um pouco diferente, sendo assim, eu os discutirei depois. Vamos lá, copie o restante do código para que seu PaisDAO fique igual ao da Listagem 4.11.

```
Listagem 4.11: Implementação da classe PaisDAO (padroesempratica/
   dao/PaisDAO.java)
package padroesempratica.dao;
2
3 import java.sql.PreparedStatement;
4 import java.sql.ResultSet;
5 import java.sql.SQLException;
6 import java.util.ArrayList;
7 import java.util.List;
8 import padroesempratica.entidades.Pais;
  /**
10
   * DAO para a classe Pais.
11
12
    * @author Prof. Dr. David Buzatto
13
14
  public class PaisDAO extends DAO<Pais> {
15
16
       public PaisDAO() throws SQLException {
17
           super();
18
       }
19
20
       @Override
21
       public void salvar( Pais obj ) throws SQLException {
22
23
           String sql = """
24
                        INSERT INTO pais( nome, sigla )
25
                        VALUES( ?, ? );
26
27
28
           PreparedStatement stmt = getConnection().prepareStatement( sql );
29
           stmt.setString( 1, obj.getNome() );
30
           stmt.setString( 2, obj.getSigla() );
31
32
           stmt.executeUpdate();
33
           stmt.close();
34
```

```
35
       }
36
37
       @Override
38
       public void atualizar( Pais obj ) throws SQLException {
39
40
           String sql = """
                         UPDATE pais
42
                         SET
43
                             nome = ?,
44
                              sigla = ?
45
46
                          WHERE
                              id = ?;
47
                          0.00
48
49
           PreparedStatement stmt = getConnection().prepareStatement( sql );
50
           stmt.setString( 1, obj.getNome() );
51
           stmt.setString( 2, obj.getSigla() );
52
           stmt.setInt( 3, obj.getId() );
54
           stmt.executeUpdate();
55
           stmt.close();
56
57
       }
58
       @Override
       public void excluir( Pais obj ) throws SQLException {
61
62
           String sql = "DELETE FROM pais WHERE id = ?;";
63
64
           PreparedStatement stmt = getConnection().prepareStatement( sql );
           stmt.setInt( 1, obj.getId() );
66
67
           stmt.executeUpdate();
68
           stmt.close();
69
70
       }
71
72
       @Override
73
       public List<Pais> listarTodos() throws SQLException {
74
75
           List<Pais> lista = new ArrayList<>();
76
```

```
String sql = "SELECT * FROM pais;";
77
78
            PreparedStatement stmt = getConnection().prepareStatement( sql );
79
            ResultSet rs = stmt.executeQuery();
80
81
            while ( rs.next() ) {
82
83
                Pais pais = new Pais();
84
                pais.setId( rs.getInt( "id" ) );
85
                pais.setNome( rs.getString( "nome" ) );
86
                pais.setSigla( rs.getString( "sigla" ) );
87
88
                lista.add( pais );
89
90
            }
91
92
            rs.close();
93
            stmt.close();
94
95
            return lista;
96
97
        }
98
99
        @Override
100
        public Pais obterPorId( int id ) throws SQLException {
101
102
            Pais pais = null;
103
            String sql = "SELECT * FROM pais WHERE id = ?;";
104
105
            PreparedStatement stmt = getConnection().prepareStatement( sql );
            stmt.setInt( 1, id );
107
108
109
            ResultSet rs = stmt.executeQuery();
110
            if ( rs.next() ) {
111
112
                pais = new Pais();
113
                pais.setId( rs.getInt( "id" ) );
114
                pais.setNome( rs.getString( "nome" ) );
115
                pais.setSigla( rs.getString( "sigla" ) );
116
117
            }
118
```

Não se esqueça de importar as classes java.sql.ResultSet, java.util.ArrayList e java.util.List. Para finalizar essa seção, vamos discutir o código da Listagem 4.11. O método (listarTodos()) retorna uma lista que pode conter apenar objetos do tipo Pais. Essa lista que será retornada é instanciada e inicializada na linha 76. Neste momento, temos a lista de objetos do tipo Pais, só que ela ainda está vazia. Na linha 75, cria-se o PreparedStatement com o SQL que foi definido na linha 77 e, na linha 79, executamos o PreparedStatement usando o método (executeQuery()), que retorna um objeto do tipo ResultSet. Os objetos do tipo ResultSet representam os resultados que são retornados em consultas SQL (instrução SELECT). Na linha 82 usamos um [while], que é executado enquanto [rs.next()] retornar [true]. Na primeira vez que [rs.next()] é invocado, o ponteiro de registros do ResultSet passa a apontar para o primeiro resultado obtido na consulta. Dentro do while entre as linhas 84 e 87, nós criamos um objeto do tipo Pais e configuramos seus dados a partir dos dados obtidos no ResultSet, usando o método apropriado para cada tipo de cada coluna. Na linha 89, o objeto Pais que acabou de ser criado e configurado é inserido na lista. O corpo do while é repetido até que rs.next() retorne false, ou seja, quando todos os resultados da consulta foram obtidos. Nesse ponto, temos a lista de objetos completa. Nas linhas 93 e 94 são fechados o ResultSet e o PreparedStatement. Por fim, na linha 96, a lista com os objetos do tipo Pais é retornada.

Note que o método obterPorId(int id) tem uma implementação muito parecida com o método listarTodos(), só que no caso do obterPorId, apenas um objeto pode ser retornado, visto que cada registro da tabela pais tem um identificador específico. Caso o id passado como parâmetro não represente um registro da tabela pais, o método retorna null.

Agora, como exercício, modifique a classe TestePaisDAO para testar todos os métodos que foram implementados. Verifique se todos estão corretos usando o MySQL Workbench para comparar os resultados obtidos. Tente entender a implementação de

cada método do CRUD. Não fez o exercício? Então faça! Ele não é opcional.

Viu só que legal? Com tudo isso que fizemos, conseguimos isolar toda a camada de persistência da nossa aplicação. Para cada tabela nova que tivermos na base de dados, precisamos implementar a classe correspondente à tabela (que chamamos de entidade) e a classe DAO que vai manipular os objetos da classe criada e fazer o intercâmbio entre o mundo orientado a objetos com o mundo relacional. Essa comunicação entre objetos e o modelo relacional é chamada de Object-Relational Mapping (ORM). Existem alguns *frameworks* que automatizam esse processo, gerando todo o código SQL automaticamente para nós. Infelizmente não teremos tempo para aprendê-los. Um desses *frameworks* é o Hibernate.

(i) | Saiba Mais

Quer saber mais sobre ORM? Dê uma olhada nesses *links*: http://en.wikipedia.org/wiki/Mapeamento_objeto-relacional, http://en.wikipedia.org/wiki/Object-relational_mapping, http://www.hibernate.org/

Estamos quase acabando! Vamos para o último padrão de projeto.

4.5 Padrão de Projeto *Model-View-Controller* (MVC)

O padrão MVC é um padrão muito utilizado no desenvolvimento de aplicações que utilizam linguagens orientadas a objetos. Este padrão ajuda os desenvolvedores a separar as regras de negócio da aplicação, ou seja, as regras de como os dados devem se armazenados, qual a ordem que devem ser gravados etc., da lógica de apresentação desses dados, ou seja, como eles serão exibidos aos usuários do sistema. Essa separação se dá utilizando três camadas:

- Model (Modelo): A camada Model é usada para organizar como os dados são armazenados e gerenciados dentro da aplicação. No exemplo que construímos durante este Capítulo, as classes Pais e PaisDAO fazem parte do modelo;
- *View* (Visualização): Essa camada organiza os recursos utilizados para exibir ao usuário os dados que são gerenciados pela camada de modelo;
- *Controller* (Controle): A camada *Controller*, como o próprio nome já diz, é responsável por controlar o fluxo de execução da aplicação.

A partir dessas definições, nós podemos ver nitidamente, nos exemplos que estamos construindo desde o início do livro, qual recurso faz parte de cada camada. A entidade Pais faz parte do modelo. Uma JSP faz parte da visualização, pois é usada pelo usuário tanto para inserir dados no sistema quanto para visualizá-los. Os Servlets que construímos fazem parte do controle, pois são eles que recebem os dados, os

4.6. RESUMO 93

utilizam usando a camada de modelo e decidem para onde o fluxo da aplicação deve ser direcionado, por exemplo, uma página JSP que vai exibir a saída gerada por eles.

Essa foi uma pequena introdução do MVC, pois durante Capítulo 5 nós iremos usá-lo extensivamente no projeto que iremos criar. Tenho certeza que você vai achar muito legal e útil! Como de costume, pratique o que você aprendeu durante este Capítulo com as atividades de aprendizagem.

4.6 Resumo

Neste Capítulo demos um passo muito importante na nossa vida como desenvolvedores. Nós aprendemos que existem os chamados "padrões de projeto" ou "*design patterns*", que são padrões que guiam os desenvolvedores na solução de problemas recorrentes através do uso de soluções de sucesso. Estudamos os padrões *Factory* e DAO implementando exemplos e aprendemos o básico do funcionamento do padrão MVC. No Capítulo 5 iremos implementar um projeto completo usando esses três padrões.

4.7 Exercícios

Exercício 4.1: Defina, com suas palavras, o padrão Factory.

Exercício 4.2: Defina, com suas palavras, o padrão DAO.

Exercício 4.3: Defina, com suas palavras, o padrão MVC.

4.8 Projetos

Projeto 4.1: Da mesma forma que fizemos para a tabela pais, crie uma tabela na base de dados testes_padroes, usando o MySQL Workbench, com o nome de "fruta". Essa tabela deve ter como colunas um campo identificador (INT), um campo que armazenará o nome da fruta (VARCHAR) e um campo para armazenar a cor predominante da fruta (VARCHAR). Implemente, no projeto que criamos durante este Capítulo, a entidade Fruta e a classe FrutaDAO. Crie uma classe de testes chamada TesteFrutaDAO para testar os métodos do DAO da fruta.

Projeto 4.2: Repita o Projeto 4.1, só que agora para a tabela "carro". Um carro deve ter um identificador, um nome (VARCHAR), um modelo (VARCHAR) e um ano de fabricação (INT).

Projeto 4.3: Repita o Projeto 4.1, só que agora para a tabela "produto". Um produto deve ter um identificador, uma descrição (VARCHAR) e uma quantidade em estoque (INT).

SISTEMA PARA CONTROLE DE CLIENTES

"É fazendo que se aprende a fazer aquilo que se deve aprender a fazer".

Aristóteles



ESTE Capítulo teremos como objetivos entender e realizar a construção de uma aplicação Web em Java completa.

5.1 Introdução

Chegou a hora de colocar em prática tudo o que aprendemos nos Capítulos anteriores com o objetivo de criar uma aplicação Web em Java completa. Iremos passar por todos os passos do desenvolvimento da aplicação para que no Capítulo 6, você possa usar um conjunto de requisitos para desenvolver um sistema sozinho. Vamos começar?

5.2 Analisando os Requisitos

Imagine que fomos contratados para criar um sistema para controle de cadastro de clientes. Esse sistema deve manter vários dados de um cliente: nome, sobrenome, data de nascimento, CPF, e-mail, logradouro, número, bairro, cidade e CEP. O contratante

também deseja que seja possível manter um cadastro de cidades e de estados, sendo que as cidades devem ter um nome e um estado, enquanto um estado deve ter um nome e uma sigla. Cada um dos cadastros (cliente, cidade e estado), deve conter as funcionalidades de inserir, alterar e excluir um determinado registro.

Vamos analisar esses requisitos. Primeiramente, vamos identificar os tipos de entidades que farão parte do sistema, fazendo a seguinte pergunta: Quais são os tipos de "coisas" que o sistema deve gerenciar? O sistema deve manter um cadastro de Clientes, um cadastro de Cidades e um cadastro de Estados. Sendo assim, identificamos três entidades, ou seja, Cliente, Cidade e Estado.

Cada um desses tipos de entidade tem uma determinada lista de características ou atributos. Vamos organizá-las em uma tabela. Veja essa organização na Tabela 5.1.

Tabela 5.1: Atributos dos tipos de entidade

- nome		
- sobrenome		
- data de nascimento		
- CPF		
- e-mail		
- logradouro		
- número		
- bairro		
- Cidade		
- CEP		
- nome		
- Estado		
- nome		
- sigla		

Fonte: Elaborada pelo autor

Sabemos que esses tipos de entidade que foram identificados se tornarão tabelas na nossa base de dados relacional não é mesmo? Cada atributo de cada tipo de entidade se tornará uma coluna na tabela correspondente. Sabemos também que cada registro de uma determinada tabela precisa ser diferenciado dos outros não é mesmo? Para isso analisamos as tabelas até que consigamos identificar as chaves primárias de cada uma delas. Uma chave primária é o conjunto mínimo de um ou mais atributos de um determinado tipo de entidade que garante a unicidade de um registro, sendo assim, precisamos encontrar, na lista de características de cada entidade, uma ou mais características que, usadas em conjunto, garantem que um registro é diferente

de outro. Tomemos como exemplo o tipo de entidade Estado. Veja na Tabela 5.2 uma lista de registros da tabela estado do nosso provável banco de dados.

Tabela 5.2: Exemplos de registros da tabela "estado"

Tabela: estado			
nome	sigla		
São Paulo	SP		
Rio de Janeiro			
Minas Gerais			
	•••		

Fonte: Elaborada pelo autor

O que diferencia um estado de outro? Se usarmos os atributos nome e sigla, nós sempre teremos um estado diferente do outro, ou seja, não seria permitido a criação de um novo estado que tivesse o mesmo nome e a mesma sigla do que ume estado que já existe na tabela, concorda? Agora pense, e se usarmos apenas o nome ou somente a sigla, qual seria suficiente? Temos então três opções para definir a chave primária dessa tabela. Podemos usar nome e sigla, somente nome ou somente sigla. Dentre essas três opções, qual ou quais são as que têm o menor conjunto de atributos? Somente nome ou somente sigla, correto? Como temos duas opções, podemos escolher qualquer uma delas, desde que, para o cenário ou "minimundo" que está sendo modelado, um desses atributos garanta a unicidade de tupla. Vamos dizer que nós escolhemos a opção de definir a chave primária da tabela usando o atributo sigla. Legal, agora sabemos que um Estado é diferenciado do outro pela sua sigla, então não pode existir mais de um registro com a mesma sigla.

Agora temos outro problema: o desempenho do banco de dados. Quando criarmos a tabela cidade, esta vai ter que referenciar a tabela estado, usando uma coluna que vai ter o mesmo tipo da coluna que representa a chave primária da tabela estado. Essa coluna, como você deve se lembrar, é denominada chave estrangeira. Como cada estado tem uma sigla de dois caracteres, sempre que um estado for referenciado em um registro da tabela cidade, essa referência vai ter que ter o mesmo valor do registro contido na tabela sigla. Apesar de essa abordagem funcionar, nós podemos atacar esse problema de outra forma. Podemos definir que a coluna sigla tem valor único (*unique*) nos registros da tabela estado e então criar uma chave primária que contém apenas um número. Essa chave primária é chamada de chave artificial, ou *surrogate*, sendo que normalmente é chamada de id (identificador). Note que como o próprio nome diz, essa chave é artificial. O que vai garantir a unicidade dos registros é a configuração de cada uma das colunas.

O que ganhamos com isso? Ganhamos desempenho, pois estamos usando números

para referenciar colunas de outras tabelas, não Strings. Imagine definir a chave primária de um Cliente como CPF. Ao precisarmos referenciar um cliente em outra tabela, digamos uma tabela de pedidos, precisaríamos ter uma cópia do CPF do cliente em cada pedido, gastando cerca de 11 a 14 caracteres (um CPF é no formato 000.000.000-00), ao passo que poderíamos usar apenas um número! Veja, se cada caractere ocupar 4 bytes em disco/memória, um CPF de 11 dígitos (só os números) ocupará 44 bytes (352 bits), enquanto um número inteiro provavelmente ocupará 4 ou 8 bytes (32 ou 64 bits). Pense nas implicações! Sabendo de tudo isso, podemos partir para o projeto do banco de dados.

5.3 Projetando Banco de Dados

Não irei documentar aqui todo o processo de projeto do banco de dados, que vai desde a análise dos requisitos, passando pela criação do Diagrama Entidade-Relacionamento (DER) até a implementação do modelo físico, pois este não é o objetivo deste livro, mas note que no desenvolvimento de um sistema esses passos são normalmente realizados. Iremos partir diretamente para a implementação do modelo físico usando como base o DER modelado no MySQL Workbench. Antes de criarmos o código escrito em *Structured Query Language* (SQL) para a criação da estrutura da nossa base de dados, vamos organizar os atributos das nossas tabelas –que são baseadas nas características dos tipos de entidade– e especificar suas características. Veja a Tabela 5.3.

Tabela 5.3: Detalhamento das colunas de cada tabela

Tabela	Coluna	Tipo	É único?	
	id*	INT	X	
cliente	nome	VARCHAR(45)		
	sobrenome	VARCHAR(45)		
	data_nascimento	DATE		
	cpf	VARCHAR(14)	X	
	email	VARCHAR(60)		
	logradouro	VARCHAR(50)		
	numero	VARCHAR(6)		
	bairro	VARCHAR(30)		
	cep	VARCHAR(9)		
	cidade_id**	INT		
	id*	INT	X	
cidade	nome	(VARCHAR(30)		
	estado_id**	INT		
estado	id*	INT	X	
	nome	VARCHAR(30)		
	sigla	VARCHAR(2)	X	
* chave primária				
** chave estrangeira				
todas as colunas são não nulas (NOT NULL)				

Fonte: Elaborada pelo autor

Usei o MySQL Workbench para criar um DER com essas tabelas. Veja como ficou na Figura 5.1.

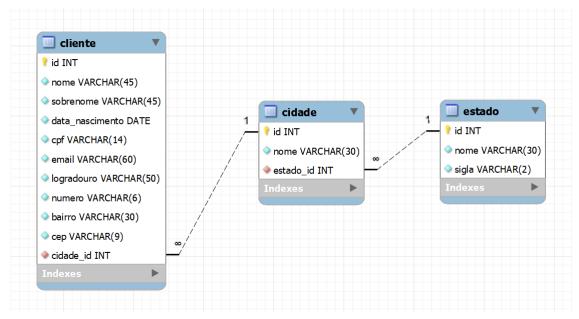


Figura 5.1: DER com as tabelas apresentadas

Fonte: Elaborada pelo autor

Vamos agora implementar o banco de dados. No MySQL Workbench, crie uma nova base de dados com o nome de cadastro_clientes como feito na Seção 4.2. Com a base criada, torne-a padrão, abra uma aba para digitar código SQL e copie o código da Listagem 5.1 no editor e execute o *script*.

```
Listagem 5.1: Script SQL para criação da tabela "estado"

CREATE TABLE IF NOT EXISTS estado (
   id INT NOT NULL AUTO_INCREMENT,
   nome VARCHAR(30) NOT NULL,
   sigla VARCHAR(2) NOT NULL,
   PRIMARY KEY (id),
   UNIQUE INDEX sigla_UNIQUE (sigla ASC)

PENGINE = InnoDB;
```

Faça o mesmo processo para a Listagem 5.2 e para a Listagem 5.3.

```
Listagem 5.2: Script SQL para criação da tabela "cidade"
  CREATE TABLE IF NOT EXISTS cidade (
    id INT NOT NULL AUTO_INCREMENT,
    nome VARCHAR(30) NOT NULL,
    estado_id INT NOT NULL,
    PRIMARY KEY (id),
5
    INDEX fk_cidade_estado_idx (estado_id ASC),
6
    CONSTRAINT fk cidade estado
7
      FOREIGN KEY (estado_id)
      REFERENCES estado (id)
      ON DELETE RESTRICT
10
      ON UPDATE CASCADE
12 ) ENGINE = InnoDB;
```

```
Listagem 5.3: Script SQL para criação da tabela "cliente"
  CREATE TABLE IF NOT EXISTS cliente (
     id INT NOT NULL AUTO_INCREMENT,
2
    nome VARCHAR(45) NOT NULL,
3
     sobrenome VARCHAR(45) NOT NULL,
     data_nascimento DATE NOT NULL,
5
    cpf VARCHAR(14) NOT NULL,
6
     email VARCHAR(60) NOT NULL,
7
     logradouro VARCHAR(50) NOT NULL,
8
    numero VARCHAR(6) NOT NULL,
     bairro VARCHAR(30) NOT NULL,
10
     cep VARCHAR(9) NOT NULL,
11
     cidade_id INT NOT NULL,
12
     PRIMARY KEY (id),
13
     UNIQUE INDEX cpf_UNIQUE (cpf ASC),
14
     INDEX fk_cliente_cidade_idx (cidade_id ASC),
     CONSTRAINT fk_cliente_cidade
      FOREIGN KEY (cidade_id)
17
      REFERENCES cidade (id)
18
       ON DELETE RESTRICT
19
       ON UPDATE CASCADE
21 ) ENGINE = InnoDB;
```

Ao fazer esses três passos, temos nossas três tabelas criadas dentro da base de dados. Expanda o banco cadastro_clientes e então expanda o nó *Tables*. Lá dentro estarão as três tabelas criadas. Muito bem, terminamos a implementação do banco.

Vamos agora criar um diagrama de classes para representar cada uma das nossas entidades, que serão mapeamentos das nossas tabelas no mundo orientado a objetos.

5.4 Criando o Diagrama de Classes

Para criar nosso diagrama de classes UML eu usei a ferramenta Astah UML. Existem diversas ferramentas de modelagem gratuítas. Infelizmente o Astah não poussi mais esse tipo de versão desde 2018. Você pode usar qualquer uma caso queria fazer a modelagem, mas ela não é obrigatória. Como já disse cada tabela do nosso banco de dados vai ter uma representação na nossa aplicação. Essa representação, como você já sabe, é criada usando classes. No diagrama de classes da Figura 5.2, você pode ver as três classes que representam nossas entidades. Note que cada classe contém uma série de atributos privados, denotados pelo sinal de "–", que representam as colunas da tabela e que o acesso a esses atributos é feito usando os métodos get e set correspondentes, omitidos no diagrama.

Cliente - id : int - nome : String - sobrenome : String Estado Cidade - cidade estado - dataNascimento : Date - cpf : String id : int - nome : String - email : String nome : int - sigla : String - logradouro : String - numero : String - bairro : String cep : String

Figura 5.2: Diagrama de classes das entidades

Fonte: Elaborada pelo autor

Outro detalhe é que criei apenas o diagrama das classes que são as entidades do sistema, não me preocupando com as outras classes que nosso sistema conterá. Novamente, como no exemplo do nosso banco de dados, em um sistema de verdade, normalmente são desenvolvidos diagramas de classes muito mais completos e complexos, dependendo do nível de representação que se deseja, bem como outros tipos de diagramas UML que forem necessários. Com tudo isso pronto, podemos partir para o desenvolvimento do sistema propriamente dito! Novamente, essa não é a nossa preocupação neste livro.

5.5 Construindo o Sistema

Agora nossa tarefa será criar o projeto no NetBeans. Para isso, abra o NetBeans e crie um novo projeto do tipo Java Web com o nome de "CadastroClientes" (sem as aspas). Siga os passos que você tem seguido em todos os projetos criados, além de, é claro, configurar a bibliteca do *driver* JDBC do MariaDB, como descrito no Capítulo 4. Em *Source Packages*, crie o pacote "cadastroclientes" (sem as aspas) e, dentro dele, os pacotes "controladores", "dao", "entidades", "jdbc" e "testes" dentro do pacote "cadastroclientes". Em *Web Pages*, crie os diretórios "css" e "formularios" (sem acento) e dentro deste, crie os diretórios "cidades", "clientes" e "estados". Apague o arquivo index.html e crie um arquivo JSP chamado index.jsp. Veja na Figura 5.3 como deve ficar a estrutura do projeto. Ignore os pacotes . . . filtros e . . . servicos por enquanto!

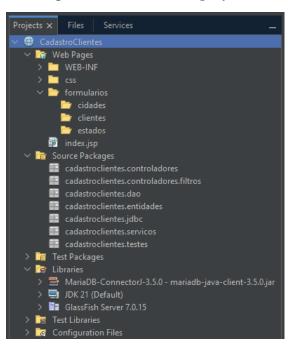


Figura 5.3: Estrutura do projeto

Fonte: Elaborada pelo autor

Com a estrutura configurada, vamos agora copiar algumas classes do projeto "Padro-esEmPatrica" que criamos no Capítulo 4. Para isso abra esse projeto, se ainda não estiver aberto, no NetBeans. Expanda o pacote "padroesempratica.jdbc", clique com o botão direito no arquivo "ConnectionFactory.java" e escolha *Copy*. Volte ao projeto "CadastroClientes", clique com o botão direito no pacote "cadastroclientes.jdbc", escolha *Paste* e então *Refactor Copy...*. Um diálogo será aberto. Clique

no botão *Refactor*). O arquivo será copiado para o projeto e as alterações que forem necessárias fazer no arquivo, como mudar a cláusula package, serão feitas pelo NetBeans. Faça o mesmo processo para o arquivo "DAO.java" contido no pacote "pradoesempratica.dao" do projeto "PadroesEmPratica", copiando-o no pacote "cadastroclientes.dao" do projeto "CadastroClientes". Talvez o NetBeans aponte um erro no arquivo depois da cópia. Para corrigir, abra o arquivo no editor e altere o import que está com problema.

Outro detalhe é que precisamos mudar a URL da nossa fábrica de conexões para fazer com que as conexões criadas sejam relativas à base de dados cadastro_clientes. Para isso, abra o arquivo "ConnectionFactory.java" do pacote "cadastroclientes.jdbc" e mude a URL de:

```
[jdbc:mariadb://localhost/testes_padroes]
Para:
[jdbc:mariadb://localhost/cadastro_clientes]
```

Agora vamos preparar toda a camada de persistência. No pacote "cadastroclientes.entidades", crie três classes: "Estado", "Cidade" e "Cliente" (sem as aspas). Nas três listagens a seguir estão listados os códigos-fonte das três classes. Note que estou omitindo os gets e os sets, mas isso não significa que eles não devam existir. Fica por sua conta criá-los ok? Não se esqueça de fazer isso!

```
Listagem 5.4: Entidade "Estado"
  Arquivo: cadastroclientes/entidades/Estado.java
  package cadastroclientes.entidades;
2
  /**
3
   * Entidade Estado.
4
5
    * @author Prof. Dr. David Buzatto
6
7
  public class Estado {
10
      private int id;
      private String nome;
11
      private String sigla;
12
13
      public int getId() {
14
15
           return id;
```

```
}
16
17
       public void setId( int id ) {
18
           this.id = id;
19
20
21
       public String getNome() {
           return nome;
23
       }
24
25
       public void setNome( String nome ) {
26
           this.nome = nome;
27
28
29
       public String getSigla() {
30
           return sigla;
31
32
33
       public void setSigla( String sigla ) {
           this.sigla = sigla;
35
       }
36
37
38 }
```

```
Listagem 5.5: Entidade "Cidade"
  Arquivo: cadastroclientes/entidades/Cidade.java
package cadastroclientes.entidades;
2
  /**
   * Entidade Cidade.
5
   * @author Prof. Dr. David Buzatto
7
  public class Cidade {
      private int id;
10
      private String nome;
11
      private Estado estado;
12
13
14
      public int getId() {
```

```
return id;
15
       }
16
17
       public void setId( int id ) {
18
           this.id = id;
19
       }
20
21
       public String getNome() {
22
           return nome;
23
       }
24
25
       public void setNome( String nome ) {
26
           this.nome = nome;
27
28
29
       public Estado getEstado() {
30
           return estado;
31
32
       public void setEstado( Estado estado ) {
34
           this.estado = estado;
35
36
37
38 }
```

```
Listagem 5.6: Entidade "Cliente"
  Arquivo: cadastroclientes/entidades/Cliente.java
package cadastroclientes.entidades;
3 import java.sql.Date;
4
5 /**
  * Entidade Cliente.
6
7
   * @author Prof. Dr. David Buzatto
public class Cliente {
11
      private int id;
12
13
      private String nome;
```

```
private String sobrenome;
14
       private Date dataNascimento;
15
       private String cpf;
16
       private String email;
17
       private String logradouro;
18
       private String numero;
19
       private String bairro;
       private String cep;
21
       private Cidade cidade;
22
23
       public int getId() {
24
25
           return id;
26
27
       public void setId( int id ) {
28
           this.id = id;
29
30
31
       public String getNome() {
32
           return nome;
33
       }
34
35
       public void setNome( String nome ) {
36
37
           this.nome = nome;
       }
39
       public String getSobrenome() {
40
           return sobrenome;
41
       }
42
43
       public void setSobrenome( String sobrenome ) {
           this.sobrenome = sobrenome;
45
       }
46
47
       public Date getDataNascimento() {
48
           return dataNascimento;
       }
50
51
       public void setDataNascimento( Date dataNascimento ) {
52
           this.dataNascimento = dataNascimento;
53
       }
54
55
```

```
public String getCpf() {
56
           return cpf;
57
       }
58
59
       public void setCpf( String cpf ) {
60
           this.cpf = cpf;
61
62
63
       public String getEmail() {
64
           return email;
65
       }
66
67
       public void setEmail( String email ) {
68
           this.email = email;
69
       }
70
71
       public String getLogradouro() {
72
           return logradouro;
73
       }
75
       public void setLogradouro( String logradouro ) {
76
           this.logradouro = logradouro;
77
78
79
       public String getNumero() {
80
           return numero;
81
82
83
       public void setNumero( String numero ) {
84
           this.numero = numero;
85
       }
86
87
       public String getBairro() {
88
           return bairro;
89
       }
90
91
       public void setBairro( String bairro ) {
92
           this.bairro = bairro;
93
       }
94
95
       public String getCep() {
96
           return cep;
97
```

```
}
98
99
        public void setCep( String cep ) {
100
             this.cep = cep;
101
102
103
        public Cidade getCidade() {
104
            return cidade;
105
106
107
        public void setCidade( Cidade cidade ) {
108
109
             this.cidade = cidade;
110
111
112 }
```

Com as entidades prontas, vamos criar os DAOs. No pacote "cadastroclientes.dao", crie três classes: "EstadoDAO", "CidadeDAO" e "ClienteDAO". O código-fonte de cada uma dessas classes é apresentado nas listagens a seguir.

```
Listagem 5.7: Código da classe "EstadoDAO"
  Arquivo: cadastroclientes/dao/EstadoDAO.java
  package cadastroclientes.dao;
3 import cadastroclientes.entidades.Estado;
4 import java.sql.PreparedStatement;
5 import java.sql.ResultSet;
6 import java.sql.SQLException;
7 import java.util.ArrayList;
  import java.util.List;
  /**
10
   * DAO para a entidade Estado.
11
12
13
    * @author Prof. Dr. David Buzatto
  public class EstadoDAO extends DAO<Estado> {
16
      public EstadoDAO() throws SQLException {
17
      }
18
```

```
19
       @Override
20
       public void salvar( Estado obj ) throws SQLException {
21
22
           PreparedStatement stmt = getConnection().prepareStatement(
23
                    0.00
24
                    INSERT INTO
25
                    estado( nome, sigla )
26
                    VALUES( ?, ? );
27
                    """):
28
29
           stmt.setString( 1, obj.getNome() );
30
           stmt.setString( 2, obj.getSigla() );
31
32
           stmt.executeUpdate();
33
           stmt.close();
34
35
       }
36
37
       @Override
38
       public void atualizar( Estado obj ) throws SQLException {
39
40
           PreparedStatement stmt = getConnection().prepareStatement(
41
42
                    UPDATE estado
43
                    SET
44
                        nome = ?,
45
                        sigla = ?
46
                    WHERE
47
                        id = ?;
48
                    """);
49
50
           stmt.setString( 1, obj.getNome() );
51
           stmt.setString( 2, obj.getSigla() );
52
           stmt.setInt( 3, obj.getId() );
53
54
           stmt.executeUpdate();
55
           stmt.close();
56
57
       }
58
59
       @Override
60
```

```
public void excluir( Estado obj ) throws SQLException {
61
62
            PreparedStatement stmt = getConnection().prepareStatement(
63
64
                    DELETE FROM estado
65
                    WHERE
66
                         id = ?;
                     """);
68
69
            stmt.setInt( 1, obj.getId() );
70
71
            stmt.executeUpdate();
72
            stmt.close();
73
74
       }
75
76
        @Override
77
        public List<Estado> listarTodos() throws SQLException {
78
79
            List<Estado> lista = new ArrayList<>();
80
81
            PreparedStatement stmt = getConnection().prepareStatement(
82
83
                    SELECT * FROM estado
84
                     ORDER BY nome, sigla;
                     """);
86
87
            ResultSet rs = stmt.executeQuery();
88
89
            while ( rs.next() ) {
90
                Estado e = new Estado();
92
93
                e.setId( rs.getInt( "id" ) );
94
                e.setNome( rs.getString( "nome" ) );
95
                e.setSigla( rs.getString( "sigla" ) );
                lista.add( e );
98
99
            }
100
101
            rs.close();
102
```

```
stmt.close();
103
104
            return lista;
105
106
        }
107
108
        @Override
109
        public Estado obterPorId( int id ) throws SQLException {
110
111
            Estado estado = null;
112
113
            PreparedStatement stmt = getConnection().prepareStatement(
114
115
                     SELECT * FROM estado
116
                     WHERE id = ?;
117
                     """);
118
119
            stmt.setInt( 1, id );
120
121
            ResultSet rs = stmt.executeQuery();
122
123
            if ( rs.next() ) {
124
125
                 estado = new Estado();
126
127
                 estado.setId( rs.getInt( "id" ) );
128
                 estado.setNome( rs.getString( "nome" ) );
129
                 estado.setSigla( rs.getString( "sigla" ) );
130
131
            }
133
            rs.close();
134
135
            stmt.close();
136
            return estado;
137
138
        }
139
140
141 }
```

```
Listagem 5.8: Código da classe "CidadeoDAO"
  Arquivo: cadastroclientes/dao/CidadeDAO.java
package cadastroclientes.dao;
3 import cadastroclientes.entidades.Cidade;
4 import cadastroclientes.entidades.Estado;
5 import java.sql.PreparedStatement;
6 import java.sql.ResultSet;
7 import java.sql.SQLException;
8 import java.util.ArrayList;
9 import java.util.List;
10
  /**
11
   * DAO para a entidade Cidade.
12
13
    * @author Prof. Dr. David Buzatto
14
   */
public class CidadeDAO extends DAO<Cidade> {
17
       public CidadeDAO() throws SQLException {
18
19
       }
20
       @Override
21
       public void salvar( Cidade obj ) throws SQLException {
23
           PreparedStatement stmt = getConnection().prepareStatement(
24
25
                   INSERT INTO
                   cidade( nome, estado_id )
27
                   VALUES( ?, ? );
28
                   """);
29
30
           stmt.setString( 1, obj.getNome() );
           stmt.setInt( 2, obj.getEstado().getId() );
33
           stmt.executeUpdate();
34
           stmt.close();
35
36
       }
37
       @Override
39
```

```
public void atualizar( Cidade obj ) throws SQLException {
40
41
           PreparedStatement stmt = getConnection().prepareStatement(
42
43
                    UPDATE cidade
44
                    SET
45
                        nome = ?,
46
                        estado_id = ?
47
                    WHERE
48
                        id = ?;
49
                    """);
50
51
           stmt.setString( 1, obj.getNome() );
52
           stmt.setInt( 2, obj.getEstado().getId() );
53
           stmt.setInt( 3, obj.getId() );
54
55
           stmt.executeUpdate();
56
           stmt.close();
57
       }
59
60
       @Override
61
       public void excluir( Cidade obj ) throws SQLException {
62
63
           PreparedStatement stmt = getConnection().prepareStatement(
64
65
                    DELETE FROM cidade
66
                    WHERE
67
                        id = ?;
68
                    """);
69
70
           stmt.setInt( 1, obj.getId() );
71
72
           stmt.executeUpdate();
73
           stmt.close();
74
75
       }
76
77
       @Override
78
       public List<Cidade> listarTodos() throws SQLException {
79
80
           List<Cidade> lista = new ArrayList<>();
81
```

```
82
            PreparedStatement stmt = getConnection().prepareStatement(
83
84
                     SELECT
85
                         c.id idCidade,
86
                         c.nome nomeCidade,
87
                         e.id idEstado,
                          e.nome nomeEstado,
89
                         e.sigla siglaEstado
90
                     FROM
91
                         cidade c,
92
93
                         estado e
                     WHERE
                          c.estado_id = e.id
95
                     ORDER BY c.nome, e.nome, e.sigla;
96
                     """);
97
98
            ResultSet rs = stmt.executeQuery();
99
            while ( rs.next() ) {
101
102
                 Cidade c = new Cidade();
103
104
                 Estado e = new Estado();
105
                 c.setId( rs.getInt( "idCidade" ) );
106
                 c.setNome( rs.getString( "nomeCidade" ) );
107
                 c.setEstado( e );
108
109
                 e.setId( rs.getInt( "idEstado" ) );
110
                 e.setNome( rs.getString( "nomeEstado" ) );
111
                 e.setSigla( rs.getString( "siglaEstado" ) );
112
113
114
                 lista.add( c );
115
            }
116
117
            rs.close();
118
            stmt.close();
119
120
            return lista;
121
122
        }
123
```

```
124
        @Override
125
        public Cidade obterPorId( int id ) throws SQLException {
126
127
            Cidade cidade = null;
128
129
            PreparedStatement stmt = getConnection().prepareStatement(
130
131
                     SELECT
132
                         c.id idCidade,
133
                          c.nome nomeCidade,
134
135
                          e.id idEstado,
                          e.nome nomeEstado,
136
                          e.sigla siglaEstado
137
                     FROM
138
                          cidade c,
139
                          estado e
140
                     WHERE
141
                          c.id = ? AND
142
                          c.estado_id = e.id;
143
                     """);
144
145
            stmt.setInt( 1, id );
146
147
            ResultSet rs = stmt.executeQuery();
148
149
            if ( rs.next() ) {
150
151
                 cidade = new Cidade();
152
                 Estado e = new Estado();
153
154
                 cidade.setId( rs.getInt( "idCidade" ) );
155
                 cidade.setNome( rs.getString( "nomeCidade" ) );
156
                 cidade.setEstado( e );
157
158
                 e.setId( rs.getInt( "idEstado" ) );
159
                 e.setNome( rs.getString( "nomeEstado" ) );
160
                 e.setSigla( rs.getString( "siglaEstado" ) );
161
162
            }
163
164
            rs.close();
165
```

```
166 stmt.close();
167
168 return cidade;
169
170 }
171
172
173 }
```

```
Listagem 5.9: Código da classe "ClienteDAO"
  Arquivo: cadastroclientes/dao/ClienteDAO.java
  package cadastroclientes.dao;
3 import cadastroclientes.entidades.Cidade;
4 import cadastroclientes.entidades.Cliente;
5 import cadastroclientes.entidades.Estado;
6 import java.sql.PreparedStatement;
7 import java.sql.ResultSet;
8 import java.sql.SQLException;
9 import java.util.ArrayList;
10 import java.util.List;
11
  /**
12
   * DAO para a entidade Cliente.
13
14
   * @author Prof. Dr. David Buzatto
15
public class ClienteDAO extends DAO<Cliente> {
18
19
      public ClienteDAO() throws SQLException {
       }
20
21
      @Override
22
      public void salvar( Cliente obj ) throws SQLException {
23
24
          PreparedStatement stmt = getConnection().prepareStatement(
26
                   INSERT INTO
27
                   cliente(
28
                       nome,
29
```

```
sobrenome,
30
                        data_nascimento,
31
                        cpf,
32
                        email,
33
                        logradouro,
34
                        numero,
35
                        bairro,
                        cep,
37
                        cidade_id )
38
                    VALUES( ?, ?, ?, ?, ?, ?, ?, ?, ?);
39
                    """);
40
41
           stmt.setString( 1, obj.getNome() );
42
           stmt.setString( 2, obj.getSobrenome() );
43
           stmt.setDate( 3, obj.getDataNascimento() );
44
           stmt.setString( 4, obj.getCpf() );
45
           stmt.setString( 5, obj.getEmail() );
46
           stmt.setString( 6, obj.getLogradouro() );
47
           stmt.setString( 7, obj.getNumero() );
           stmt.setString( 8, obj.getBairro() );
49
           stmt.setString( 9, obj.getCep() );
50
           stmt.setInt( 10, obj.getCidade().getId() );
51
52
           stmt.executeUpdate();
53
           stmt.close();
54
55
       }
56
57
       @Override
58
       public void atualizar( Cliente obj ) throws SQLException {
59
60
           PreparedStatement stmt = getConnection().prepareStatement(
61
62
                    UPDATE cliente
63
                    SET
64
65
                        nome = ?,
                        sobrenome = ?,
66
                        data_nascimento = ?,
67
                        cpf = ?,
68
                        email = ?,
69
                        logradouro = ?,
70
                        numero = ?,
71
```

```
bairro = ?,
72
                         cep = ?,
73
                         cidade_id = ?
74
                     WHERE
75
                         id = ?:
76
                     """);
77
78
            stmt.setString( 1, obj.getNome() );
79
            stmt.setString( 2, obj.getSobrenome() );
80
            stmt.setDate( 3, obj.getDataNascimento() );
81
            stmt.setString( 4, obj.getCpf() );
82
            stmt.setString( 5, obj.getEmail() );
83
            stmt.setString( 6, obj.getLogradouro() );
84
            stmt.setString( 7, obj.getNumero() );
85
            stmt.setString( 8, obj.getBairro() );
86
            stmt.setString( 9, obj.getCep() );
87
            stmt.setInt( 10, obj.getCidade().getId() );
88
            stmt.setInt( 11, obj.getId() );
89
            stmt.executeUpdate();
91
            stmt.close();
92
93
        }
94
95
        @Override
96
        public void excluir( Cliente obj ) throws SQLException {
97
98
            PreparedStatement stmt = getConnection().prepareStatement(
99
100
                     DELETE FROM cliente
101
                     WHERE
102
                         id = ?;
103
104
                     """);
105
            stmt.setInt( 1, obj.getId() );
106
107
            stmt.executeUpdate();
108
            stmt.close();
109
110
        }
111
112
        @Override
113
```

```
public List<Cliente> listarTodos() throws SQLException {
114
115
            List<Cliente> lista = new ArrayList<>();
116
117
            PreparedStatement stmt = getConnection().prepareStatement(
118
                     0.00
119
                     SELECT
120
121
                         c.id idCliente,
122
                         c.nome nomeCliente,
123
                         c.sobreNome sobrenomeCliente,
                         c.data_nascimento dataNascimentoCliente,
124
125
                         c.cpf cpfCliente,
                         c.email emailCliente,
126
                         c.logradouro logradouroCliente,
127
                         c.numero numeroCliente,
128
                         c.bairro bairroCliente,
129
                         c.cep cepCliente,
130
                         ci.id idCidade,
131
                         ci.nome nomeCidade,
132
                         e.id idEstado,
133
                         e.nome nomeEstado,
134
                         e.sigla siglaEstado
135
136
                     FROM
                         cliente c,
137
                         cidade ci,
138
                         estado e
139
                     WHERE
140
                         c.cidade_id = ci.id AND
141
                         ci.estado_id = e.id
142
                     ORDER BY c.nome, c.sobreNome, ci.nome;
143
                     """);
144
145
            ResultSet rs = stmt.executeQuery();
146
147
            while ( rs.next() ) {
148
149
                 Cliente c = new Cliente();
150
                 Cidade ci = new Cidade();
151
                 Estado e = new Estado();
152
153
                 c.setId( rs.getInt( "idCliente" ) );
154
                 c.setNome( rs.getString( "nomeCliente" ) );
155
```

```
c.setSobrenome( rs.getString( "sobrenomeCliente" ) );
156
                c.setDataNascimento( rs.getDate( "dataNascimentoCliente" ) );
157
                c.setCpf( rs.getString( "cpfCliente" ) );
158
                c.setEmail( rs.getString( "emailCliente" ) );
159
                c.setLogradouro( rs.getString( "logradouroCliente" ) );
160
                c.setNumero( rs.getString( "numeroCliente" ) );
161
                c.setBairro( rs.getString( "bairroCliente" ) );
162
                c.setCep( rs.getString( "cepCliente" ) );
163
                c.setCidade( ci );
164
165
                ci.setId( rs.getInt( "idCidade" ) );
166
                ci.setNome( rs.getString( "nomeCidade" ) );
167
                ci.setEstado( e );
168
169
                e.setId( rs.getInt( "idEstado" ) );
170
                e.setNome( rs.getString( "nomeEstado" ) );
171
                e.setSigla( rs.getString( "siglaEstado" ) );
172
173
                lista.add( c );
174
175
            }
176
177
            rs.close();
178
            stmt.close();
179
180
181
            return lista;
182
        }
183
184
        @Override
185
        public Cliente obterPorId( int id ) throws SQLException {
186
187
            Cliente cliente = null;
188
189
            PreparedStatement stmt = getConnection().prepareStatement(
190
191
192
                     SELECT
                         c.id idCliente,
193
                         c.nome nomeCliente,
194
                         c.sobreNome sobrenomeCliente,
195
                         c.data_nascimento dataNascimentoCliente,
196
                         c.cpf cpfCliente,
197
```

```
c.email emailCliente,
198
                         c.logradouro logradouroCliente,
199
                         c.numero numeroCliente,
200
                         c.bairro bairroCliente,
201
                         c.cep cepCliente,
202
                         ci.id idCidade,
                         ci.nome nomeCidade,
204
205
                         e.id idEstado,
                         e.nome nomeEstado,
206
207
                         e.sigla siglaEstado
                     FROM
208
209
                         cliente c,
                         cidade ci,
210
                         estado e
211
212
                     WHERE
                         c.id = ? AND
213
                         c.cidade_id = ci.id AND
214
                         ci.estado id = e.id;
215
                     """);
216
217
            stmt.setInt( 1, id );
218
219
220
            ResultSet rs = stmt.executeQuery();
221
            if ( rs.next() ) {
222
223
                cliente = new Cliente();
224
                Cidade ci = new Cidade();
225
                Estado e = new Estado();
226
227
                cliente.setId( rs.getInt( "idCliente" ) );
228
                cliente.setNome( rs.getString( "nomeCliente" ) );
229
230
                cliente.setSobrenome( rs.getString( "sobrenomeCliente" ) );
                cliente.setDataNascimento( rs.getDate("dataNascimentoCliente") );
231
                cliente.setCpf( rs.getString( "cpfCliente" ) );
                cliente.setEmail( rs.getString( "emailCliente" ) );
233
                cliente.setLogradouro( rs.getString( "logradouroCliente" ) );
234
                cliente.setNumero( rs.getString( "numeroCliente" ) );
235
                cliente.setBairro( rs.getString( "bairroCliente" ) );
236
                cliente.setCep( rs.getString( "cepCliente" ) );
237
                cliente.setCidade( ci );
238
239
```

```
ci.setId( rs.getInt( "idCidade" ) );
240
                 ci.setNome( rs.getString( "nomeCidade" ) );
241
                 ci.setEstado( e );
242
243
                 e.setId( rs.getInt( "idEstado" ) );
244
                 e.setNome( rs.getString( "nomeEstado" ) );
245
                 e.setSigla( rs.getString( "siglaEstado" ) );
246
247
            }
248
249
            rs.close();
250
251
            stmt.close();
252
            return cliente;
253
254
        }
255
256
257
   }
```

Quantos códigos hein? Copiou tudo? Crie algumas classes de teste no pacote "cadastroclientes.teste" e teste a persistência de cada entidade. Com isso, terminamos a parte da persistência do nosso projeto.

Agora nós vamos começar a implementar as visualizações e os controladores. Nossa aplicação terá três *links* no index. jsp, sendo que cada *link* levará a um determinado cadastro. Cada cadastro vai conter uma página principal onde todos os itens desse cadastro serão exibidos e onde poderão ser alterados, excluídos ou então poderemos cadastrar um novo item.

Vamos começar pelo cadastro de estados. Na pasta "estados" dentro da pasta "formularios", crie um arquivo JSP com o nome de "listagem. jsp" (sem as aspas). Nesse arquivo vão ser listados todos os estados, portanto precisamos obter esses estados de alguma forma. Você se lembra que nos nossos DAOs existe um método chamado [listarTodos()] que retorna todos os registros de uma determinada tabela? Nós não vamos usar esse método diretamente no JSP, então vamos criar uma classe de serviços que vai instanciar o DAO, gerar a lista e fechar a conexão para nós. Nos pacotes de código-fonte, crie um novo pacote dentro do "cadastroclientes" chamado "servicos" (sem o "ç"). Nesse pacote, crie uma classe chamada "EstadoServices" (sem as aspas). Essa classe vai ter apenas um método, chamado [getTodos()], que retornará uma lista de estados. Veja o código-fonte dela na Listagem 5.10.

```
Listagem 5.10: Código-fonte da classe de serviços para Estados
  Arquivo: cadastroclientes/servicos/EstadoServices.java
package cadastroclientes.servicos;
3 import cadastroclientes.dao.EstadoDAO;
4 import cadastroclientes.entidades.Estado;
5 import java.sql.SQLException;
6 import java.util.ArrayList;
7 import java.util.List;
  /**
9
   * Classe de serviços para a entidade Estado.
10
11
    * @author Prof. Dr. David Buzatto
12
13
  public class EstadoServices {
14
15
       /**
16
        * Usa o EstadoDAO para obter todos os estados.
17
18
        * @return Lista de estados.
19
20
       public List<Estado> getTodos() {
21
22
           List<Estado> lista = new ArrayList<>();
23
           EstadoDAO dao = null;
24
25
           try {
26
               dao = new EstadoDAO();
27
               lista = dao.listarTodos();
28
           } catch ( SQLException exc ) {
29
               exc.printStackTrace();
30
           } finally {
31
               if ( dao != null ) {
32
                   try {
33
                       dao.fecharConexao();
34
                   } catch ( SQLException exc ) {
35
                       exc.printStackTrace();
36
                   }
37
               }
38
           }
39
```

```
40
41 return lista;
42
43 }
44
45 }
```

Criamos essa classe para que ela encapsule todo o processo de obtenção da lista de estados. Note que é no método getTodos() que o EstadoDAO vai ser instanciado e gerenciado.

Agora que temos o JSP que vai obter a lista de estados para nós, além de gerenciar o DAO, nós podemos implementar o nosso arquivo de listagem de estados. Abra o arquivo /formularios/estados/listagem.jsp e copie o código da Listagem 5.10.

```
Listagem 5.11: Código da listagem de Estados
  Arquivo: /formularios/estados/listagem.jsp
  <%@page contentType="text/html" pageEncoding="UTF-8"%>
2 %@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
3 <c:set var="cp" value="${pageContext.request.contextPath}"/>
4 <c:set var="prefixo" value="processaEstados?acao=preparar"/>
  <!DOCTYPE html>
7
  <html>
     <head>
8
       <title>Estados Cadastrados</title>
9
       <meta charset="UTF-8">
10
       <meta name="viewport"
11
             content="width=device-width, initial-scale=1.0">
12
       <link rel="stylesheet"</pre>
13
             href="${cp}/css/estilos.css"/>
14
     </head>
15
16
     <body>
17
18
       <h1>Estados Cadastrados</h1>
19
20
       >
21
         <a href="${cp}/formularios/estados/novo.jsp">
22
           Novo Estado
23
```

```
</a>
24
     25
26
     27
       <thead>
28
        29
          Id
30
          Nome
31
          Sigla
32
          Alterar
33
          Excluir
34
        35
       </thead>
36
       37
38
        <jsp:useBean
39
            id="servicos"
40
            scope="page"
41
            class="cadastroclientes.servicos.EstadoServices"/>
42
43
        <c:forEach items="${servicos.todos}" var="estado">
44
45
            ${estado.id}
46
            ${estado.nome}
47
            ${estado.sigla}
48
            49
             <a href="${cp}/${prefixo}Alteracao&id=${estado.id}">
50
51
               Alterar
             </a>
52
            53
            <a href="${cp}/${prefixo}Exclusao&id=${estado.id}">
55
               Excluir
56
             </a>
57
            58
          59
        </c:forEach>
60
       61
62
     63
64
     >
65
```

```
<a href="${cp}/formularios/estados/novo.jsp">
66
           Novo Estado
67
         </a>
68
       69
70
       <a href="${cp}/index.jsp">Tela Principal</a>
71
72
     </body>
73
74
   </html>
75
```

Perceba que a identação do código foi feita com dois espaços para economizar espaço nas listagens, mas você pode manter os quatro espaços usados por padrão.

Vamos analisar o código, detalhando as novidades que aparecerem. Nas linhas 3 e 4 usamos a tag <c:set> para configurar no escopo da página dois valores que usaremos no código. Um deles, chamado de cp será o caminho do contexto da aplicação, obtido através da instrução \${pageContext.request.contextPath}. Essa instrução da EL retornará no nosso caso o valor /CadastroClientes, pois é esse o contexto da aplicação configurado na criação do projeto. Faremos dessa forma para que, independente do contexto, ele seja obtido apropriadamente. Estamos fazendo isso para que possamos configurar sempre caminhos absolutos para os nossos recursos, evitando problemas de referenciamento relativo. Poderemos acessar esse valor agora usando a construção \${cp}. O mesmo acontece na linha 4, onde o valor processaEstados?acao=preparar é configurado na variável prefixo. Veremos o motivo adiante.

Entre as linhas 21 e 25, criamos um link que aponta para o arquivo /CadastroClientes/formularios/estado/novo.jsp —que ainda não implementamos— e que será o formulário responsável em criar um novo estado. Perceba que usamos a variável de página cp para obter o contexto da aplicação. Este mesmo código é repetido entre as linhas 65 e 69. Na linha 27, abrimos a tag de uma tabela e, até a linha 36, criamos seu cabeçalho. Na linha 31, usamos a tag <jsp:useBean> para instanciar um objeto do tipo EstadoServices, que contém o método que vamos usar para obter a lista de estados. Demos o nome de servicos para essa instância. Na linha 44, usamos um <c:forEach> para iterar sobre a lista retornada pelo método getTodos() da instância servicos. Note que a chamada do método getTodos() é somente todos, pois seguimos o padrão JavaBeans nas ELs como você deve se lembrar. Essa chamada é feita usando EL no atributo items. No atributo var, damos o nome da variável que vai armazenar a instância atual durante a iteração, no caso, estado. Entre as linhas 45 e 59 nós definimos o código que será gerado a cada ite-

ração do c:forEach, que corresponde à uma linha da tabela. As três primeiras colunas da tabela são fáceis de entender, entretanto a quarta e a quinta mudam um pouco. Entre as linhas 49 e 53, a coluna é formada por um link que aponta para \${cp}/\${prefixo}Alteracao&id=\$estado.id, que após ser processado gerará o caminho /CadastroClientes/processaEstados?acao=preparararAlteracao&id=AlgumId. Veja que usamos cp e prefixo, sendo que, respetivamente serão substituídas por /CadastroClientes e processaEstados?acao=prepararar. Note que estamos codificando na URL duas variáveis. A primeira, chamada "acao", vai informar para o Servlet que vai estar mapeado para /processaEstados o que queremos fazer, no caso, "prepararAlteracao". A segunda variável, chamada "id" vai conter o identificador do estado daquela linha da tabela, ou seja, queremos alterar um estado que tem um determinado id. O mesmo acontece entre as linhas 54 e 59, mudando a ação para "prepararExclusao". Sei que pode estar um pouco confuso, mas na hora que terminarmos todos os arquivos do cadastro de estados tudo isso vai ficar fácil de entender, não se preocupe.

Note que deixei a linha 13 por último. Nela usamos a tag link> para referenciar um arquivo de folhas de estilos. Nos exemplos dos Capítulos anteriores nós usamos estilos declarados dentro dos arquivos HTML e/ou JSP usando a tag <style>. A partir de agora iremos separar os estilos em arquivos e é por isso que foi usada a tag para a pontar para \${cp}/css/estilos.css. Note novamente o uso da variável cp!. Vamos criar esse arquivo? Na pasta "css" dentro de \$\overline{Web Pages}\$ -que criamos quando preparamos o projeto- clique com o botão direito sobre ele e escolha \$\overline{New}\$. Provavelmente o item que queremos não estará visível, então escolha \$\overline{Other...}\$. Escolha \$\overline{Web}\$ na categoria e em \$\overline{File Types}\$ escolha \$\overline{Cascading Style Sheet}\$. Clique em próximo e dê o nome de "estilos" ao arquivo. Copie o código da Listagem 5.12. neste arquivo.

```
Listagem 5.12: Arquivo de estilos da aplicação
  Arquivo: /css/estilos.css
  body {
      font-family: Verdana, Arial, Helvetica, sans-serif;
2
3
      font-size: 14px;
      background-color: #FFFFFF;
  }
5
6
7
  a {
8
      color: #0484AE;
      text-decoration: none;
```

```
font-weight: bold;
10
   }
11
12
  a:hover {
13
       color: #000000;
14
       text-decoration: underline;
15
  }
16
17
   .tabelaListagem {
18
       background: #000000;
19
   }
20
21
   .tabelaListagem th {
22
       background: #EEEEEE;
23
       padding: 5px;
24
   }
25
26
   .tabelaListagem td {
27
       background: #FFFFFF;
28
       padding: 5px;
29
   }
30
31
   .alinharDireita {
32
       text-align: right;
33
   }
34
```

Execute o projeto e aponte o navegador para a página de listagem para ver como está ficando. Se você já tiver alguns estados previamente cadastrados via SQL, vai ver que eles aparecerão na tabela. Vamos agora criar nosso formulário para criar estados. Na pasta /formularios/estados, crie um arquivo JSP chamado "novo" (sem as aspas). O código-fonte do arquivo pode ser visto na Listagem 5.13.

```
<head>
7
      <title>Novo Estado</title>
8
      <meta charset="UTF-8">
9
      <meta name="viewport"</pre>
10
            content="width=device-width, initial-scale=1.0">
11
      <link rel="stylesheet"</pre>
12
            href="${cp}/css/estilos.css"/>
13
    </head>
14
15
    <body>
16
17
      <h1>Novo Estado</h1>
18
19
      <form method="post" action="${cp}/processaEstados">
20
21
        <input name="acao" type="hidden" value="inserir"/>
22
23
        24
          25
            Nome:
26
            27
              <input name="nome"</pre>
28
                    type="text"
29
                    size="20"
30
                    maxlength="30"
31
                    required/>
32
            33
          34
          35
            Sigla:
36
            >
37
              <input name="sigla"</pre>
38
                    type="text"
39
                    size="3"
40
                    maxlength="2"
41
                    required/>
42
            43
          44
          45
46
47
              <a href="${cp}/formularios/estados/listagem.jsp">
                Voltar
48
```

```
</a>
49
          50
          51
           <input type="submit" value="Salvar"/>
52
53
        54
       56
     </form>
57
58
   </body>
59
60
  </html>
```

Salve o arquivo e veja se ele está sendo exibido corretamente no navegador. Acesse-o pelo link "Novo Estado" da página de listagem de estados. Vamos analisar o código. Na linha 12 referenciamos o nosso arquivo de estilos. Na linha 20 declaramos a tag do formulário. Note que a action está apontando para \${cp}/processaEstados que será a URL que mapearemos o Servlet que tratará os estados. A novidade nesse formulário é o uso de um *input* do tipo hidden (escondido). Esses tipos de *input* são usados para guardar valores que o usuário do sistema não tem acesso diretamente. No nosso caso, esse input, que tem o nome de acao e valor inserir, vai indicar ao Servlet que queremos criar um novo estado. Nós precisamos tratar isso na implementação do nosso Servlet ok? Além disso, perceba que utilizamos alguns atributos nos inputs para que nosso formulário seja validado antes de ser submetido. Por exemplo, o input do atributo nome do estado tem tamanho máximo de 30 caracteres (maxlength) e é obrigatório (required). Como já temos nossa página de listagem e o nosso formulário de criação de estados, está na hora de criarmos o Servlet que vai gerenciar isso. No pacote cadastroclientes.controladores, crie um Servlet com o nome de "EstadosServlet" (sem as aspas) e configure o mapeamento dele para "/processaEstados" (sem as aspas).

A seguir, na Listagem 5.14 é apresentado o código completo da classe EstadosServlet.

```
Listagem 5.14: Código-fonte do Servlet "EstadosServlet"
Arquivo: cadastroclientes/controladores/EstadosServlet.java

package cadastroclientes.controladores;

import cadastroclientes.dao.EstadoDAO;
import cadastroclientes.entidades.Estado;
```

```
5 import java.io.IOException;
6 import java.sql.SQLException;
7 import jakarta.servlet.RequestDispatcher;
8 import jakarta.servlet.ServletException;
9 import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
13
14
   /**
   * Servlet para tratar Estados.
15
16
   * @author Prof. Dr. David Buzatto
17
18
   @WebServlet( name = "EstadosServlet",
19
                urlPatterns = { "/processaEstados" } )
20
   public class EstadosServlet extends HttpServlet {
21
22
       protected void processRequest(
23
               HttpServletRequest request,
24
               HttpServletResponse response )
25
               throws ServletException, IOException {
26
27
           String acao = request.getParameter( "acao" );
28
           EstadoDAO dao = null;
29
           RequestDispatcher disp = null;
30
31
           try {
32
33
               dao = new EstadoDAO();
34
35
               if ( acao.equals( "inserir" ) ) {
36
37
                   String nome = request.getParameter( "nome" );
38
                   String sigla = request.getParameter( "sigla" );
39
40
                   Estado e = new Estado();
41
                   e.setNome( nome );
42
                   e.setSigla( sigla );
43
44
                   dao.salvar( e );
45
46
```

```
disp = request.getRequestDispatcher(
47
                            "/formularios/estados/listagem.jsp" );
48
49
               } else if ( acao.equals( "alterar" ) ) {
50
51
                    int id = Integer.parseInt(request.getParameter( "id" ));
52
                    String nome = request.getParameter( "nome" );
                   String sigla = request.getParameter( "sigla" );
54
55
                   Estado e = new Estado();
56
                    e.setId( id );
57
58
                    e.setNome( nome );
                    e.setSigla( sigla );
60
                   dao.atualizar( e );
61
62
                    disp = request.getRequestDispatcher(
63
                            "/formularios/estados/listagem.jsp" );
64
               } else if ( acao.equals( "excluir" ) ) {
66
67
                    int id = Integer.parseInt(request.getParameter( "id" ));
68
69
                   Estado e = new Estado();
70
                    e.setId( id );
71
72
                   dao.excluir( e );
73
74
                    disp = request.getRequestDispatcher(
75
                            "/formularios/estados/listagem.jsp");
76
77
               } else {
78
79
                    int id = Integer.parseInt(request.getParameter( "id" ));
80
                   Estado e = dao.obterPorId( id );
81
                    request.setAttribute( "estado", e );
                    if ( acao.equals( "prepararAlteracao" ) ) {
84
                        disp = request.getRequestDispatcher(
85
                                "/formularios/estados/alterar.jsp" );
86
                    } else if ( acao.equals( "prepararExclusao" ) ) {
87
                        disp = request.getRequestDispatcher(
88
```

```
"/formularios/estados/excluir.jsp" );
89
                     }
90
91
                 }
92
93
            } catch ( SQLException exc ) {
94
                 exc.printStackTrace();
95
            } finally {
96
                 if ( dao != null ) {
97
                     try {
98
                          dao.fecharConexao();
99
                     } catch ( SQLException exc ) {
100
                          exc.printStackTrace();
101
                     }
102
                 }
103
            }
104
105
            if ( disp != null ) {
106
                 disp.forward( request, response );
107
            }
108
109
        }
110
111
        @Override
112
        protected void doGet(
113
                 HttpServletRequest request,
114
                 HttpServletResponse response )
115
116
                 throws ServletException, IOException {
            processRequest( request, response );
117
        }
118
119
        @Override
120
121
        protected void doPost(
                 HttpServletRequest request,
122
                 HttpServletResponse response )
123
                 throws ServletException, IOException {
124
            processRequest( request, response );
125
        }
126
127
        @Override
128
        public String getServletInfo() {
129
            return "EstadosServlet";
130
```

Perceba que a linha que contém o código para a configuração do encoding do request que estávamos usando até agora foi removida pois, para não precisarmos ter essa configuração em cada Servlet, criaremos um Filtro para realizar essa ação padrão para nós. Os filtros das aplicações Web em Java são componentes que podem atuar antes e/ou depois de requisições à outros componentes da aplicação, inclusive outros filtros. Para primeiramente "filtros" O pacote dentro "controladores". No pacote "filtros', crie uma nova classe chamada "ConfigurarEncodingFilter.java" e copie o código apresentado na Listagem 5.15. Note que poderíamos ter criado o filtro usando a opção apropriada nos tipos de arquivo disponíveis no NetBeans, entretanto, nosso filtro é bastante simples e todo o código inserido pelo NetBeans quando criamos usando o template é praticamente desnecessário para a nossa necessidade, sendo assim, faremos manualmente.

```
Listagem 5.15: Filtro de configuração de encoding padrão
   Arquivo: cadastroclientes/controladores/filtros/ConfigurarEncoding-
  Filter.java
  package cadastroclientes.controladores.filtros;
3 import java.io.IOException;
4 import jakarta.servlet.Filter;
5 import jakarta.servlet.FilterChain;
6 import jakarta.servlet.ServletException;
  import jakarta.servlet.ServletRequest;
8 import jakarta.servlet.ServletResponse;
  import jakarta.servlet.annotation.WebFilter;
10
  /**
11
   * Filtro para configurar o encoding das requisições de todos
12
   * os recursos da aplicação para UTF-8.
13
14
   * @author Prof. Dr. David Buzatto
15
   */
16
  @WebFilter( filterName = "ConfigurarEncodingFilter",
17
              urlPatterns = { "/*" } )
18
  public class ConfigurarEncodingFilter implements Filter {
```

```
20
       @Override
21
       public void doFilter(
22
                ServletRequest request,
23
                ServletResponse response,
24
                FilterChain chain )
25
                throws IOException, ServletException {
27
           request.setCharacterEncoding( "UTF-8" );
28
           chain.doFilter( request, response );
29
30
       }
31
32
33 }
```

Veja que na linha 17 é usada anotação @WebFilter que será responsável em indicar ao servidor de aplicações que essa classe é um Filtro. O nome do filtro, indicado pelo atributo filterName, é usado para indentificar o filtro, enquanto o atributo urlPatterns indica em quais URLs esse filtro será aplicado. No nosso caso, queremos que todas as requisições da aplicação passem por ele, então usamos o padrão /*, onde o asterisco denota "tudo". O método doFilter() realiza a atividade de filtragem, sendo que será executado antes das requisições. Queremos sempre definir a codificação para UTF-8 para padronizarmos o *encoding* que a aplicação utiliza, evitando problemas com caracteres acentuados e símbolos especiais. Isso é feito na linha 28. Na linha 29, dá-se a chance de outros filtros que porventura tenham sido criados atuarem na requisição.

Voltando ao EstadosServlet apresentado na Listagem 5.14, copie todo o código e execute o projeto. Acesse a listagem de estados e clique para criar um novo estado. Preencha o formulário e salve. O novo estado será salvo e a listagem aparecerá novamente. Teste a inserção de estados com nomes que contém palavras acentuadas, por exemplo, "São Paulo" e verifique se os caracteres acentuados estão sendo persistidos apropriadamente. Como exercício, tente entender o que está acontecendo no Servlet. Todo o código apresentado já foi estudado nos exemplos anteriores. Perceba que foram tratadas todas as ações possíveis, mas ainda faltam dois arquivos JSP para implementarmos. O alterar. jsp e o excluir. jsp. Vamos fazer isso? Crie um arquivo JSP na pasta /formularios/estados com o nome de "alterar" (sem as aspas) e copie o código da Listagem 5.16.

>

Sigla:

<input name="sigla"</pre>

34

35

36

37

38

39

Listagem 5.16: Formulário de alteração de Estados cadastrados Arquivo: /formularios/estados/alterar.jsp %@page contentType="text/html" pageEncoding="UTF-8"%> %@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %> 3 <c:set var="cp" value="\${pageContext.request.contextPath}"/> 4 <!DOCTYPE html> <html> 6 <head> 7 8 <title>Alterar Estado</title> <meta charset="UTF-8"> 9 <meta name="viewport" 10 content="width=device-width, initial-scale=1.0"> 11 <link rel="stylesheet"</pre> 12 href="\${cp}/css/estilos.css"/> 13 14 </head> 15 <body> 16 17 <h1>Alterar Estado</h1> 18 19 <form method="post" action="\${cp}/processaEstados"> 20 <input name="acao" type="hidden" value="alterar"/> 22 <input name="id" type="hidden" value="\${requestScope.estado.id}"/> 23 24 25 Nome: 28 <input name="nome"</pre> 29 type="text" 30 size="20" maxlength="30" value="\${requestScope.estado.nome}"/> 33

```
type="text"
40
                   size="3"
41
                   maxlength="2"
42
                   value="${requestScope.estado.sigla}"/>
43
           44
         45
         46
           >
47
             <a href="${cp}/formularios/estados/listagem.jsp">
48
               Voltar
49
             </a>
50
51
           52
             <input type="submit" value="Alterar"/>
53
54
         55
        56
57
      </form>
59
    </body>
60
61
  </html>
62
```

As diferenças importantes em relação ao arquivo novo. jsp são poucas. O que foi alterado é o *input* hidden nomeado como acao, que agora tem o valor alterar. Foi criado outro *input* hidden para armazenar o id do estado que será alterado. Note que os valores dos campos são preenchidos usando o atributo "estado" que foi configurado no request dentro do Servlet, na seção do código que trata a ação prepararAlteracao.

O arquivo excluir. jsp é bem parecido, só que neste arquivo não precisamos ter campos de entrada para nome e sigla, pois não vamos alterar esses dados. O importante é o id, que novamente vai ser configurado em um campo escondido. Veja na Listagem 5.17 o código do arquivo /formularios/estados/excluir. jsp que você deve criar.

Listagem 5.17: Formulário de exclusão de Estados cadastrados Arquivo: /formularios/estados/excluir.jsp

```
%@page contentType="text/html" pageEncoding="UTF-8"%>
  %@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
3 <c:set var="cp" value="${pageContext.request.contextPath}"/>
4 <!DOCTYPE html>
  <html>
6
    <head>
7
8
      <title>Excluir Estado</title>
      <meta charset="UTF-8">
9
      <meta name="viewport"</pre>
10
           content="width=device-width, initial-scale=1.0">
11
      <link rel="stylesheet"</pre>
12
           href="${cp}/css/estilos.css"/>
13
14
    </head>
15
    <body>
16
17
      <h1>Excluir Estado</h1>
18
19
      <form method="post" action="${cp}/processaEstados">
20
        <input name="acao" type="hidden" value="excluir"/>
22
        <input name="id" type="hidden" value="${requestScope.estado.id}"/>
23
24
25
        Nome:
           ${requestScope.estado.nome}
28
          29
          30
            Sigla:
           ${requestScope.estado.sigla}
          33
          34
            >
35
              <a href="${cp}/formularios/estados/listagem.jsp">
36
37
             </a>
38
            39
```

```
40
           <input type="submit" value="Excluir"/>
41
         42
        43
      44
45
     </form>
46
47
   </body>
48
49
  </html>
50
```

Copiou tudo? Teste! Verifique se tudo está funcionando corretamente. Antes de partirmos para os outros cadastros, vamos alterar o index. jsp criando três links, um para cada listagem dos cadastros. O novo código do index. jsp pode ser visto na Listagem 5.18.

```
Listagem 5.18: Código-fonte do "index.jsp"
   Arquivo: /index.jsp
  %@page contentType="text/html" pageEncoding="UTF-8"%>
2 %@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
3 <c:set var="cp" value="${pageContext.request.contextPath}"/>
4 <!DOCTYPE html>
5
   <html>
6
     <head>
7
       <title>Sistema para Cadastro de Clientes</title>
8
       <meta charset="UTF-8">
9
       <meta name="viewport"</pre>
10
             content="width=device-width, initial-scale=1.0">
11
12
       <link rel="stylesheet"</pre>
             href="${cp}/css/estilos.css"/>
13
     </head>
14
15
16
     <body>
17
       <h1>Sistema para Cadastro de Clientes</h1>
18
19
       >
20
         <a href="${cp}/formularios/clientes/listagem.jsp">Clientes</a>
21
```

```
22
       >
23
         <a href="${cp}/formularios/cidades/listagem.jsp">Cidades</a>
24
       25
26
       <q>>
         <a href="${cp}/formularios/estados/listagem.jsp">Estados</a>
27
       29
     </body>
30
31
  </html>
32
```

Teste o index.jsp e veja que agora ele tem três links para cada um dos cadastros. O que falta agora para terminarmos nosso projeto é implementar todos os JSPs dos outros cadastros, bem como seus respectivos Servlets e classes de serviço. A seguir, vou listar cada um desses arquivos, agrupando-os por tipo de cadastro, sendo que só irei comentar as linhas que contenham código que ainda não utilizamos ou que precisem de alguma explicação. Não se esqueça de testar o projeto sempre que tiver implementado os arquivos necessários para o funcionamento de uma determinada funcionalidade. No cabeçalho de cada listagem é indicado o arquivo em que ela deve estar, sendo assim lembre-se de criá-lo antes! Vamos começar?

```
Listagem 5.19: Código-fonte da classe de serviços para Cidades
   Arquivo: cadastroclientes/servicos/CidadeServices.java
  package cadastroclientes.servicos;
  import cadastroclientes.dao.CidadeDAO;
  import cadastroclientes.entidades.Cidade;
5 import java.sql.SQLException;
  import java.util.ArrayList;
  import java.util.List;
8
9
   * Classe de serviços para a entidade Cidade.
10
11
   * @author Prof. Dr. David Buzatto
12
   */
13
  public class CidadeServices {
14
15
       /**
16
```

```
* Usa o CidadeDAO para obter todos os Cidades.
17
18
        * @return Lista de Cidades.
19
20
       public List<Cidade> getTodos() {
21
22
           List<Cidade> lista = new ArrayList<>();
23
           CidadeDAO dao = null;
24
25
           try {
26
                dao = new CidadeDAO();
27
                lista = dao.listarTodos();
28
           } catch ( SQLException exc ) {
29
                exc.printStackTrace();
30
           } finally {
31
                if ( dao != null ) {
32
                    try {
33
                         dao.fecharConexao();
34
                    } catch ( SQLException exc ) {
35
                         exc.printStackTrace();
36
                    }
37
                }
38
           }
39
40
           return lista;
41
42
       }
43
44
45 }
```

```
<title>Cidades Cadastradas</title>
9
      <meta charset="UTF-8">
10
      <meta name="viewport"</pre>
11
           content="width=device-width, initial-scale=1.0">
12
      <link rel="stylesheet"</pre>
13
           href="${cp}/css/estilos.css"/>
14
    </head>
15
16
    <body>
17
18
      <h1>Cidades Cadastradas</h1>
19
20
      >
21
        <a href="${cp}/formularios/cidades/novo.jsp">
22
         Nova Cidade
23
        </a>
24
      25
26
      27
        <thead>
28
          29
           Id
30
           Nome
31
           Estado
32
           Alterar
           Excluir
34
          35
        </thead>
36
        37
          <jsp:useBean
             id="servicos"
40
             scope="page"
41
             class="cadastroclientes.servicos.CidadeServices"/>
42
43
          <c:forEach items="${servicos.todos}" var="cidade">
           ${cidade.id}
46
             ${cidade.nome}
47
             ${cidade.estado.sigla}
48
             49
               <a href="${cp}/${prefixo}Alteracao&id=${cidade.id}">
50
```

```
Alterar
51
               </a>
52
             53
             54
               <a href="${cp}/${prefixo}Exclusao&id=${cidade.id}">
55
                 Excluir
56
               </a>
57
             58
            59
          </c:forEach>
60
        61
      62
63
      >
        <a href="${cp}/formularios/cidades/novo.jsp">
65
          Nova Cidade
66
        </a>
67
      68
      <a href="${cp}/index.jsp">Tela Principal</a>
70
71
    </body>
72
73
74 </html>
```

```
Listagem 5.21: Formulário de cadastro de novas Cidades
  Arquivo: /formularios/cidades/novo.jsp
1 < @page contentType="text/html" pageEncoding="UTF-8"%>
2 %@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
3 <c:set var="cp" value="${pageContext.request.contextPath}"/>
4 <!DOCTYPE html>
5
6 <html>
    <head>
7
      <title>Nova Cidade</title>
8
      <meta charset="UTF-8">
9
      <meta name="viewport"</pre>
10
            content="width=device-width, initial-scale=1.0">
11
      <link rel="stylesheet"</pre>
12
13
             href="${cp}/css/estilos.css"/>
```

```
</head>
14
15
    <body>
16
17
      <h1>Nova Cidade</h1>
18
19
      <form method="post" action="${cp}/processaCidades">
20
21
        <input name="acao" type="hidden" value="inserir"/>
22
23
        24
25
          Nome:
           >
27
             <input name="nome"</pre>
28
                    type="text"
29
                    size="20"
30
                    maxlength="30"
31
                    required/>
32
           33
          34
35
           Estado:
36
           >
37
             <jsp:useBean
                 id="servicos"
40
                 scope="page"
41
                 class="cadastroclientes.servicos.EstadoServices"/>
42
43
             <select name="idEstado" required>
               <c:forEach items="${servicos.todos}" var="estado">
45
                 <option value="${estado.id}">
46
                   ${estado.nome} - ${estado.sigla}
47
                 </option>
48
               </c:forEach>
             </select>
51
           52
          53
          54
           >
55
```

```
<a href="${cp}/formularios/cidades/listagem.jsp">Voltar</a>
56
           57
           58
            <input type="submit" value="Salvar"/>
59
60
         61
       62
63
     </form>
64
65
    </body>
66
67
  </html>
68
```

Note que na linha 45 da Listagem 5.21 nós usamos o serviço [getTodos()] da classe EstadoServices para obter todos os estados cadastrados e com isso gerar as opções (tag <option>) do select (combo box). Note que o valor de cada option é o id associado a determinado estado, enquanto o que aparece ao usuário é a concatenação do nome e da sigla do mesmo estado. O id do estado selecionado será enviado ao Servlet por meio do parâmetro idEstado, configurada no atributo name da tag <select>).

```
Listagem 5.22: Código-fonte do Servlet "CidadesServlet"
  Arquivo: cadastroclientes/controladores/CidadesServlet.java
  package cadastroclientes.controladores;
2
3 import cadastroclientes.dao.CidadeDAO;
  import cadastroclientes.entidades.Cidade;
5 import cadastroclientes.entidades.Estado;
6 import java.io.IOException;
7 import java.sql.SQLException;
8 import jakarta.servlet.RequestDispatcher;
  import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
  import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
14
15
   * Servlet para tratar Cidades.
```

```
17
    * @author Prof. Dr. David Buzatto
18
19
   @WebServlet( name = "CidadesServlet",
20
                urlPatterns = { "/processaCidades" } )
21
   public class CidadesServlet extends HttpServlet {
22
23
       protected void processRequest(
24
               HttpServletRequest request,
25
               HttpServletResponse response )
26
               throws ServletException, IOException {
27
28
           String acao = request.getParameter( "acao" );
           CidadeDAO dao = null;
           RequestDispatcher disp = null;
31
32
           try {
33
34
               dao = new CidadeDAO();
36
               if ( acao.equals( "inserir" ) ) {
37
38
                    String nome = request.getParameter( "nome" );
39
                    int idEstado = Integer.parseInt(
40
                            request.getParameter( "idEstado" ) );
41
42
                    Estado e = new Estado();
43
                    e.setId( idEstado );
44
45
                    Cidade c = new Cidade();
46
                    c.setNome( nome );
47
                    c.setEstado( e );
48
49
                    dao.salvar( c );
50
51
                    disp = request.getRequestDispatcher(
                            "/formularios/cidades/listagem.jsp" );
54
               } else if ( acao.equals( "alterar" ) ) {
55
56
                    int id = Integer.parseInt(request.getParameter( "id" ));
57
                    String nome = request.getParameter( "nome" );
58
```

```
int idEstado = Integer.parseInt(
59
                             request.getParameter( "idEstado" ) );
60
61
                    Estado e = new Estado();
62
                    e.setId( idEstado );
63
64
                    Cidade c = new Cidade();
65
                    c.setId( id );
66
                    c.setNome( nome );
67
                    c.setEstado( e );
68
69
                    dao.atualizar( c );
70
71
                    disp = request.getRequestDispatcher(
72
                             "/formularios/cidades/listagem.jsp" );
73
74
                } else if ( acao.equals( "excluir" ) ) {
75
76
                    int id = Integer.parseInt(request.getParameter( "id" ));
77
78
                    Cidade c = new Cidade();
79
                    c.setId( id );
80
81
                    dao.excluir( c );
82
83
                    disp = request.getRequestDispatcher(
84
                             "/formularios/cidades/listagem.jsp" );
85
86
                } else {
87
88
                    int id = Integer.parseInt(request.getParameter( "id" ));
89
                    Cidade c = dao.obterPorId( id );
90
                    request.setAttribute( "cidade", c );
91
92
                    if ( acao.equals( "prepararAlteracao" ) ) {
93
                        disp = request.getRequestDispatcher(
                                 "/formularios/cidades/alterar.jsp" );
95
                    } else if ( acao.equals( "prepararExclusao" ) ) {
96
                        disp = request.getRequestDispatcher(
97
                                 "/formularios/cidades/excluir.jsp");
98
                    }
99
100
```

```
}
101
102
            } catch ( SQLException exc ) {
103
                 exc.printStackTrace();
104
            } finally {
105
                 if ( dao != null ) {
106
                     try {
107
                          dao.fecharConexao();
108
                     } catch ( SQLException exc ) {
109
110
                          exc.printStackTrace();
111
                 }
112
            }
113
114
             if ( disp != null ) {
115
                 disp.forward( request, response );
116
            }
117
118
        }
119
120
        @Override
121
        protected void doGet(
122
123
                 HttpServletRequest request,
                 HttpServletResponse response )
124
                 throws ServletException, IOException {
125
            processRequest( request, response );
126
        }
127
128
        @Override
129
        protected void doPost(
130
                 HttpServletRequest request,
131
                 HttpServletResponse response )
132
133
                 throws ServletException, IOException {
            processRequest( request, response );
134
        }
135
136
        @Override
137
        public String getServletInfo() {
138
            return "CidadesServlet";
139
        }
140
141
   }
142
```

```
Listagem 5.23: Formulário de alteração de Cidades cadastradas
  Arquivo: /formularios/cidades/alterar.jsp
  <%@page contentType="text/html" pageEncoding="UTF-8"%>
2 < 0 cm/jsp/jstl/core %>
3 <c:set var="cp" value="${pageContext.request.contextPath}"/>
4 <!DOCTYPE html>
5
  <html>
6
    <head>
7
      <title>Alterar Cidade</title>
8
      <meta charset="UTF-8">
9
      <meta name="viewport"</pre>
10
           content="width=device-width, initial-scale=1.0">
11
      <link rel="stylesheet"</pre>
12
           href="${cp}/css/estilos.css"/>
13
14
    </head>
15
    <body>
16
17
      <h1>Alterar Cidade</h1>
18
19
      <form method="post" action="${cp}/processaCidades">
20
21
        <input name="acao" type="hidden" value="alterar"/>
22
        <input name="id" type="hidden" value="${requestScope.cidade.id}"/>
23
24
25
        26
            Nome:
27
            28
             <input name="nome"</pre>
29
                    type="text"
30
                    size="20"
31
                    maxlength="30"
32
                    required
33
                    value="${requestScope.cidade.nome}"/>
34
            35
          36
          37
            Estado:
38
            39
```

```
40
              <jsp:useBean
41
                  id="servicos"
42
                  scope="page"
43
                  class="cadastroclientes.servicos.EstadoServices"/>
44
45
              <select name="idEstado" required>
                <c:forEach items="${servicos.todos}" var="estado">
47
                  <c:choose>
48
                    <c:when test="${requestScope.cidade.estado.id eq
49

    estado.id}">

                      <option value="${estado.id}" selected>
50
                        ${estado.nome} - ${estado.sigla}
                      </option>
                    </c:when>
53
                    <c:otherwise>
54
                      <option value="${estado.id}">
55
                        ${estado.nome} - ${estado.sigla}
56
                      </option>
                    </c:otherwise>
58
                  </c:choose>
59
                </c:forEach>
60
              </select>
61
62
            65
            66
              <a href="${cp}/formularios/cidades/listagem.jsp">Voltar</a>
67
            <input type="submit" value="Alterar"/>
70
71
          72
        73
74
      </form>
76
    </body>
77
78
  </html>
79
```

Na Listagem 5.23 temos algo muito interessante. Note que construímos nosso select da mesma forma que fizemos na Listagem 5.21, entretanto precisamos saber qual é o estado que é usado na cidade que será alterada. Para isso, ao usarmos o <c:forEach>, nós comparamos o id de cada estado do cadastro com o id do estado associado à cidade que vai ser alterada. Caso os ids sejam iguais, isso quer dizer que é o estado relacionado à cidade, sendo assim, a tag <option> é gerada com um atributo a mais, o selected, que não possui valor. Usando esse atributo, nós dizemos ao navegador que aquele item deve estar selecionado por padrão.

```
Listagem 5.24: Formulário de exclusão de Cidades cadastradas
  Arquivo: /formularios/cidades/excluir.jsp
  <%Cpage contentType="text/html" pageEncoding="UTF-8"%>
2 %@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
  <c:set var="cp" value="${pageContext.request.contextPath}"/>
  <!DOCTYPE html>
5
  <html>
6
7
    <head>
      <title>Excluir Cidade</title>
8
      <meta charset="UTF-8">
9
10
      <meta name="viewport"</pre>
            content="width=device-width, initial-scale=1.0">
11
      <link rel="stylesheet"</pre>
12
            href="${cp}/css/estilos.css"/>
13
    </head>
14
15
    <body>
16
17
      <h1>Excluir Cidade</h1>
18
19
      <form method="post" action="${cp}/processaCidades">
20
21
        <input name="acao" type="hidden" value="excluir"/>
22
        <input name="id" type="hidden" value="${requestScope.cidade.id}"/>
23
24
        25
26
            Nome:
27
            ${requestScope.cidade.nome}
28
          29
```

```
30
         Estado:
31
         ${requestScope.cidade.estado.nome} -
32

    $\requestScope.cidade.estado.sigla}

        33
        <a href="${cp}/formularios/cidades/listagem.jsp">Voltar</a>
         37
         38
           <input type="submit" value="Excluir"/>
39
40
         41
      43
     </form>
44
45
46
   </body>
  </html>
48
```

```
Listagem 5.25: Código-fonte da classe de serviços para Clientes
   Arquivo: cadastroclientes/servicos/ClienteServices.java
package cadastroclientes.servicos;
3 import cadastroclientes.dao.ClienteDAO;
4 import cadastroclientes.entidades.Cliente;
5 import java.sql.SQLException;
6 import java.util.ArrayList;
  import java.util.List;
  /**
   * Classe de serviços para a entidade Cliente.
10
11
    * @author Prof. Dr. David Buzatto
12
  public class ClienteServices {
15
16
       /**
17
        * Usa o ClienteDAO para obter todos os Clientes.
```

```
18
         * @return Lista de Clientes.
19
20
       public List<Cliente> getTodos() {
21
22
           List<Cliente> lista = new ArrayList<>();
23
           ClienteDAO dao = null;
24
25
           try {
26
                dao = new ClienteDAO();
27
                lista = dao.listarTodos();
28
           } catch ( SQLException exc ) {
29
                exc.printStackTrace();
30
           } finally {
31
                if ( dao != null ) {
32
                    try {
33
                         dao.fecharConexao();
34
                    } catch ( SQLException exc ) {
35
                         exc.printStackTrace();
                    }
37
                }
38
           }
39
40
           return lista;
41
42
       }
43
44
45 }
```

```
<meta charset="UTF-8">
10
      <meta name="viewport"</pre>
11
           content="width=device-width, initial-scale=1.0">
12
      <link rel="stylesheet"</pre>
13
           href="${cp}/css/estilos.css"/>
14
    </head>
15
16
    <body>
17
18
      <h1>Clientes Cadastrados</h1>
19
20
21
        <a href="${cp}/formularios/clientes/novo.jsp">
22
         Novo Cliente
23
        </a>
24
      25
26
      27
       <thead>
28
         29
           Id
30
           Nome
31
           Sobrenome
32
           E-mail
33
           CPF
           Cidade
35
           Alterar
36
           Excluir
37
         38
       </thead>
        41
         <jsp:useBean
42
             id="servicos"
43
             scope="page"
44
             class="cadastroclientes.servicos.ClienteServices"/>
         <c:forEach items="${servicos.todos}" var="cliente">
47
           48
             ${cliente.id}
49
             ${cliente.nome}
50
             ${cliente.sobrenome}
51
```

```
${cliente.email}
52
             ${cliente.cpf}
53
             ${cliente.cidade.nome}
54
55
               <a href="${cp}/${prefixo}Alteracao&id=${cliente.id}">
56
                 Alterar
57
               </a>
             59
             60
               <a href="${cp}/${prefixo}Exclusao&id=${cliente.id}">
61
                 Excluir
62
               </a>
63
             64
           65
          </c:forEach>
66
        67
68
      69
70
      >
71
        <a href="${cp}/formularios/clientes/novo.jsp">
72
         Novo Cliente
73
        </a>
74
      75
76
      <a href="${cp}/index.jsp">Tela Principal</a>
77
78
    </body>
79
80
81 </html>
```

```
<title>Novo Cliente</title>
8
      <meta charset="UTF-8">
9
      <meta name="viewport"</pre>
10
           content="width=device-width, initial-scale=1.0">
11
      <link rel="stylesheet"</pre>
12
           href="${cp}/css/estilos.css"/>
13
    </head>
14
15
    <body>
16
17
      <h1>Novo Cliente</h1>
18
19
      <form method="post" action="${cp}/processaClientes">
20
21
        <input name="acao" type="hidden" value="inserir"/>
22
23
        24
25
          Nome:
           27
             <input name="nome"</pre>
28
                    type="text"
29
                    size="20"
30
                    maxlength="45"
31
                    required/>
           33
          34
          35
           Sobrenome:
36
           37
             <input name="sobrenome"</pre>
                    type="text"
39
                    size="20"
40
                    maxlength="45"
41
                    required/>
42
           44
          45
           Data de Nascimento:
46
47
             <input name="dataNascimento"</pre>
48
                    type="date"
49
```

```
size="8"
50
                 placeholder="dd/mm/yyyy"
51
                 required/>
52
          53
        54
         55
          CPF:
56
          >
57
            <input name="cpf"</pre>
58
                 type="text"
59
                 size="13"
60
                 pattern="\d{3}.\d{3}.\d{3}-\d{2}"
61
                 placeholder="999.999.999-99"
62
                 required/>
63
          64
        65
         66
          E-mail:
67
          68
            <input name="email"</pre>
69
                 type="email"
70
                 size="20"
71
                 maxlength="60"
72
                 required/>
73
          74
        75
        76
          Logradouro:
77
          78
            <input name="logradouro"</pre>
79
                 type="text"
80
                 size="25"
81
                 maxlength="50"
82
                 required/>
83
          84
        85
         86
          Número:
87
          88
            <input name="numero"</pre>
89
90
                 type="text"
                 size="6"
91
```

```
maxlength="6"
92
                    required/>
93
            94
          95
          96
            Bairro:
97
            >
              <input name="bairro"</pre>
99
                    type="text"
100
                    size="15"
101
                    maxlength="30"
102
103
                    required/>
            104
          105
          106
            CEP:
107
108
              <input name="cep"</pre>
109
                    type="text"
110
111
                    size="7"
                    pattern="\d{5}-\d{3}"
112
                    placeholder="99999-999"
113
114
                    required/>
            115
          116
117
          Cidade:
118
119
            >
120
              <jsp:useBean
121
                 id="servicos"
122
                 scope="page"
123
124
                 class="cadastroclientes.servicos.CidadeServices"/>
125
              <select name="idCidade" required>
126
                <c:forEach items="${servicos.todos}" var="cidade">
127
                  <option value="${cidade.id}">
128
                   ${cidade.nome}
129
                 </option>
130
                </c:forEach>
131
132
              </select>
133
```

```
134
          135
          136
           137
             <a href="${cp}/formularios/clientes/listagem.jsp">
138
               Voltar
139
             </a>
140
           141
           142
             <input type="submit" value="Salvar"/>
143
           144
145
         146
147
      </form>
148
149
    </body>
150
151
  </html>
152
```

Na Listagem 5.27 usamos diversos atributos para implementar a validação do formulário do cliente que será cadastrado. No *input* do tipo date (data), apresentado na linha 48, usamos o atributo placeholder para indicar o formato esperado para a data, que no caso é "dd/mm/yyyy", ou seja, dia com dois dígitos (dd), mês com dois dígitos (mm) e ano com quatro dígitos (yyyy), todos separados por barra. Esse tipo de *input* normaliza o formato da data que será enviado e que receberá como valor. Esse formato é sempre "yyyy-mm-dd". No *input* do CPF usamos o atributo placeholder para indicar ao usuário o formato esperado e o atributo pattern para validar esse formato. O valor do atributo pattern é uma expressão regular que no nosso caso é "\d{3}.\d{3}.\d{3}-\d{2}". Cada \d indica que ali é esperado um dígito e o valor entre chaves é a quantidade esperada de símbolos. Sendo assim, para o CPF temos três digitos, um ponto, mais três dígitos e um ponto, mais três dígitos, um traço/hífen e mais dois dígitos.

```
Listagem 5.28: Código-fonte do Servlet "ClientesServlet"
Arquivo: cadastroclientes/controladores/ClientesServlet.java

package cadastroclientes.controladores;

import cadastroclientes.dao.ClienteDAO;
```

```
import cadastroclientes.entidades.Cidade;
5 import cadastroclientes.entidades.Cliente;
6 import java.io.IOException;
7 import java.sql.Date;
8 import java.sql.SQLException;
  import java.time.LocalDate;
import java.time.format.DateTimeFormatter;
import jakarta.servlet.RequestDispatcher;
import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
17
  /**
18
19
   * Servlet para tratar Clientes.
20
    * @author Prof. Dr. David Buzatto
21
    */
  @WebServlet( name = "ClientesServlet",
23
                urlPatterns = { "/processaClientes" } )
24
  public class ClientesServlet extends HttpServlet {
25
26
       protected void processRequest(
27
               HttpServletRequest request,
               HttpServletResponse response )
               throws ServletException, IOException {
30
31
           String acao = request.getParameter( "acao" );
32
          ClienteDAO dao = null;
          RequestDispatcher disp = null;
35
          DateTimeFormatter dtf = DateTimeFormatter.ofPattern("yyyy-MM-dd");
37
          try {
               dao = new ClienteDAO();
41
               if ( acao.equals( "inserir" ) ) {
42
43
                   String nome = request.getParameter( "nome" );
44
                   String sobrenome = request.getParameter( "sobrenome" );
45
```

```
String dataNascimento =
46

→ request.getParameter("dataNascimento");
                   String cpf = request.getParameter( "cpf" );
47
                   String email = request.getParameter( "email" );
48
                   String logradouro = request.getParameter( "logradouro" );
49
                   String numero = request.getParameter( "numero" );
50
                   String bairro = request.getParameter( "bairro" );
                   String cep = request.getParameter( "cep" );
52
                   int idCidade = Integer.parseInt(
53
                            request.getParameter( "idCidade" ) );
54
55
                   Cidade ci = new Cidade();
56
                   ci.setId( idCidade );
57
58
                   Cliente c = new Cliente();
59
                   c.setNome( nome );
60
                   c.setSobrenome( sobrenome );
61
                   c.setDataNascimento( Date.valueOf(
62
                            LocalDate.parse( dataNascimento, dtf ) );
63
                   c.setCpf( cpf );
64
                   c.setEmail( email );
65
                   c.setLogradouro( logradouro );
66
                   c.setNumero( numero );
67
                   c.setBairro( bairro );
68
                   c.setCep( cep );
69
                   c.setCidade( ci );
70
71
                   dao.salvar( c );
72
73
                   disp = request.getRequestDispatcher(
74
                            "/formularios/clientes/listagem.jsp" );
75
76
               } else if ( acao.equals( "alterar" ) ) {
77
78
                   int id = Integer.parseInt(request.getParameter( "id" ));
79
                   String nome = request.getParameter( "nome" );
                   String sobrenome = request.getParameter( "sobrenome" );
81
                   String dataNascimento = request.getParameter( "dataNascimento"
82
                    → );
                   String cpf = request.getParameter( "cpf" );
83
                   String email = request.getParameter( "email" );
84
                   String logradouro = request.getParameter( "logradouro" );
85
```

```
String numero = request.getParameter( "numero" );
86
                     String bairro = request.getParameter( "bairro" );
87
                     String cep = request.getParameter( "cep" );
88
                     int idCidade = Integer.parseInt(
89
                             request.getParameter( "idCidade" ) );
90
91
                     Cidade ci = new Cidade();
                     ci.setId( idCidade );
93
94
                     Cliente c = new Cliente();
95
                     c.setId( id );
96
                     c.setNome( nome );
97
                     c.setSobrenome( sobrenome );
                     c.setDataNascimento( Date.valueOf(
99
                             LocalDate.parse( dataNascimento, dtf ) );
100
                     c.setCpf( cpf );
101
                     c.setEmail( email );
102
                     c.setLogradouro( logradouro );
103
                     c.setNumero( numero );
104
                     c.setBairro( bairro );
105
                     c.setCep( cep );
106
                     c.setCidade( ci );
107
108
                     dao.atualizar( c );
109
110
                     disp = request.getRequestDispatcher(
111
                              "/formularios/clientes/listagem.jsp" );
112
113
                } else if ( acao.equals( "excluir" ) ) {
114
115
                     int id = Integer.parseInt(request.getParameter( "id" ));
116
117
118
                     Cliente c = new Cliente();
                     c.setId( id );
119
120
                     dao.excluir( c );
121
122
                     disp = request.getRequestDispatcher(
123
                              "/formularios/clientes/listagem.jsp" );
124
125
                } else {
126
127
```

```
int id = Integer.parseInt(request.getParameter( "id" ));
128
                     Cliente c = dao.obterPorId( id );
129
                     request.setAttribute( "cliente", c );
130
131
                     if ( acao.equals( "prepararAlteracao" ) ) {
132
                         disp = request.getRequestDispatcher(
133
                                  "/formularios/clientes/alterar.jsp" );
134
                     } else if ( acao.equals( "prepararExclusao" ) ) {
135
                         disp = request.getRequestDispatcher(
136
                                  "/formularios/clientes/excluir.jsp" );
137
                     }
138
139
                }
140
141
            } catch ( SQLException exc ) {
142
                 exc.printStackTrace();
143
            } finally {
144
                 if ( dao != null ) {
145
                     try {
146
                         dao.fecharConexao();
147
                     } catch ( SQLException exc ) {
148
                         exc.printStackTrace();
149
150
                }
151
            }
152
153
            if ( disp != null ) {
154
                disp.forward( request, response );
155
            }
156
157
        }
158
159
        @Override
160
        protected void doGet(
161
                HttpServletRequest request,
162
                HttpServletResponse response )
163
                throws ServletException, IOException {
164
            processRequest( request, response );
165
        }
166
167
        @Override
168
        protected void doPost(
169
```

```
HttpServletRequest request,
170
                 HttpServletResponse response )
171
                 throws ServletException, IOException {
172
            processRequest( request, response );
173
        }
174
175
        @Override
176
        public String getServletInfo() {
177
            return "ClientesServlet";
178
179
        }
180
181 }
```

Note que no início da Listagem 5.28 (linha 36), nós usamos a classe java.time.format.DateTimeFormatter para converter a data inserida pelo usuário no formulário, que vem como texto, para um objeto do tipo java.sql.Date. Para fazer essa conversão, precisamos criar o DateTimeFormatter com o formato que a data chegará, no caso "yyyy-MM-dd", ou seja, ano com quatro dígitos (yyyy), mês com dois dígitos (MM) e dia com dois dígitos (dd). Com o formatador criado, usamos o método parse(...) de java.time.LocalDate, sendo que a String com a data que é passada para o método parse(...) deve estar no formato especificado no construtor do DateTimeFormatter. Ainda, o objeto do tipo java.time.LocalDate é usado como parâmetro do método valueOf(...) da classe java.sql.Date para converter o java.time.LocalDate para java.sql.Date e configurar no objeto do tipo Cliente que está sendo populado com os dados apropriados.

```
<meta name="viewport"
11
           content="width=device-width, initial-scale=1.0">
12
      <link rel="stylesheet"</pre>
13
           href="${cp}/css/estilos.css"/>
14
    </head>
15
16
    <body>
17
18
      <h1>Alterar Cliente</h1>
19
20
      <form method="post" action="${cp}/processaClientes">
21
22
        <input name="acao" type="hidden" value="alterar"/>
23
        <input name="id" type="hidden" value="${requestScope.cliente.id}"/>
24
25
        26
          27
           Nome:
28
           29
             <input name="nome"</pre>
30
                    type="text"
31
                    size="20"
32
                    maxlength="45"
33
                    required
34
                    value="${requestScope.cliente.nome}"/>
35
           36
          37
          38
           Sobrenome:
39
           40
41
             <input name="sobrenome"</pre>
                    type="text"
42
                    size="20"
43
                    maxlength="45"
44
                    required
45
                    value="${requestScope.cliente.sobrenome}"/>
46
           47
          48
          49
           Data de Nascimento:
50
           51
             <fmt:formatDate
52
```

```
pattern="yyyy-MM-dd"
53
                 value="${requestScope.cliente.dataNascimento}"
54
                 var="data" scope="page"/>
55
             <input name="dataNascimento"</pre>
56
                    type="date"
57
                    size="8"
58
                    placeholder="dd/mm/yyyy"
                    required
60
                    value="${data}"/>
61
           62
          63
64
          CPF:
           >
66
             <input name="cpf"</pre>
67
                    type="text"
68
                    size="13"
69
                    pattern="\d{3}.\d{3}.\d{3}-\d{2}"
70
                    placeholder="999.999.999-99"
71
                    required
72
                    value="${requestScope.cliente.cpf}"/>
73
           74
          75
          76
           E-mail:
77
           >
78
             <input name="email"</pre>
79
                    type="email"
80
                    size="20"
81
                    maxlength="60"
                    required
83
                    value="${requestScope.cliente.email}"/>
84
           85
          86
          87
           Logradouro:
           >
             <input name="logradouro"</pre>
90
                    type="text"
91
                    size="25"
92
                    maxlength="50"
93
                    required
94
```

```
value="${requestScope.cliente.logradouro}"/>
95
           96
         97
         98
           Número:
99
           100
             <input name="numero"</pre>
101
                  type="text"
102
                   size="6"
103
                  maxlength="6"
104
                  required
105
                   value="${requestScope.cliente.numero}"/>
106
           107
         108
109
         Bairro:
110
           111
             <input name="bairro"</pre>
112
                   type="text"
113
114
                   size="15"
                  maxlength="30"
115
                   value="${requestScope.cliente.bairro}"/>
116
           117
         118
         119
           CEP:
120
           121
122
             <input name="cep"</pre>
123
                   type="text"
                   size="7"
124
                   pattern="\d{5}-\d{3}"
125
                  placeholder="99999-999"
126
127
                  required
                   value="${requestScope.cliente.cep}"/>
128
           130
         131
           Cidade:
132
           133
134
135
             <jsp:useBean
                id="servicos"
136
```

```
scope="page"
137
                   class="cadastroclientes.servicos.CidadeServices"/>
138
139
               <select name="idCidade" required>
140
                 <c:forEach items="${servicos.todos}" var="cidade">
141
                   <c:choose>
142
                     <c:when test="${requestScope.cliente.cidade.id eq
143

    cidade.id}">

                       <option value="${cidade.id}" selected>
144
                          ${cidade.nome}
145
                        </option>
146
                     </c:when>
147
                      <c:otherwise>
148
                       <option value="${cidade.id}">
149
                         ${cidade.nome}
150
                       </option>
151
                     </c:otherwise>
152
                   </c:choose>
153
                 </c:forEach>
154
               </select>
155
156
             157
           158
           159
             160
               <a href="${cp}/formularios/clientes/listagem.jsp">Voltar</a>
161
             162
             163
               <input type="submit" value="Alterar"/>
164
             165
           166
         167
168
       </form>
169
170
171
     </body>
172
   </html>
173
```

Para que possamos apresentar a data de nascimento (tipo java.sql.Date) do cliente que está passando pelo processo de edição em um formato específico, nós usamos a *tag* (fmt:formatDate) (linha 52 da Listagem 5.29), que faz parte da JSTL. Note

que a TagLib que contém essa *tag* é declarada no início da Listagem 5.29 usando o prefixo fmt. Na *tag* <fmt:formatDate>, usamos o atributo pattern (padrão) para configurarmos o formato da String que deve ser gerada a partir da data de nascimento do cliente. O atributo value é usado para indicar o objeto da data que faz parte do objeto cliente contido no requestScope, ou seja, o objeto que queremos formatar. O atributo var é usado para indicar o nome da variável usada para armazenar o resultado da formatação, sendo que no caso, configuramos o nome da variável como data. Por fim, o atributo scope é usado para definir o escopo que essa variável vai existir, no caso, cofiguramos para page, ou seja, a variável só vai existir dentro dessa página. A seguir, como valor do input da data de nascimento, usamos a variável data, obtida usando EL na forma \${data}.

```
Listagem 5.30: Formulário de exclusão de Clientes cadastrados
   Arquivo: /formularios/clientes/excluir.jsp
  %@page contentType="text/html" pageEncoding="UTF-8"%>
  %0taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
3 %@taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
4 <c:set var="cp" value="${pageContext.request.contextPath}"/>
  <!DOCTYPE html>
6
  <html>
7
     <head>
8
       <title>Excluir Cliente</title>
9
       <meta charset="UTF-8">
10
       <meta name="viewport"</pre>
11
             content="width=device-width, initial-scale=1.0">
12
       <link rel="stylesheet"</pre>
13
             href="${cp}/css/estilos.css"/>
14
     </head>
15
16
17
     <body>
18
       <h1>Excluir Cliente</h1>
19
20
       <form method="post" action="${cp}/processaClientes">
21
22
         <input name="acao" type="hidden" value="excluir"/>
23
         <input name="id" type="hidden" value="${requestScope.cliente.id}"/>
24
25
         26
```

```
27
       Nome:
28
       ${requestScope.cliente.nome}
29
      30
      31
       Sobrenome:
32
       ${requestScope.cliente.sobrenome}
      34
      35
       Data de Nascimento:
36
       >
37
38
        <fmt:formatDate
          pattern="dd/MM/yyyy"
          value="${requestScope.cliente.dataNascimento}"/>
40
       41
      42
      43
       CPF:
44
       ${requestScope.cliente.cpf}
      46
      47
       E-mail:
48
       ${requestScope.cliente.email}
49
      50
      Logradouro:
52
       ${requestScope.cliente.logradouro}
53
54
      55
       Número:
56
       ${requestScope.cliente.numero}
      58
      59
       Bairro:
60
       ${requestScope.cliente.bairro}
61
      CEP:
64
       ${requestScope.cliente.cep}
65
      66
      67
       Cidade:
68
```

```
${requestScope.cliente.cidade.nome}
69
         70
         71
          72
            <a href="${cp}/formularios/clientes/listagem.jsp">
73
              Voltar
74
            </a>
75
          76
          77
            <input type="submit" value="Excluir"/>
78
          79
80
         81
82
     </form>
83
84
    </body>
85
86
  </html>
87
```

Veja que no final da Listagem 5.30 usamos novamente um formatador de datas (tag <fmt:formatDate>) para apresentar a data de nascimento do cliente. Note que desta vez o uso do formatador foi simplificado, visto que agora não precisamos inserir a data formatada dentro de um input como fizemos na Listagem 5.29. Quando queremos apenas exibir a data formatada, basta usarmos a tag <fmt:formatDate> com os atributos "pattern" e "value" configurados que a data formatada será gerada onde a tag foi usada.

Ufa! Quanta coisa! Copiou todos os arquivos? Testou tudo o que você fez? Que bom! Se tudo funcionou, parabéns! Caso tenha dado algum problema, verifique o que pode ter acontecido, principalmente comparando o código que você copiou com o código das listagens. Agora você é capaz de criar uma aplicação Web em Java que contenha cadastros. Muito legal não é mesmo? Com essa bagagem teórica e prática que tivemos neste e nos Capítulos anteriores, nós seremos capazes de trabalhar no projeto que será proposto no Capítulo 6.

5.6 Resumo

Neste Capítulo construímos uma aplicação Web completa que mantém o cadastro de Clientes, Cidades e Estados. Durante o nosso aprendizado, nós usamos os padrões que aprendemos no Capítulo 4 para criar a arquitetura da nossa aplicação, dividindo

5.7. PROJETOS 173

as responsabilidades entre as classes, bem como usando as camadas propostas no padrão MVC, organizando assim o nosso projeto.

5.7 Projetos

Projeto 5.1: Na listagem de clientes, insira uma nova coluna na tabela para apresentar a data de nascimento de cada cliente. Use a *tag* (fmt:formatDate) para formatar a data no formato dd/MM/yyyy (dia com dois dígitos, mês com dois dígitos e ano com quatro dígitos). Não se esqueça de usar a diretiva <%@taglib . . . %> para configurar a TagLib de formatadores da JSTL.

Projeto 5.2: No projeto implementado durante o Capítulo, implementamos a validação dos formulários através de diversos atributos dos *inputs*. Como você deve saber, esse tipo de validação pode ser contornado pelo usuário usando a inspeção do código fonte pelas ferramentas do desenvolvedor. Tente criar um mecanismo de validação dos dados fornecidos pelo usuário no cadastro de estados. Caso algum campo não tenha as características necessárias (sigla com mais de dois caracteres, por exemplo), direcione o usuário para uma página de erro, onde deve ser apresentado ao usuário qual erro ocorreu. Nessa página deve existir um botão "Voltar", que levará o usuário de volta à listagem de estados.

Projeto 5.3: Crie um mecanismo de validação de dados para o cadastro de cidades, da mesma forma que você fez para o cadastro de estados.

Projeto 5.4: Crie um mecanismo de validação de dados para o cadastro de clientes, da mesma forma que você fez para o cadastro de estados.

PRIMEIRO PROJETO: SISTEMA PARA LOCAÇÃO DE DVDs v1.0

"Só se conhece o que se pratica".

Barão de Montesquieu



ESTE Capítulo aplicaremos o conhecimento adquirido até o momento na construção de uma aplicação Web em Java.

6.1 Introdução

Neste Capítulo será apresentada uma série de requisitos que devem ser usados para criar uma aplicação Web da mesma forma que fizemos no Capítulo 5. Tudo que será requisitado estará baseado no que já aprendemos, sendo assim, todas as funcionalidades requeridas poderão ser implementadas com recursos já vistos no Capítulo 5. Note que apesar do projeto ser intitulado como um sistema para locação de DVDs, por enquanto a implementação da locação em si será deixada de lado, pois você a fará no projeto do Capítulo 9. Isso se dá porque ainda precisamos aprender mais algumas coisas, principalmente do lado do cliente, para sermos capazes de implementar cadastros que lidem com relacionados muitos-para-muitos.

Apresentação dos Requisitos **6.2**

Você foi contratado para criar um sistema para controle de cadastro de DVDs. Esse sistema irá manter apenas o cadastro de DVDs e não irá gerenciar a locação dos mesmos. Os dados que deverão ser mantidos para todos os DVDs são: título, ano de lançamento, ator principal, ator coadjuvante, data de lançamento, duração em minutos, gênero e classificação etária. Os dados dos atores, dos gêneros e das classificações etárias deverão ser mantidos através de cadastros específicos. Um ator deve ter um nome, um sobrenome e uma data de estreia (indica quando foi o primeiro filme do ator). Um gênero deve ter uma descrição. Uma classificação etária deve ter também apenas uma descrição. Cada um dos cadastros (DVD, ator, gênero e classificação etária), deve conter as funcionalidades de criar, alterar e excluir um determinado registro. A página principal da aplicação deve conter um link para cada tipo de cadastro.

Desenvolvimento do Projeto 6.3

O projeto Web que deverá ser criado deve ter o nome de "LocacaoDVDs". Configure o projeto para conter as bibliotecas internamente. O pacote de código-fonte base do projeto deve ter o nome de "locacaodvds". A estrutura do projeto deve ser igual à estrutura do projeto criado no Capítulo 5, sendo que, obviamente, o nome das classes e suas respectivas implementações serão diferentes do projeto daquele Capítulo. A base de dados com as tabelas das entidades obtidas a partir da análise dos requisitos na seção anterior deve ter o nome de "locacao dvds". Note que nos arquivos de código fonte disponibilizados, na pasta deste Capítulo, é fornecido um arquivo do MySQL Workbench com um modelo do cenário apresentado acima. Não se esqueça de que cada entidade deverá conter um identificador. As páginas da aplicação deverão ter sua aparência configurada usando uma folha de estilos, da mesma forma que fizemos no projeto do Capítulo 5. Personalize os estilos, mudando as cores etc., além de usar imagens ou quaisquer outros artifícios que julgar interessante para mudar a aparência da sua aplicação.

6.4 Resumo

Neste Capítulo foi requisitado que você implementasse uma aplicação Web em Java para gerenciar o cadastro de DVDs de uma locadora (ainda sem a locação), por isso, não há atividades a serem realizadas.

Parte II O Que Falta do Básico?

Introdução à Linguagem JavaScript

"A vida vai ficando cada vez mais dura perto do topo".

Friedrich Nietzsche



ESTE Capítulo temos como objetivo aprender as construções básicas da linguagem de programação JavaScript, vastamente utilizada no desenvolvimento de aplicações Web.

7.1 Introdução

Chegamos a talvez à cereja do bolo, ou à cereja do livro ou então à cereja do desenvolvimento para Web: a linguagem de *script* JavaScript. A primeira coisa que precisamos deixar claro é que Java e JavaScript são duas linguagens diferentes, com sintaxe baseada em C e com construções similares, mas o comum entre as duas para por aqui. A linguagem JavaScript não é uma linguagem orientada a objetos, mas sim baseada em protótipos. Nesse tipo de linguagem não existem classes, mas apenas objetos. Novos objetos são criados a partir de cópias de objetos existentes. O JavaScript "moderno", baseado no padrão ECMAScript 2015 (sexta edição), possui algumas construções que lembram àquelas de linguagens orientadas a objetos como classes e herança, mas tudo isso é *syntax sugar* para simplificar coisas que já existiam anteriormente. Outras

construções também são suportadas na linguagem como as funções de primeira classe etc.

Neste Capítulo não entraremos em detalhes sobre a história da linguagem e da sua evolução, mas sim no que eu acho útil e fundamental para que possamos começar a usá-la. Iremos ter uma visão geral da linguagem como declaração de variáveis, manipulação do *Document Object Model* (DOM) e requisições assíncronas. Durante o Capítulo serão apresentadas inúmeras caixas do tipo "Saiba Mais" com *links* úteis. A maioria desses links serão da *Mozilla Developer Network*¹ (MDN), uma referência confiável e oficial da maioria, senão de todas, das tecnologias para Web. Sempre fornecerei os *links* da versão em inglês do site, mas se quiser, você pode verificar a versão em português clicando no botão "*Change language*" presente em todas as páginas do site. Sinceramente recomendo a leitura em inglês, pois o texto sempre estará completo, atualizado com a terminologia correta.

(i) | Saiba Mais

Quer conhecer um pouco mais da história e dos detalhes da linguagem JavaScript? Veja o *link* https://developer.mozilla.org/en-US/docs/Web/JavaScript.

(i) | Saiba Mais

A referência da linguagem JavaScript pode ser acessada pelo *link* https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference.

(i) | Saiba Mais

As documentações e referências das tecnologias e APIs usadas para o desenvolvimento para Web podem ser acessadas pelo *link* https://developer.mozilla.org/en-US/docs/Web/API.

(i) | Saiba Mais

Caso deseje fazer um tutorial completo sobre a linguagem, recomendo o ótimo tutorial da própria MDN: https://developer.mozilla.org/en-US/docs/Learn/J avaScript>.

Antes de começarmos a falar do JavaScript propriamente dito, vamos montar nosso palco, que é um projeto Java para Web. Neste Capítulo ainda faremos a construção

¹<https://developer.mozilla.org/>

dos projetos do zero, mas a partir do próximo focarei apenas nas novidades que serão apresentadas.

Vamos lá. Crie um projeto Java Web com o nome de "ExemplosEmJavaScript" da forma que tem feito até aqui. Os passos descritos a seguir são somente estruturais. Os respectivos códigos dos arquivos serão apresentados e explicados posteriormente. Sendo assim, no nó *Web Pages* do projeto:

- Remova o arquivo index.html;
- Crie um JSP chamado index.jsp;
- Crie uma pasta chamada css;
- Crie uma pasta chamada js (JavaScript);
- Dentro da pasta css crie um arquivo CSS chamado estilos.css;
- Dentro da pasta js crie 12 arquivos JavaScript, chamados exemplo01. js, exemplo02. js... exemplo12. js;
- Em <u>Source Packages</u> crie os pacotes exemplosemjavascript.pojo e exemplosemjavascript.servlets;
- No pacote exemplosemjavascript.pojo crie uma classe chamada Pessoa;
- No pacote exemplosemjavascript.servlets crie os Servlets CalculaTabuadaServlet e ListagemPessoasServlet;
- Clique com o botão direito do mouse no nó raiz do projeto e escolha o último item do menu de contexto, chamado *Properties*;
 - Do lado esquerdo, em Categories: , clique no item CDNJS , situado dentro do nó JavaScript Libraries ;
 - Do lado direito, clique no botão (Add);
 - No diálogo que abriu, intitulado (Add CDNJS Library), preencha o campo
 [Find:] com "jquery" (sem as aspas) e clique em (Search);
 - Após a busca na Content Delivery Network (CDN) aparecerão diversos componentes na Graphical User Interface (GUI). Em Libraries: escolha jquery, provavelmente o primeiro item;
 - Em *Files*: marque a *checkbox* na frente do item jquery.min.js e clique em *Add Library* como na Figura 7.1;

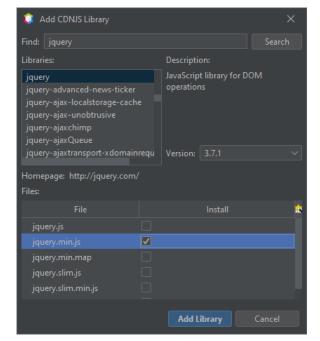


Figura 7.1: Adicionando uma biblioteca JavaScript

Fonte: Elaborada pelo autor

 Clique em OK. A biblioteca jQuery será baixada e inserida no projeto dentro da pasta js/libs/jquery.

Realizando todos os passos descritos anteriormente, você terá um projeto com a estrutura apresentada na Figura 7.2. Agora vamos começar a preencher cada um dos arquivos e aprender o que está acontecendo em cada um deles.

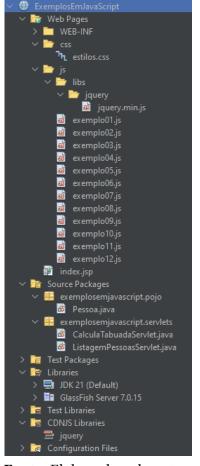


Figura 7.2: Estrutura do projeto

Fonte: Elaborada pelo autor

Começaremos com o index. jsp apresentado na Listagem 7.1. Entre as linhas 10 e 22 usamos a tag <script> para carregarmos no documento treze arquivos de código JavaScript. Inclusive, essa mesma tag pode ser utilizada para inserir código JavaScript no próprio documento. Veremos isso mais adiante. Na linha 10 é carregada a biblioteca jQuery que há alguns anos atrás era absolutamente relevante, mas que hoje em dia tem caído em desuso visto a evolução do JavaScript. Ela será tratada no livro pela sua importância em software legado, por facilitar e padronizar algumas coisas e também, é claro, por preferência minha :D. No restante das linhas são associados os arquivos com os exemplos que aprenderemos. O restante do documento consiste na construção de uma GUI com alguns componentes e tags que serão manipulados pelo nosso código em JavaScript.

Os cinco primeiros exemplos são relativos às construções principais da linguagem.

Veja que da linha 30 à 34 temos um parágrafo (tag ()) com um botão dentro (tag (<button>)) e que em seu evento click, representado pelo atributo onclick, é registrado uma função tratadora/manipuladora/ouvinte de evento (event handler ou event listener). Para registrar uma função como tratadora de um determinado evento, basta inserir seu nome no valor do atributo onclick e adicionar, entre os parânteses da mesma, a palavra event. Esse event carregará o objeto do evento que será disparado pela tag e ouvido pela função. Essa função precisa estar declarada e implementada em algum lugar. No nosso caso, estará no arquivo exemplo01. js, referenciado acima. Note que como o JavaScript é interpretado, para se poder usar algo, essa "coisa" precisa ter sido declarada antes ou as vezes "vista" pelo interpretador pela primeira vez, sendo que esse segundo comportamento pode gerar muitos problemas caso não seja entendido apropriadamente, mas veremos isso também. Resumindo, ao se clicar (onclick) nesse primeiro botão, afunção executarExemplo01(event) será invocada. Fácil não é? Existe uma infinidade de eventos permitidos para cada tag, mas falaremos de alguns deles à medida que for necessário. Esse padrão de um botão invocando uma função se repetirá em praticamente todos os exemplos. Os cinco primeiros têm a mesma estrutura.

Nos exemplos 06, 07 e 08 trataremos da manipulação das *tags*, como dinamicamente inserir conteúdo nas mesmas ou ler/escrever dados em componentes de formulário. Esses exemplos estão dentro da seção "Manipulação do DOM", onde DOM significa *Document Object Mode*l que nos bastidores é uma árvore composta de objetos que representam o resultado do processo de *parsing* do arquivo HTML pelo navegador ou outro tipo de cliente. Usando JavaScript podemos mexer nessa árvore, alterando atributos dos nós, que na maioria das vezes representam as *tags*, além de inserir e remover nós. Todas as modificações são replicadas automaticamente pelo navegador que entende que a árvore foi alterada e precisa ser redesenhada no processo de renderização do documento. Antigamente, quando isso era novidade há mais de 20 anos, era chamado de "HTML Dinâmico" (*Dynamic* HTML (DHTML)). Sim, estou ficando velho:D. Perceba que nesses exemplos, além dos botões temos *divs* que serão usadas para mostrar o resultado de algum processamento, além de componentes de formulário e outros botões no exemplo 08.

No exemplo 09 falaremos um pouco de tratamento de eventos, como já comentei anteriormente. Nesse exemplo, na linha 171, temos a invocação da função registrarEventosExemplo09() em que, programaticamente, faremos o registro dos ouvintes de eventos ao invés de usar os atributos prefixados com "on" das *tags*.

No exemplo 10 trataremos do uso da tag(<canvas>) usada para desenhar programaticamente, realizando uma simulação física de uma bolinha.

Nos dois últimos exemplos trataremos das requisições assíncronas e de intercâmbio

de dados entre cliente e servidor.

```
Listagem 7.1: Página principal da aplicação
   Arquivo: /index.jsp
  %@page contentType="text/html" pageEncoding="UTF-8"%>
  %@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
  <c:set var="cp" value="${pageContext.request.contextPath}"/>
  <!DOCTYPE html>
  <html>
     <head>
       <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
8
       <title>Exemplos em JavaScript</title>
9
       <script src="${cp}/js/libs/jquery/jquery.min.js"></script>
10
       <script src="${cp}/js/exemplo01.js"></script>
11
       <script src="${cp}/js/exemplo02.js"></script>
12
       <script src="${cp}/js/exemplo03.js"></script>
13
14
       <script src="${cp}/js/exemplo04.js"></script>
15
       <script src="${cp}/js/exemplo05.js"></script>
       <script src="${cp}/js/exemplo06.js"></script>
16
       <script src="${cp}/js/exemplo07.js"></script>
17
       <script src="${cp}/js/exemplo08.js"></script>
18
       <script src="${cp}/js/exemplo09.js"></script>
19
       <script src="${cp}/js/exemplo10.js"></script>
20
       <script src="${cp}/js/exemplo11.js"></script>
21
       <script src="${cp}/js/exemplo12.js"></script>
22
       <link rel="stylesheet" href="${cp}/css/estilos.css"/>
23
     </head>
24
     <body>
25
       <div>
26
27
         <h1>Construções da Linguagem</h1>
28
29
         >
           <button onclick="executarExemplo01(event)">
31
             Exemplo 01 - Funções de E/S e Operadores Aritméticos
32
           </button>
33
         34
36
           <button onclick="executarExemplo02(event)">
37
```

```
Exemplo 02 - Declarações de Variáveis e Suas Implicações
38
           </button>
39
         40
41
42
         >
           <button onclick="executarExemplo03(event)">
43
             Exemplo 03 - Estruturas Condicionais e Operadores
44
           </button>
45
         46
47
         >
48
           <button onclick="executarExemplo04(event)">
49
             Exemplo 04 - Estruturas de Repetição e Arrays
50
           </button>
51
         52
53
         >
54
           <button onclick="executarExemplo05(event)">
55
             Exemplo 05 - "Classes" e Objetos e JSON
           </button>
57
         58
       </div>
59
60
       <hr>>
61
62
       <div>
63
64
         <h1>Manipulação do DOM</h1>
65
66
         <div>
67
68
           >
             <button onclick="executarExemplo06(event)">
69
               Exemplo 06 - JavaScript Puro
70
             </button>
71
72
           <div id="divExemplo06" class="divExemplo"></div>
73
         </div>
74
75
         <div>
76
           >
77
             <button onclick="executarExemplo07(event)">
78
               Exemplo 07 - Usando jQuery
79
```

```
</button>
80
             81
             <div id="divExemplo07" class="divExemplo"></div>
82
          </div>
83
84
          <div>
             <h2>Exemplo 08 - Manipulação de Formulários</h2>
            <div id="divExemplo08" class="divExemplo">
87
88
               <form id="form08">
89
90
91
                 Campo 1:
                 <input id="campo01"</pre>
                         type="text"
93
                        name="campo01"
94
                         value="valor campo 01"/>
95
                 <br/>
96
97
                 Campo 2:
                 <input id="campo02"</pre>
99
                         type="text"
100
                        name="campo02"
101
102
                         value="valor campo 02"/>
                 <br/>
103
104
                 Select 3:
105
                 <select id="select03" name="select03">
106
107
                   <option value="o1">Opção 1</option>
                   <option value="o2">Opção 2</option>
108
                   <option value="o3">Opção 3</option>
109
                 </select>
110
                 <br/>
111
112
                 Select 4:
113
                 <select id="select04" name="select04" size="4">
114
115
                   <option value="o1">Opção 1</option>
                   <option value="o2" selected>Opção 2</option>
116
                   <option value="o3">Opção 3</option>
117
                 </select>
118
119
                 Área 5:
120
                 <textarea id="area05"</pre>
121
```

```
name="area05"
122
                            rows="5" cols="10">
123
                     valor da área de texto 05
124
                 </textarea>
125
              </form>
126
              <hr>>
127
              <button onclick="lerDadosFormulario(event)">
128
                Ler dados
129
              </button>
130
              <button onclick="lerDadosFormularioJQuery(event)">
131
                Ler dados jQuery
132
133
              </button>
              <hr>>
134
              <button onclick="inserirDadosFormulario(event)">
135
                Inserir dados
136
              </button>
137
              <button onclick="inserirDadosFormularioJQuery(event)">
138
139
                 Inserir dados jQuery
              </button>
140
              <hr>>
141
              <button onclick="inserirNovaOpcao(event)">
142
                Inserir nova opção (Select 03)
143
              </button>
144
              <button onclick="inserirNovaOpcaoJQuery(event)">
145
                Inserir nova opção jQuery (Select 04)
146
              </button>
147
              <hr>>
148
            </div>
149
          </div>
150
151
          <div>
152
            >
153
154
              <h2>Exemplo 09 - Eventos</h2>
            155
            <div id="divExemplo09" class="divExemplo">
156
157
              Campo (digite algo e veja o console):
158
              <input id="campoExemplo09"/>
159
              <br/>
160
161
              <select id="selectExemplo09">
162
                 <option value="o1">Opção 1</option>
163
```

```
<option value="o2">Opção 2</option>
164
                 <option value="o3">Opção 3</option>
165
               </select>
166
167
            </div>
168
             <!-- precisa executar depois das
169
                  tags estarem prontas! -->
170
            <script>registrarEventosExemplo09();</script>
171
          </div>
172
173
          <div>
174
175
            >
               <h2>Exemplo 10 - Simulação Usando a Canvas API</h2>
176
            177
            <div id="divExemplo10" class="divExemplo">
178
               <canvas id="canvasExemplo10"</pre>
179
                       height="260"
180
                       width="550"
181
                       class="canvasExemplo10"/>
182
            </div>
183
            <script>prepararCanvasExemplo10();</script>
184
          </div>
185
186
        </div>
187
188
        <hr>>
189
190
191
        <div>
192
          <h1>Requisições Assíncronas e Intercâmbio de Dados</h1>
193
194
          <div>
195
196
            >
               <button onclick="executarExemplo11jQuery(event)">
197
                 Exemplo 11 - AJAX com jQuery
198
               </button>
199
               <button onclick="executarExemplo11Fetch(event)">
200
                 Exemplo 11 - AJAX com Fetch API
201
               </button>
202
            203
            <div id="divExemplo11" class="divExemplo"></div>
204
          </div>
205
```

```
206
207
          <div>
            >
208
              <button onclick="executarExemplo12jQuery(event)">
209
                Exemplo 12 - AJAX com jQuery e JSON
210
              </button>
211
              <button onclick="executarExemplo12Fetch(event)">
212
                Exemplo 12 - AJAX com Fetch API e JSON
213
              </button>
214
215
            <div id="divExemplo12" class="divExemplo"></div>
216
217
          </div>
218
        </div>
219
220
      </body>
221
222 </html>
```

Caso queira, durante os testes de execução, comente trechos do código do index. jsp para que você não precise ficar rolando a página para chegar em alguma parte toda vez que for testar uma funcionalidade.

Na Listagem 7.2 é apresentado o arquivo com as folhas de estilo usadas no index.jsp. O código já contém os comentários para você entender o que se trata cada coisa.

```
Listagem 7.2: Folhas de estilo do projeto
  Arquivo: /css/estilos.css
1 /* estilos para a tag body */
  body {
2
3
      font-family: monospace;
      font-size: 12px;
4
5 }
6
  /* estilos para o seletor de classe .divExemplo
7
   * seletores de classe iniciam com ponto (.) e são
9
   * aplicados usando o atributo class das tags.
10
  .divExemplo {
11
      border-color: #000000;
12
      border-width: 2px;
13
      border-style: solid;
14
```

```
border-radius: 5px;
15
      height: 300px;
16
      width: 600px;
17
       overflow: scroll; /* se houver estouro, permitir rolagem */
18
       overflow-style: scrollbar; /* mostrar barra de rolagem */
19
20 }
21
  /* estilo para o seletor de classe .pDOM usando
   * a pseudo-classe nth-child, indicando a seleção
   * de todos os filhos pares
24
   */
25
  .pDOM:nth-child(even) {
      background-color: #006699;
28
29
  .canvasExemplo10 {
      border: solid thin #000000;
31
       margin: 10px; /* margin (margem) é o espaçamento externo da tag */
32
  }
33
34
  .dadosPessoa {
      border: solid thin #006699;
36
       background-color: #ffa800;
37
      margin-bottom: 5px;
38
  }
39
  .dadosPessoa p {
41
      margin: 0px;
42
  }
43
44
  /* seleção da primeira tag  filha do elemento
   * que usar a classe .dadosPessoa
46
47
  .dadosPessoa p:nth-child(1) {
       background-color: #489eff;
49
50 }
  /* seleção da segunda tag  filha do elemento
   * que usar a classe .dadosPessoa
53
   */
54
  .dadosPessoa p:nth-child(2) {
55
       background-color: #248bff;
56
```

(i) | Saiba Mais

Sobre CSS, consulte https://developer.mozilla.org/en-US/docs/Web/CSS/Reference>">https://develop

Nas próximas três listagens serão mostrados os componentes do lado do servidor que utilizaremos para os dois últimos exemplos. Na Listagem 7.3 definimos a classe Pessoa, um *Plain Old Java Object* (POJO) ou *Value Object* (VO) que é uma classe que utilizaremos para criar objetos para transportar dados.

```
Listagem 7.3: Classe Pessoa
   Arquivo: exemplosemjavascript/pojo/Pessoa.java
  package exemplosemjavascript.pojo;
2
  import java.time.LocalDate;
3
4
  /**
5
   * Um Plain Old Java Object (POJO).
6
7
   * @author Prof. Dr. David Buzatto
8
9
10 public class Pessoa {
11
      private String nome;
12
      private LocalDate dataNasc;
13
      private double salario;
14
15
       public String getNome() {
16
           return nome;
17
       }
18
19
       public void setNome( String nome ) {
20
21
           this.nome = nome;
```

```
}
22
23
       public LocalDate getDataNasc() {
24
           return dataNasc;
25
       }
26
27
       public void setDataNasc( LocalDate dataNasc ) {
           this.dataNasc = dataNasc;
29
       }
30
31
       public double getSalario() {
32
33
           return salario;
       }
35
       public void setSalario( double salario ) {
36
            this.salario = salario;
37
       }
38
39
40 }
```

Na Listagem 7.4 é apresentado o código do Servlet CalculaTabuadaServlet, mapeado em /calcularTabuada que recebe um valor inteiro e retorna o texto representando a "tabuada" do número processado. Esse retorno é gerado pelo próprio Servlet no seu fluxo de saída (linhas 41, 42 e 43), sendo que o objeto response é configurado para indicar ao cliente que o que está chegando é no formato de texto/html (linha 28).

```
Listagem 7.4: Servlet de tabuada
Arquivo: exemplosemjavascript/servlets/CalculaTabuadaServlet.java

1 package exemplosemjavascript.servlets;

2 import java.io.IOException;
4 import java.io.PrintWriter;
5 import jakarta.servlet.ServletException;
6 import jakarta.servlet.annotation.WebServlet;
7 import jakarta.servlet.http.HttpServlet;
8 import jakarta.servlet.http.HttpServletRequest;
9 import jakarta.servlet.http.HttpServletResponse;

10

11 /**
12 * Calcula a tabuada de 0 a 10 de um valor passado como
```

```
* parâmetro na requisição e retorna o resultado como
13
    * texto puro.
14
15
    * @author Prof. Dr. David Buzatto
16
17
   @WebServlet( name = "CalculaTabuadaServlet",
18
                urlPatterns = { "/calcularTabuada" } )
   public class CalculaTabuadaServlet extends HttpServlet {
20
21
       protected void processRequest(
22
               HttpServletRequest request,
23
               HttpServletResponse response )
24
               throws ServletException, IOException {
25
26
           // resposta em texto puro codificado em UTF-8
27
           response.setContentType( "text/html;charset=UTF-8" );
28
29
           StringBuilder sb = new StringBuilder();
30
31
           int numero = Integer.parseInt(
32
                    request.getParameter( "numero" ) );
33
34
           for ( int i = 0; i <= 10; i++ ) {
35
               sb.append( String.format( "%d * %d = %d <br/>",
36
                        numero, i, numero * i ) );
37
           }
38
39
           // escreve na resposta
40
           try ( PrintWriter out = response.getWriter() ) {
41
                out.print( sb.toString() );
42
           }
43
44
       }
45
46
       @Override
47
       protected void doGet(
48
               HttpServletRequest request,
49
               HttpServletResponse response )
50
               throws ServletException, IOException {
51
           processRequest( request, response );
52
       }
53
54
```

```
@Override
55
       protected void doPost(
56
                HttpServletRequest request,
57
                HttpServletResponse response )
58
                throws ServletException, IOException {
59
           processRequest( request, response );
60
       }
62
       @Override
63
       public String getServletInfo() {
64
           return "CalculaTabuadaServlet";
65
       }
66
67
  }
68
```

Por fim, na Listagem 7.5 é apresentado o código do Servlet ListagemPessoasServlet, mapeado em /listarPessoas que indica ao cliente que os dados que serão retornados irão no formato JavaScript Object Notation² (JSON) (linha 33). Nesse Servlet é criada uma lista de objetos do tipo Pessoa, baseada na quantidade recebida via requisição e essa lista é serializada em JSON usando a camada de *bindind* JSON-B do Java/Jakarta EE. Na linha 35 é criado o objeto serializador e na linha 56 ele é usado, convertendo a lista com os objetos do tipo Pessoa em uma representação em texto, que no nosso caso é o JSON.

```
Listagem 7.5: Servlet de listagem de pessoas usando JSON
Arquivo: exemplosemjavascript/servlets/ListagemPessoasServlet.java

1 package exemplosemjavascript.pojo.Pessoa;
2 import exemplosemjavascript.pojo.Pessoa;
4 import java.io.IOException;
5 import java.io.PrintWriter;
6 import java.time.LocalDate;
7 import java.time.temporal.ChronoUnit;
8 import java.util.ArrayList;
9 import java.util.List;
10 import jakarta.json.bind.JsonbBuilder;
11 import jakarta.json.bind.Jsonb;
```

²O formato JSON será tratado no exemplo 05. Por enquanto assuma que é uma forma de codificar os dados de um objeto em forma de texto.

```
import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
17
18
   * Cria uma lista de pessoas, serializa em JSON e
19
   * retorna ao cliente.
20
21
    * @author Prof. Dr. David Buzatto
22
23
   @WebServlet( name = "ListagemPessoasServlet",
24
                urlPatterns = { "/listarPessoas" } )
   public class ListagemPessoasServlet extends HttpServlet {
26
27
       protected void processRequest(
28
               HttpServletRequest request,
29
               HttpServletResponse response )
               throws ServletException, IOException {
31
32
           response.setContentType( "application/json; charset=UTF-8" );
33
34
           Jsonb jb = JsonbBuilder.create();
35
           List<Pessoa> pessoas = new ArrayList<>();
36
37
           int quantidade = Integer.parseInt(
38
                   request.getParameter( "quantidade" ) );
39
40
           for ( int i = 1; i <= quantidade; i++ ) {</pre>
41
42
               Pessoa p = new Pessoa();
43
               p.setNome( String.format( "João da Silva %do", i ) );
44
45
               LocalDate d = LocalDate.now();
46
               d = d.plus( i, ChronoUnit.DAYS );
47
               p.setDataNasc( d );
48
49
               p.setSalario( 1000 * i );
50
               pessoas.add( p );
51
52
           }
53
```

```
54
           try ( PrintWriter out = response.getWriter() ) {
55
                out.print( jb.toJson( pessoas ) );
56
57
58
       }
59
       @Override
61
       protected void doGet(
62
                HttpServletRequest request,
63
                HttpServletResponse response )
64
65
                throws ServletException, IOException {
           processRequest( request, response );
66
       }
67
68
       @Override
69
       protected void doPost(
70
71
                HttpServletRequest request,
                HttpServletResponse response )
72
                throws ServletException, IOException {
73
           processRequest( request, response );
74
       }
75
76
       @Override
77
       public String getServletInfo() {
           return "ListagemPessoasServlet";
79
       }
80
81
   }
82
```

Agora que temos toda a infraestrutura básica do nosso projeto, podemos começar a falar sobre JavaScript. Vamos começar!

7.2 Funções de E/S e Operadores Aritméticos

Começaremos nossa breve jornada de descoberta da linguagem JavaScript aprendendo uma forma de obter dados do usuário, que normalmente não é usada em um software em produção, mas para aprender conceitos vai nos servir no momento, como gerar saída, declarar variáveis e realizar as operações aritméticas básicas. Na Listagem 7.6 pode ser visto o código completo do primeiro exemplo. Veja que logo na primeira linha há a declaração da função (executarExemplo01(event)) que é a

função que tratará o evento click do primeiro botão do index. jsp.

```
Listagem 7.6: Exemplo 01
   Arquivo: /js/exemplo01.js
  function executarExemplo01( event ) {
2
       // let indica a declaração de uma variável local
3
       // a função prompt retorna uma string
4
       let n1 = prompt( "Entre com o primeiro número" );
5
6
       // a função Number converte a string retornada
7
       // por prompt em um número
8
       let n2 = Number( prompt( "Entre com o segundo número" ) );
9
10
       // perceba que n1 é uma string!
11
       let adicao = n1 + n2;
                               // concatenação
12
       let subtracao = n1 - n2; // aqui subtração!
13
       let multiplicacao = n1 * n2;
14
       let divisao = n1 / n2;
15
       let resto = n1 % n2;
16
17
       // interpolação usando "`"
18
       let saida = \$\{n1\} + \$\{n2\} = \$\{adicao\} \ +
19
                   // aspas simples
20
                   n1 + ' - ' + n2 + ' = ' + subtracao + '\n' +
21
                    // ou duplas
                   n1 + " * " + n2 + " = " + multiplicacao + " \n" +
23
                    \$\{n1\} / \$\{n2\} = \$\{divisao\} \ +
24
                    `${n1} % ${n2} = ${resto}`;
25
26
       // um alerta. cuidado! alert é bloqueante,
27
       // assim como prompt e confirm.
28
       alert( saida );
29
30
       if ( confirm( "Mostrar a saída no console?" ) ) {
31
           // saída no console
32
           console.log( saida );
33
       }
34
35
36 }
```

Na linha 5 é declarada uma variável local usando a palavra chave let³, com o identificador n1 e atribuímos a ela o retorno da função prompt. Essa função recebe como parâmetro uma String e, ao ser executada, apresenta ao usuário um diálogo com uma mensagem (a String passada), um campo de texto, um botão de confirmação e um de cancelamento. Ao se clicar no botão de confirmação o valor fornecido do campo de texto será retornado ao chamador, no caso, atribuído à variável n1 e se o diálogo for cancelado, será retornado o valor null. O retorno, quando válido, é do tipo String. Note que não declaramos o tipo das variáveis em JavaScript, pois a tipagem das variáveis é dinâmica, visto que o tipo de cada variável depende do valor atribuído ou referenciado por ela.

Na linha 9 fazemos basicamente a mesma coisa para a variável n2, mas o retorno da função prompt é usada como argumento da função Number que converterá a String retornada por prompt em um número e então esse valor será atribuído a n2. Entre as linhas 12 e 16 declaramos cinco novas variáveis e atribuímos a elas o resultado de cinco operações. Note que como n1 referencia uma String, o operador + será tratado como operador de concatenação de Strings ao invés de adição, ou seja, n2 será convertida para String e concatenada com n1! Como os outros operadores como são aplicáveis apenas à números, n1 será convertido implicitamente e a operação será realizada. O resultado disso será visto na saída que será gerada e exibida.

Falando da saída, na linha 19 declaramos a variável saida e concatenamos diversas Strings para gerar o resultado. Em JavaScript existem três literais para Strings:

- 1. Delimitadas por aspas simples (apóstrofo): uma string
- 2. Delimitadas por aspas duplas (aspas): "outra string";
- 3. Delimitadas por acento grave (crase): <u>mais uma string</u>

Os dois primeiros são análogos, com a diferença que quando se usa aspas simples como delimitador e queremos ter uma aspas simples dentro da String, precisamos escapá-la com contrabarra (barra invertida) e as aspas duplas não precisam. Por exemplo, \(\begin{arra} a \begin{arra} b \begin{arra} c \begin{arra} c \begin{arra} a \begin{arra} b \begin{arra} c \begi \begin{arra} c \begin{arra} c \begin{arra} c \begin{arra} c \beg

O terceiro tipo de delimitador é mais interessante, pois permite que façamos a interpolação de valores dentro da String usando uma notação parecida com a da EL do Java/Jakarta EE, mas que não tem relação a não ser a sintaxe similar. Para o nosso exemplo, se n1 valer "10" e n2 valer 5, o resultado de `\${n1} e \${n2}` será 10 e 5.

³Veremos o propósito da palavra chave let no exemplo 02.

Por fim, para apresentar a String gerada, usamos duas formas. A primeira, na linha 29, com a função alert que, assim como prompt, é bloqueante, fazendo a execução do código parar naquele ponto ao esperar a interação do usuário. Essa função recebe uma String como parâmetro e a mostra num diálogo ao ser executada. A outra forma é usando a função log do objeto console, que recebe um ou vários objetos como parâmetro e os mostram no console do navegador. No nosso exemplo, a exibição no console está condicionada ao retorno da função confirm que exibe uma mensagem ao usuário e aguarda a interação. Caso o usuário confirme a mensagem, a função retornará um valor verdadeiro, usado na estrutura condicional if do exemplo.

7.3 Declarações de Variáveis e Suas Implicações

Como já dito, as variáveis em JavaScript não tem um tipo definido, visto que a linguagem é dinamicamente tipada, implicando que o tipo da variável varia de acordo com o que ela referencia. Em JavaScript temos Strings, números, valores lógicos, funções, objetos entre outros.

Toda variável em JavaScript ao ser declarada passará pelo processo de *hoisting*. Nesse processo, a variável será elevada ou içada até o início ou topo do contexto em que ela foi declarada e que passará a existir. A ideia é que quando o interpretador encontra uma declaração de variável e ela é bem sucedida, ou seja, é válida sintática e semanticamente, ela passará a existir como se houvesse sido declarada no início do escopo em que reside.

Podemos influenciar em como a inicialização das variáveis será feita. Veja a lista abaixo, temos quatro formas de declarar variáveis:

```
1. let variavel = "valor";;
2. const constante = "valor";;
3. var variavel = "valor";;
4. variavel = "valor";;
```

Quando a palavra-chave let é usada, a variável só poderá ser usada depois da sua inicialização, mesmo havendo *hoisting* para ela. O mesmo acontece com as constantes, declaradas com const. Já as variáveis declaradas com a palavra-chave var serão inicializadas com undefined. Por fim, as variáveis que são declaradas sem indicar nenhuma dessas três palavras-chave passarão a existir no escopo global, o que pode trazer uma série de problemas. Imagine que você declarou mais de uma variável com o mesmo nome em dois ou mais escopos diferentes. A declaração de fato ocorrerá quando o interpretador a encontrar pela primeira vez e, independende de onde for, ela passará a existir no escopo global e a partir desse ponto você pode perder o controle

do valor que a variável referencia se não tomar muito cuidado com o que está fazendo. O ideal é não utilizar ok?

O exemplo apresentado na Listagem 7.7 mostra todos esses efeitos quando for executado. O "problema" da variável declarada sem <u>let</u>, <u>var</u> ou <u>const</u> pode ser reproduzido ao se clicar pela segunda vez no botão do exemplo 02.

```
Listagem 7.7: Exemplo 02
  Arquivo: /js/exemplo02.js
  function executarExemplo02( event ) {
2
3
      // a declaração de variáveis em JavaScript
      // merece uma certa atenção. além de não precisarem
4
      // de um tipo na declaração, elas tem alguns comportamentos
      // dependendo de como são declaradas.
      testarVariaveis(); // erro! v1, v2 e v3 não definidas
8
      //console.log(v1); // erro, não há valor atribuído
9
      //console.log( v2 );
                              // ok, inicialização com undefined
10
      //console.log( v3 );
                            // erro, não definida na primeira execução
11
      console.log( "----" );
12
13
14
      let v1 = 10; // escopo local, só existe dentro da função.
                     // análoga a uma varíavel de pilha.
15
                     // inicialização nesse ponto.
16
                     // const tem o mesmo comportamento.
17
18
                             // erro! v1, v2 e v3 não definidas
      testarVariaveis();
19
      //console.log( v1 );
                             // ok, imprime 10!
20
      //console.log(v2); // ok, inicialização com undefined
21
                            // erro, não definida na primeira execução
      //console.log( v3 );
22
      console.log( "----"):
24
      var v2 = 20; // escopo local, só existe dentro da função.
25
                     // análoga a uma varíavel de pilha.
26
                     // inicialização com undefined
27
                     // e alteração do valor nesse ponto.
29
      testarVariaveis();
                             // erro! v1, v2 e v3 não definidas
                            // ok, imprime 10!
      //console.log( v1 );
31
      //console.log( v2 );
                            // ok, imprime 20!
32
```

```
//console.log( v3 ); // erro, não definida
33
       console.log( "----" );
34
35
       v3 = 30;
                      // variável GLOBAL!!!
36
                      // não faça isso :P
37
38
       testarVariaveis();
                            // erro! v1, v2 não definidas e v3?
                           // ok, imprime 10!
       console.log( v1 );
40
       console.log( v2 ); // ok, imprime 20!
41
       console.log( v3 ); // aqui mora o perigo...
42
43
       alert( "Clique no botão novamente e inicie o caos!" );
44
45
46 }
47
  function testarVariaveis() {
49
       try {
50
          // v1 não existe neste escopo nem
           // em um escopo externo
52
           console.log( v1 );
53
           v1++; // nunca chegará aqui
54
       } catch ( e ) {
55
           console.log( "v1 não declarada!" );
56
       }
57
58
       try {
59
          // v2 não existe neste escopo nem
60
          // em um escopo externo
61
           console.log( v2 );
62
          v2++; // nem aqui
63
       } catch ( e ) {
64
           console.log( "v2 não declarada!" );
65
       }
66
67
       try {
           // v3 passará a ser acessível quando
69
           // for encontrada pelo interpretador!
70
           console.log( v3 );
71
           v3++; // PERIGO!!!
72
       } catch ( e ) {
73
           console.log( "v3 não declarada!" );
74
```

```
75 }
76 77 }
```

Veja o exemplo, todo o código está comentado, não sendo necessário entrar em mais detalhes. Recomendo que você dê uma olhada nos *links* disponibilizados nas próximas duas caixas "Saiba Mais".

(i) | Saiba Mais

Para uma explicação mais detalhada sobre essas implicações, acesse http://www.constletvar.com/>.

(i) | Saiba Mais

Para mais detalhes sobre declaração de variáveis em JavaScript, acesse https://docs/Web/JavaScript/Reference/Statements.

Esse conceito pode gerar muita confusão, inclusive se declararmos uma variável com var no contexto global (fora de funções) ela será uma variável global (uma propriedade do objeto window), ao passo que dentro de uma função ela terá escopo local, assim como let e const, mas inicializada como undefined. Ainda, é importante frisar que uma constante tem ligação imutável (*immutable binding*) com o que ela referencia, ou seja, ela não pode receber um novo valor, mas o objeto que ela referencia pode ser modificado (ele não é imutável).

7.4 Estruturas Condicionais e Operadores

Em JavaScript temos as mesmas estruturas condicionais presentes na maioria das linguagens de programação ou seja, um if com else s aninhados e opcionais e um switch. Os operadores relacionais, de igualdade e lógicos também são os operadores padrão encontrados na maioria das linguages derivadas de C, com a adição de mais dois de identidade que são o operador de identidade (===) e o operador de não identidade (!==). Ao passo que os operadores de igualdade e de desigualdade verificam se o valor dos operandos comparados são respectivamente iguais ou diferentes, inclusive após a conversão implícita de algum deles, os operadores de identidade e de não identidade verificam, além do valor (sem conversão implícita), respectivamente, se o tipo é o mesmo ou diferente. Na Listagem 7.8 pode ser visto o

emprego das estruturas condicionais e a declaração de variáveis com alguns valores permitidos.

```
Listagem 7.8: Exemplo 03
   Arquivo: /js/exemplo03.js
1 function executarExemplo03( event ) {
2
                           // número
       let v0 = 0;
3
       let v1 = 2;
                           // número
4
       let v2 = "2";
                           // string
5
                         // boolean
       let v3 = true;
6
                      // nulo
7
       let v4 = null;
       let v5 = undefined; // indefinido
8
       let v6 = NaN;
                     // Not a Number
9
10
       // O, false (óbvio), null e undefined
11
12
       // são avaliados como falso
13
       if ( v0 ) {
14
           console.log( "não devia chegar aqui" ) // "; " não obrigatório,
15
                                                    // mas é padrão usar
16
       }
17
18
       if ( v1 ) {
19
           console.log( "aqui sim :)" );
20
       }
21
22
       // operador de igualdade
23
       // converte os tipos e testa igualdade de valor!
24
       if ( v1 == v2 ) {
25
           console.log( "como assim???" );
26
       }
27
28
       // operador de identidade
29
       // verifica se os operandos têm mesmo tipo e mesmo valor!
30
       if ( v1 === v2 ) {
31
           console.log( "aqui não!" );
32
       }
33
34
       if ( v3 ) {
35
           console.log( "óbvio!" );
36
```

```
}
37
38
       if ( v4 ) {
           console.log( "não!" );
40
41
42
       if ( v5 ) {
           console.log( "não tbm!" );
44
       }
45
46
       if ( v6 ) {
47
           console.log( "não tbm!" );
48
       }
49
       // NaN é um valor especial
51
       if ( v6 == NaN ) {
           console.log( "pq não?" );
53
       }
54
       if ( v6 === NaN ) {
56
           console.log( "uai!?" );
57
       }
58
59
       if ( isNaN( v6 ) ) {
60
           console.log( "pra NaN, só assim..." );
       }
62
63
       // operadores de igualdade:
64
              igualdade: == (mesmo valor com conversão implícita)
65
               diferente: != (valor diferente com conversão implícita)
       // operadores de identidade:
68
             identidade: === (mesmo valor e mesmo tipo)
69
       // não identidade: !== (valor diferente e tipo diferente)
70
71
       // operadores relacionais:
                  menor: <
73
       // menor ou igual: <=
74
                  maior: >
       //
75
       // maior ou igual: >=
76
77
       // operadores lógicos
78
```

```
// e lógico: 88
79
        // ou lógico: //
80
        // não lógico: !
81
82
        // veja a documentação referenciada no livro para mais detalhes
83
84
        switch ( v1 ) {
85
            case 1:
86
                console.log( "v1 vale 1" );
87
                break;
88
            case 2:
89
                console.log( "v1 vale 2" );
90
                break;
91
            default:
92
                console.log( "v1 vale alguma coisa..." );
93
                break;
94
        }
95
96
        // sem conversão automática!
        switch ( v1 ) {
98
            case "1":
99
                console.log( "v1 vale 1" );
100
101
                break;
            case "2":
102
                console.log( "v1 vale 2" );
103
                break;
104
            default:
105
106
                console.log( "v1 vale alguma coisa..." );
                break;
107
        }
108
109
        switch ( v2 ) {
110
111
            case "1":
                console.log( "v2 vale \"1\"" );
112
113
            case "2":
114
                console.log( "v2 vale \"2\"" );
115
                break;
116
            default:
117
                console.log( "v2 vale alguma coisa..." );
118
119
        }
120
```

```
121
122 }
```

Para mais detalhes sobre os operadores em JavaScript, acesse https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators.

7.5 Estruturas de Repetição e Arrays

Os Arrays em JavaScript atuam como arrays na linguagem Java e C, sendo indexados iniciando em 0, mas podendo crescer ou diminuir quando necessário, assemelhandose mais com listas do que arrays de tamanho fixo. Podemos usar a notação de colchetes para "simular" arrays associativos (tabelas de símbolos), mas de fato o que acontece é que estamos criando ou lendo propriedades do objeto do array. Na Listagem 7.9 pode se ver a criação de três arrays e a utilização da estrutura de repetição for para iterar por seus elementos.

```
Listagem 7.9: Exemplo 04
  Arquivo: /js/exemplo04.js
  function executarExemplo04( event ) {
2
       // um array de uma dimensão
3
       let a1 = [1, 2, 3, 4];
4
5
       // um array de arrays
6
       let a2 = [ [ 1, 2 ], [ 3, 4 ] ];
7
8
       // um array vazio
9
       let a3 = [];
10
       a3["a"] = 2; // simulação de array associativo!
11
       a3["b"] = 4; // análogo à uma tabela de símbolos
12
       a3["c"] = 6; // mas as propriedades são inseridas
13
       a3["d"] = 8; // no objeto!
15
       // arrays em JavaScript podem crescer e diminuir
16
       // livremente usando os métodos:
17
               push (insere no fim)
       //
18
```

```
//
                pop (remove do fim)
19
       //
            unshift (insere no início)
20
       //
             shift (remove do início)
21
             splice (remove de uma posição fornecida)
22
23
       for ( let i = 0; i < a1.length; i++ ) {
24
           console.log( a1[${i}] = ${a1[i]} );
       }
26
27
       // iterando usando o método forEach do
28
       // objeto Array
29
       a1.forEach( function( valor, indice ) {
30
           console.log( `a1[${indice}] = ${valor}` );
31
       });
32
33
       for ( let i = 0; i < a2.length; i++ ) {
34
           console.log( a2[${i}] = ${a2[i]});
35
       }
36
37
       // notação de closure
38
       a2.forEach( ( valor, indice ) => {
39
           console.log( `a2[${indice}] = ${valor}` );
40
       });
41
42
       // a3 não tem elementos, pois o usamos como um
43
       // "array associativo"
44
       for ( let i = 0; i < a3.length; i++ ) {
45
           // não entra aqui...
46
           console.log( a3[${i}] = ${a3[i]} );
47
       }
48
49
       // e agora?
50
       a3.forEach( valor => {
51
           // não entra aqui também
52
           console.log( valor );
53
       });
54
55
       // assim funciona, pois iteramos pelas
56
       // propriedades do objeto
57
       for ( let chave in a3 ) {
58
           console.log(`a3[\{chave\}] = \{a3[chave]\}`);
59
       }
60
```

```
61
       // ou
62
       Object.keys( a3 ).forEach( chave => {
63
           console.log( a3[\{chave\}] = \{a3[chave]\});
64
       });
65
       // métodos de iteração every e some.
68
69
       // o método forEach não foi projetada parar
70
       // parar no meio da execução. para isso, existem
71
72
       // outras duas funções análogas que podem ser
       // usadas para esse propósito.
73
74
       // some => algum/alguns. a ideia é processar
75
       // alguns elementos do array até que uma condição seja
76
       // alcançada. o retorno true da função de callback faz
77
       // a iteração parar e falso continuar para o próximo
78
       // elemento.
       let algum;
80
       console.log( "há algum valor maior que 10?" );
81
       algum = a1.some( maiorQue10 );
82
       console.log( algum ? "sim" : "não" );
83
84
       console.log( "há algum valor menor que 10?" );
       algum = a1.some( menorQue10 );
       console.log( algum ? "sim" : "não" );
87
88
       // every => todos. a ideia é processar
89
       // todos elementos do array verificando se todos
       // passam por uma condição especificada na função
       // de callback. o retorno true faz a função continuar
92
       // para o próximo elemento, enquanto false a faz parar.
93
       let todos;
94
       console.log( "todos são maiores que 10?" );
95
       todos = a1.every( maiorQue10 );
       console.log( todos ? "sim" : "não" );
98
       console.log( "todos são menores que 10?" );
99
       todos = a1.every( menorQue10 );
100
       console.log( todos ? "sim" : "não" );
101
102
```

```
// while e do while são análogos a C, C++, Java etc.
103
104
105 }
106
  // callbacks para testes das funções some e every
107
   function maiorQue10( valor ) {
        console.log( valor );
109
        return valor > 10;
110
111 }
112
function menorQue10( valor ) {
114
        console.log( valor );
        return valor < 10;</pre>
115
116 }
```

Na linha 4 um array de uma dimensão é criado e inicializado com os valores 1, 2, 3 e 4, contidos respectivamente nas posições 0, 1, 2 e 3. Na linha 7 é criado um array em que nas posições 0 e 1 estão contidos dois outros arrays, um com os valores 1 e 2 e o outro com os valores 3 e 4.

Na linha 10 um novo array vazio é criado e entre as linhas 11 e 14 são inseridas quatro propriedades no objeto em si. Note que essas propriedades são criadas no objeto e pela sintaxe usada, há a impressão que estamos lidando com o array como se fosse um array associativo ou uma tabela de símbolos, mapa ou dicionário, mas não é o caso! Podemos usar a notação de colchetes para lidar com objetos para, por exemplo, acessar propriedades ou atributos que contenham espaço nos nomes. Note que posteriormente, ao se tentar iterar por esse array usando um for "normal", não se entrará no bloco da estrutura de repetição, visto que, apesar de parecer que o array contém quatro elementos, na verdade ele não tem nenhum, fazendo com que o atributo length valha zero.

A inserção e remoção de elementos dos arrays podem ser feitas usando os métodos push que insere um elemento no fim do array, pop que remove um elemento do fim, unshift que insere um elemento no início, shift que remove do início e splice que remove de uma posição arbitrária. Na caixa "Saiba Mais" abaixo você pode dar uma olhada na documentação do objeto Array da linguagem JavaScript.

(i) | Saiba Mais

Para mais detalhes sobre o tipo Array em JavaScript, acesse https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array.

Entre as linhas 24 e 26 usa-se um for para iterar pelos elementos do array a1. Entre as linhas 30 e 32 usa-se o método forEach do objeto Array para iterar pelos elementos, usando uma função de *callback* para processar cada posição. A mesma coisa acontece entre as linhas 34 e 41, com a diferença de se usar a notação de *closure* para a definição da função de *callback* utilizada no método forEach de a2.

Como mencionado anteriormente, o uso de um for padrão -ou mesmo um for Each para a não surtirá efeito, pois esse array está vazio! Nós inserimos quatro propriedades no mesmo, não quatro elementos. Quando usamos um array dessa forma, inclusive poderia ser qualquer objeto, e queremos iterar por essas propriedades temos basicamente duas formas: ou fazemos como entre as linhas 58 e 60, usando um for ... in ou obtemos as chaves do objeto como um array e as processamos usando um for Each como mostrado entre as linhas 63 e 65.

Além do método forEach existem alguns outros para o processamento dos elementos de um array. Dois desses métodos são mostrados a partir da linha 80: every e some. Como os nomes sugerem, o primeiro é utilizado com a premissa de testar alguma condição em todos os elementos do array, enquanto o outro espera-se que algum elemento não se enquadre em algo desejado. Podemos utilizar esses métodos para, por exemplo, executar uma busca/pesquisa sequencial/linear no array e parar a iteração quando o elemento for encontrado, retornando um valor verdadeiro ou falso, dependendo da situação e do método empregado. Veja os comentários do exemplo e teste a execução do código para entender exatamente do que se trata.

7.6 "Classes", Objetos e JSON

Antes do ECMAScript 2015 (sexta edição) a criação de objetos com um "tipo" era feito a partir do uso de uma função e o operador new. A partir do ECMAScript 2015 existe uma sintaxe para a definição de tipos abstratos de dados em forma de classes, inclusive permitindo herança entre os tipos definidos. Na Listagem 7.10, entre as linhas 1 e 17 é definida a classe Forma. As classes em JavaScript podem ter apenas um construtor e, dentro dele, as propriedades do objeto devem ser criadas utilizando a palavra chave this.

```
Listagem 7.10: Exemplo 05
Arquivo: /js/exemplo05.js

class Forma {
```

```
// só pode haver um construtor
3
       constructor( xIni, yIni, xFim, yFim ) {
4
           // criação da propriedade
5
           // usando this
6
           this.xIni = xIni;
7
           this.yIni = yIni;
8
           this.xFim = xFim;
9
           this.yFim = yFim;
10
       }
11
12
       calcularArea() {
13
           return 0;
14
15
16
17 }
18
   // "herança"
19
   class Retangulo extends Forma {
21
       // sobrescrevendo a função calcularArea
22
       calcularArea() {
23
           let largura = Math.abs( this.xFim - this.xIni );
24
           let altura = Math.abs( this.yFim - this.yIni );
25
           return largura * altura;
26
       }
27
28
  }
29
30
   class Circulo extends Forma {
31
32
       // novo construtor
33
       constructor( xCentro, yCentro, raio ) {
34
           super( xCentro, yCentro, 0, 0 );
35
           this.raio = raio;
36
       }
37
       calcularArea() {
39
           return Math.PI * this.raio * this.raio;
40
       }
41
42
43 }
44
```

```
function executarExemplo05( event ) {
46
       let r = new Retangulo(0, 0, 10, 20);
47
       let c = new Circulo(5, 10, 10);
48
49
       // um objeto criado usando o inicilizador de objetos
50
       let o = {
           nome: "joão",
52
           sobrenome: "da silva",
53
           endereco: {
54
               logradouro: "rua um",
55
               numero: 100,
56
               cep: "12345-67",
57
               cidade: {
                   nome: "Vargem Grande do Sul",
59
                   estado: {
60
                       nome: "São Paulo",
61
                        sigla: "SP"
62
                   }
               }
64
           }
65
       };
66
67
       // tipos dos objetos
68
       console.log( "Retângulo:", typeof r );
       console.log( "Círculo:", typeof c );
70
       console.log( "Genérico:", typeof o );
71
72
       // são instâncias de?
73
       console.log( "Retângulo é um Objeto?", r instanceof Object );
74
       console.log( "Retângulo é uma Forma?", r instanceof Forma );
75
       console.log( "Retângulo é um Retângulo?", r instanceof Retangulo );
76
       console.log( "Retângulo é um Círculo?", r instanceof Circulo );
77
78
       console.log( "Círculo é um Objeto?", c instanceof Object );
79
       console.log( "Círculo é uma Forma?", c instanceof Forma );
       console.log( "Círculo é um Retângulo?", c instanceof Retangulo );
       console.log( "Círculo é um Círculo?", c instanceof Circulo );
82
83
       console.log( "Genérico é um Objeto?", o instanceof Object );
84
       console.log( "Genérico é uma Forma?", o instanceof Forma );
85
       console.log( "Genérico é um Retângulo?", o instanceof Retangulo );
86
```

```
console.log( "Genérico é um Círculo?", o instanceof Circulo );
87
88
        // uma string com um objeto codificado como
89
        // JavaScript Object Notation (JSON)
90
        let json = '{ "nome": "Maria", "peso": 52.5 }';
91
92
        // converte json para objeto
93
        let json0 = JSON.parse( json );
94
95
        // converte objeto para json (string)
96
        let jsonD = JSON.stringify( o );
97
98
        // prettyprint
99
        let jsonDId = JSON.stringify( o, null, 4 );
100
101
        console.log( r );
102
        console.log( r.calcularArea() );
103
104
        r.xIni = 5;
105
        console.log( r );
106
        console.log( r.calcularArea() );
107
108
109
        console.log( c );
        console.log( c.calcularArea() );
110
111
        c.raio = 5;
112
        console.log( c );
113
114
        console.log( c.calcularArea() );
115
        console.log( o );
116
        console.log( o.nome );
117
        console.log( o[ "sobrenome" ] );
118
119
        // exibindo cada propriedade
120
        for ( var p in o ) {
121
            console.log(^{\circ}.${p} = ${o[p]}^{\circ});
122
        }
123
124
        // usando a função recursiva
125
        processarObjeto( o, "o" );
126
127
        // mostrando as conversões json <-> objeto
128
```

```
console.log( json0 );
129
        processarObjeto( jsonO, "jsonO" );
130
131
        console.log( jsonD );
132
        console.log( jsonDId );
133
134
   }
135
136
   function processarObjeto( obj, nome ) {
137
        for ( var p in obj ) {
138
            if ( typeof obj[p] === 'object' && obj[p] !== null ) {
139
140
                 processarObjeto( obj[p], p );
            } else {
141
                 console.log( `${nome}.${p} = ${obj[p]}` );
142
            }
143
        }
144
   }
145
```

Entre as linhas 20 e 43 são declaradas as classes Retangulo e Circulo que herdam de Forma, sobrescrevendo o método calcularArea(). Nas linhas 47 e 48 cria-se respectivamente uma instância de Retangulo e uma de Circulo. Entre as linhas 51 e 66 cria-se um novo objeto genérico, usando o inicializador de objetos (abre e fecha chaves). Note que esse objeto e os outros dois tem como tipo object (linhas 69 a 71), mas é possível verificar se são instância de uma determinada classe/tipo usando o operador instanceof como visto entre as linhas 74 e 87.

Podemos também representar objetos inteiros como Strings usando a notação de objetos JavaScript, chamada de JSON. Veja na linha 91 onde temos uma String codificando um objeto usando algo similar à notação de inicilização de objetos. Isso é o JSON. Para transformar essa String em um objeto, usa-se o método parse() do objeto global JSON e, quando se quer o contrário, ou seja, transformar um objeto em uma String no formato JSON, usa-se o método stringfy() também do objeto JSON. O formato JSON é amplamanete utilizado atualmente em detrimento do formado XML, pois é mais enxuto, ou seja, é necessário menos texto para codificar o mesmo objeto em JSON do que em XML. Tanto o formato JSON quanto XML são utilizados para a serialização de objetos, ou seja, transformar uma representação binária, específica de linguagem, em uma representação serial fácil de ser processada e independente de plataforma. O processo de converter um objeto para um formato de intercâmbio de dados é chamado de serialização, enquanto o processo inverso, ou seja, a partir de um formato de intercâmbio de dados gerar um objeto específico de tecnologia é chamado de desserialização.

O restante do código do exemplo é facilmente entendido ao ser lido. Nas próximas caixas "Saiba Mais" há varios links úteis sobre o tema.

(i) | Saiba Mais

Para saber mais sobre classes em JavaScript, acesse https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes.

(i) | Saiba Mais

Para consultar a documentação sobre o tipo String em JavaScript, acesse https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String.

(i) | Saiba Mais

Para consultar a documentação sobre JSON em JavaScript, acesse https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON.

7.7 Manipulação do DOM

Nesta Seção aprenderemos a manipular o DOM com objetivo de extrair dados do documento ou então modificá-lo em tempo de execução. Isso já foi comentado brevemente.

(i) | Saiba Mais

A documentação do DOM pode ser vista no *link* https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model.

Vamos manipular o DOM de duas formas, uma usando JavaScript puro, que normalmente é mais verboso, e a outra usando a biblioteca jQuery⁴, que já foi padrão na construção de aplicações para Web e tem caído em desuso, mas ainda é relevante, principalmente por facilitar algumas coisas e ainda estar presente de forma consistente em diversas aplicações criadas e que você provavelmente terá que dar manutenção na sua vida profissional.

⁴<https://jquery.com/>

7.7.1 JavaScript Puro

A ideia desse exemplo é que ao se clicar no botão, um novo nó da tag seja inserido na <math><div> que tem id igual à divExemplo06, usando um contador para mostrar as sucessivas inserções. Na linha 6 da Listagem 7.11 obtém-se tal <div> utilizando o método getElementById("id") do objeto document. Caso bem-sucedido, o método retornará uma referência ao nó dessa <div> no DOM e então poderemos manipulá-lo! Na linha 9 criamos um novo nó para inserir na <div> do tipo correspondente à tag . Entre as linhas 12 e 13 inserimos o conteúdo do parágrafo criado e defimos qual é sua classe. Veja no arquivo estilos. css que temos a definição de um seletor de classe chamado <math>pdotes que refletirá na inserção desses parágrafos, sendo que todo item par terá uma cor de fundo azulada. Por fim, na linha 16 esse novo parágrafo é inserido na <div> e essa alteração é prontamente refletida na renderização do documento! Pare e pense agora todas as possibilidades existentes!

```
Listagem 7.11: Exemplo 06
   Arquivo: /js/exemplo06.js
  var contadorExemplo06 = 1;
  function executarExemplo06( event ) {
       // obtém o elemento pelo identificador
       let div = document.getElementById( "divExemplo06" );
6
7
       // cria um novo elemento do tipo p (tag )
8
       let p = document.createElement( "p" );
9
10
11
       // configura os atributos
       p.innerHTML = `JS Puro - Contador: ${contadorExemplo06++}`;
12
       p.className = "pDOM";
13
14
       // adiciona na div
15
       div.append( p );
16
17
  }
18
```

7.7.2 Usando jQuery

Talvez os dois principais diferenciais ou chamarizes da jQuery que a tornaram famosa é a utilização da sintaxe de seletores do CSS para obter elementos do DOM, o que já foi implementado de forma nativa no JavaScript através de função querySelector do objeto document e a facilidade para lidar com requisições assíncronas com o servidor, o que também já foi endereçado nas versões mais novas do JavaScript.

No exemplo apresentado na Listagem 7.12 temos algo análogo ao exemplo anterior, só que usando as funcionalidades dessa biblioteca. Na linha 7 obtemos a <div> que tem divExemplo07 como id usando a notação de seletores do CSS! As funcionalidades da jQuery são acessadas através do símbolo de cifrão (\$) que é um *alias* (apelido) para o objeto chamado jQuery, disponível para ser utilizado nos *scripts* a partir do momento em que a biblioteca é inserida no documento. A criação do parágrafo também é mais direta, bastando inserir o par de *tags* como String no argumento da função \$ e então encadear a invocação dos métodos html e addClass. A linha 16 é idêntica ao exemplo anterior.

```
Listagem 7.12: Exemplo 07
  Arquivo: /js/exemplo07.js
  var contadorExemplo07 = 1;
2
  function executarExemplo07( event ) {
3
4
      // seleciona a div com id divExemplo07
5
      // a jQuery usa a sintaxe os seletores do CSS
6
      let div = $( "#divExemplo07" );
7
8
      // cria um novo elemento do tipo p (tag )
9
      // e configura os atributos encadeando a chamada de métodos
10
      let p = ("")
11
               .html( `jQuery - Contador: ${contadorExemplo07++}` )
12
               .addClass( "pDOM" );
13
14
       // adiciona na div
15
      div.append( p );
16
17
18 }
```

Se quiser aprender um pouco mais sobre a jQuery, acesse https://learn.jquery.com/.

(i) | Saiba Mais

Sobre a função (querySelector), acesse https://developer.mozilla.org/pt-B R/docs/Web/API/Document/querySelector>.

7.8 Manipulação de Formulários

A ideia central presente neste Capítulo é fazer com que você entenda um pouco sobre JavaScript para podermos ter formulários mais sofisticados, permitindo a construção de cadastros que envolvam relacionamos muitos-para-muitos. Claro que estamos vendo várias coisas a mais, mas a ideia é dar subsídios para implementarmos coisas novas no projeto iniciado no Capítulo 5. No exemplo apresentado na Listagem 7.13 veremos a obtenção e a inserção de dados em componentes de formulários. Detalharei alguns pontos do código e, para não ficar muito maçante, o restante é com você. Tenho certeza que entenderá o que está acontecendo com base no que foi visto até agora.

```
Listagem 7.13: Exemplo 08
   Arquivo: /js/exemplo08.js
  let contadorOpSelect03 = 4;
  let contadorOpSelect04 = 4;
  function lerDadosFormulario( event ) {
       // obtém os dados do formulário usando
       // JavaCcript puro
7
8
       // obtém por id (não deve haver mais de um!)
9
       // e pegar a propriedade value
10
       let campo01 = document.getElementById( "campo01" ).value;
11
       // obtém pelo atributo name (pode haver mais de um!)
13
       // e na seleção, usa o primeiro elemento do resultado
14
      let campo02 = document.getElementsByName( "campo02" )[0].value;
15
16
```

```
// por id
17
       let select03 = document.getElementById( "select03" ).value;
18
       let select04 = document.getElementById( "select04" ).value;
19
       let area05 = document.getElementById( "area05" ).value;
20
21
       alert(
22
              `Campo 01: {campo01}\n +
23
              Campo 02: {campo 02} n +
24
              `Select 03: ${select03}\n` +
25
              `Select 04: \{select04\}\n` +
26
              `Área 05: ${area05}`);
27
28
   }
29
30
   // funções podem ser atribuídas à variáveis!
31
   var lerDadosFormularioJQuery = function( event ) {
32
33
       // com a jQuery, usa-se a sintaxe igual
34
       // aos seletores do CSS
36
       // a função val() retornará o valor
37
       // do componente de forma padronizada
38
       let campo01 = $( "#campo01" ).val();
39
       let campo02 = $( "#campo02" ).val();
40
       let select03 = $( "#select03" ).val();
41
       let select04 = $( "#select04" ).val();
42
       let area05 = $( "#area05" ).val();
43
44
       alert(
45
               Campo 01: {campo01}\n +
46
              `Campo 02: {campo02}\n +
47
              `Select 03: {select03}\n +
48
              Select 04: {select04}\n` +
49
              `Área 05: ${area05}`);
50
51
   }; // termina com ponto e vírgula
   // pode-se usar a sintaxe de closures
   let inserirDadosFormulario = event => {
55
56
       // configurando os valores
57
       document.getElementById( "campo01" ).value = "campo 01 atualizado";
58
```

```
document.getElementsByName( "campo02" )[0].value = "campo 02 também";
59
       document.getElementById( "select03" ).value = "o2";
60
       document.getElementById( "select04" ).value = "o3";
61
       document.getElementById( "area05" ).value = "outro valor";
62
63
   }; // termina com ponto e vírgula
   function inserirDadosFormularioJQuery( event ) {
67
       // configura os valores usando a função val()
68
       $( "#campo01" ).val( "novo valor campo 01" );
69
       $( "#campo02" ).val( "outro novo valor... ");
70
       $( "#select03" ).val( "o3");
71
       $( "#select04" ).val( "o1" );
72
       $( "#area05" ).val( "mudando o valor da área" );
73
74
75
   }
76
   function inserirNovaOpcao( event ) {
77
78
       // cria um elemento do tipo option (tag <option>)
79
       let op = document.createElement( "option" );
80
81
       // configura
82
       op.innerHTML = `Opção ${contadorOpSelect03}j`;
       op.value = `o${contadorOpSelect03}j`;
85
       // obtém o select e adiciona o op
86
       document.getElementById( "select03" ).add( op );
87
88
89
       contadorOpSelect03++;
90
   }
91
92
   function inserirNovaOpcaoJQuery( event ) {
93
       // com jQuery é um pouco mais limpo
       let op = $( "<option></option>" )
96
                .html( `Opção ${contadorOpSelect04}jq`)
97
                .val( `o${contadorOpSelect04}jq` );
98
99
       $( "#select04" ).append( op );
100
```

```
101
102 contadorOpSelect04++;
103
104 }
```

Neste exemplo são definidas seis funções que tratarão o evento *click* de seis botões. A ideia é apresentar três situações de duas formas diferentes, uma com JavaScript puro e uma usando jQuery. Na linha 11 da primeira função, lerDadosFormulario(event), obtemos o valor do campo de texto identificado por campo01 em seu id. Veja que obtemos o elemento pelo id (método getElementById("id")), acessamos a propriedade value e atribuímos à variável campo01. Podemos também obter componentes pelo atributo name, entretanto pode haver mais de um componente com o mesmo name, então o método getElementsByName("name") retorna um array. No nosso exemplo há apenas um componente com o nome campo02, mas como o retorno do método getElementsByName("name") é um array, precisamos obter o primeiro elemento do array retornado e então obter o valor.

Entre as linhas 18 e 20 obtemos o valor de dois componentes <select> e um
<textarea>. Note que para os *selects* o valor retornado será o da opção selecionada no momento. Por fim, entre as linhas 22 e 27, apresentamos os valores lidos usando um alerta.

A segunda função implementada, <code>lerDadosFormularioJQuery(event)</code>, faz a mesma coisa que a primeira, entretanto usando jQuery. Perceba que o código fica mais limpo e enxuto. Basta usar o seletor de id do CSS para obter o componente desejado e usar o método <code>val()</code> para obter o valor.

O segundo conjunto de funções, inserirDadosFormulario(event) e inserirDadosFormularioJQuery(event), faz o processo inverso, ou seja, ao invés de ler os dados para fazer alguma coisa, os dados são inseridos nos componentes. Perceba que a definição da função inserirDadosFormulario(event) é feita usando a notação de *closures*, atribuindo a função à variável inserirDadosFormulario.

Por fim, na última dupla de funções, [inserirNovaOpcao(event)] e [inserirNovaOpcaoJQuery(event)] temos o exemplo de criar uma nova opção para os componentes *select*. Isso será útil no nosso próximo projeto.

7.9. EVENTOS 223

7.9 Eventos

Quase todas as *tags* HTML disparam uma vasta quantidade de tipos de eventos que podem ser tratados. Claro que a maioria dos eventos "úteis" são ouvidos em componentes visíveis, mas há inúmeras possibilidades. Na Listagem 7.14 é apresentado como se faz programaticamente o registro de tratadores de eventos em nós do DOM ao invés de fazer isso direto no HTML como estamos fazendo com os botões.

```
Listagem 7.14: Exemplo 09
  Arquivo: /js/exemplo09.js
  function registrarEventosExemplo09() {
2
3
       // JavaScript puro
       document.getElementById( "campoExemplo09" ).onkeydown = event => {
4
           console.log( `Digitou '${event.key}'` );
5
6
       };
7
       // jQuery
       $( "#selectExemplo09" ).on( "change", function( event ) {
9
           // nesse conexto, this é a mesma coisa que event.target
10
           // ou seja, o elemento em que o evento foi disparado
11
           let select = $( this );
12
           alert( `Valor: ${select.val()}` );
13
       });
14
15
  }
16
```

É importante perceber que essa função precisa ser executada para que os tratadores de eventos sejam registrados de fato, sendo assim, na linha 171 da Listagem 7.1, essa função é invocada dentro da *tag* (script). Normalmente quando se quer registrar eventos programaticamente no documento isso é feito quando todo o documento está pronto para ser processado, ou seja, todo o DOM foi construído. Veremos isso na prática no Capítulo 8.

Usando JavaScript puro, o registro de eventos é feito associando às propriedades prefixadas com on dos nós do DOM uma função para o tratamento dos mesmos. Na linha 4 da Listagem 7.14 pode-se ver a obtenção de um campo de texto e o registro do tratador de evento para o evento "key down" (tecla pressionada) que mostrará, via console.log(...), o caractere digitado no campo de texto.

Já na linha 9 fazemos o registro do evento *change* de um *select* usando jQuery. Veja como a sintaxe é diferente. O evento *change* é disparado quando se seleciona uma

opção do *select* em questão, desde que a opção seja diferente da opção selecionada no momento. No nosso caso, o valor da opção selecionada será mostrada usando um alerta.

Na caixa "Saiba Mais" abaixo você poderá conferir uma lista completa de todos os eventos que existem e podem ser usados.

(i) | Saiba Mais

A documentação completa sobre os eventos que podem ser tratados pode ser vista em https://developer.mozilla.org/en-US/docs/Web/Events.

7.10 Simulação Usando a Canvas API

Nesta Seção falaremos um pouco sobre o *tag* <anvas> e sobre a Canvas API, que nos permitirá realizar operações de desenho em Duas Dimensões (2D). Atualmente, além de trabalhar com desenhos em duas dimensões, podemos também usar a WebGL API para realizar desenhos em 2D ou Três Dimensões (3D) usando aceleração em *hardware*, mas isso foge demais do escopo deste livro. Focaremos no 2D realizando a simulação da física de "bolinhas" que serão animadas para que você possa conhecer um pouco sobre a Canvas API e sobre a utilização da função setInterval(...) que nos permetirá executar código repetidamente em um intervalo predefinido de tempo. Na Listagem 7.15 podemos ver o código completo do exemplo.

```
Listagem 7.15: Exemplo 10
  Arquivo: /js/exemplo10.js
  function prepararCanvasExemplo10() {
2
3
       class Bolinha {
4
           // x e y são os centros da bolinha
5
           constructor( x, y, velocidadeX, velocidadeY, raio, cor ) {
6
7
               this.x = x;
8
               this.y = y;
9
               this.raio = raio;
               this.cor = cor;
10
               this.velocidadeX = velocidadeX;
11
               this.velocidadeY = velocidadeY;
12
               this.elasticidade = 0.9; // elasticidade do material
13
                                        // atrito do material com o meio
               this.atrito = 0.99;
14
```

```
this.emArraste = false; // está sendo arrastada?
15
           }
16
17
           desenhar( ctx ) {
18
19
               // cor usada pelo contexto gráfico
               // para realizar o desenho
               ctx.fillStyle = this.cor;
               // inciando um caminho
24
               ctx.beginPath();
25
               // realizando um arco de uma volta (círculo)
               // no caminho iniciado
               ctx.arc( this.x, this.y, this.raio, 0, 2 * Math.PI );
29
               // fechando e preenchendo o caminho iniciado
31
               // com a cor definida acima
               ctx.fill();
34
           }
           // move a bolinha em cada passo da simulação
37
           mover() {
               // se a bolinha não estiver sendo arrastada pelo mouse
               if (!this.emArraste) {
41
42
                   // recalcula a posição baseando-se
43
                   // nas velocidades
                   this.x += this.velocidadeX;
                   this.y += this.velocidadeY;
46
47
                   // se a bolinha passou da "parede" direita do <canvas>
48
                   if ( this.x + this.raio > larguraCanvas ) {
                       // reposiciona a bolinha sem passar da fronteira
                       this.x = larguraCanvas - this.raio;
53
                       // recalcula a velocidade em x, trocando a direção
54
                       // e aplicando a elasticidade
55
                       this.velocidadeX = -this.velocidadeX * this.elasticidade;
```

```
57
                    }
58
59
                    // se a bolinha passou da "parede" esquerda do <canvas>
60
                    if ( this.x - this.raio < 0 ) {</pre>
61
62
                        // reposiciona a bolinha sem passar da fronteira
63
                        this.x = this.raio;
64
65
                        // recalcula a velocidade em x, trocando a direção
66
                        // e aplicando a elasticidade
67
                        this.velocidadeX = -this.velocidadeX * this.elasticidade;
68
69
                    }
70
71
                    // se a bolinha passou do "chão" do <canvas>
72
                    if ( this.y + this.raio > alturaCanvas ) {
73
74
                        // reposiciona a bolinha sem passar da fronteira
75
                        this.y = alturaCanvas - this.raio;
76
77
                        // recalcula a velocidade em y, trocando a direção
78
                        // e aplicando a elasticidade
79
                        this.velocidadeY = -this.velocidadeY * this.elasticidade;
80
81
                    }
82
83
                    // se a bolinha passou do "teto" do <canvas>
84
                    if ( this.y - this.raio < 0 ) {</pre>
85
86
                        // reposiciona a bolinha sem passar da fronteira
87
                        this.y = this.raio;
88
89
                        // recalcula a velocidade em y, trocando a direção
90
                        // e aplicando a elasticidade
91
                        this.velocidadeY = -this.velocidadeY * this.elasticidade;
92
93
                    }
94
95
                    // aplica o atrito
96
                    this.velocidadeX *= this.atrito;
97
                    this.velocidadeY *= this.atrito;
98
```

```
99
                     // aplica a gravidade
100
                     this.velocidadeY += gravidade;
101
102
                 }
103
104
            }
105
106
            // a coordenada x, y está dentro da bolinha
107
108
            intercepta( x, y ) {
109
                 // pitágoras ;)
110
                return Math.hypot( this.x - x, this.y - y ) <= this.raio;</pre>
111
112
            }
113
114
            // cria uma nova velocidade aleatória para a bolinha
115
            gerarNovaVelocidade() {
116
                this.velocidadeX = gerarValorAleatorio( -30, 30 );
117
                 this.velocidadeY = gerarValorAleatorio( -30, 30 );
118
            }
119
120
        }
121
122
        // obtém o canvas que será usado
123
        let canvas = document.getElementById( "canvasExemplo10" );
124
125
126
        // a partir do canvas obtido, requisita um contexto
        // gráfico 2D
127
        let context = canvas.getContext( "2d" );
128
129
        // variáveis para largura e altura do <canvas>
130
131
        // para facilitar o entendimento
        let larguraCanvas = canvas.width;
132
        let alturaCanvas = canvas.height;
133
134
        // diferenças em x e y no clique para ajuste
135
        // de posição durante o arraste
136
        let dx;
137
        let dy;
138
139
        // posições antigas em x e y para recálculo das
140
```

```
// velocidades durante o arraste
141
       let xAntigo;
142
       let yAntigo;
143
144
       // gravidade
145
       let gravidade = 1;
146
147
        // cria uma bolinha
148
        let bolinha = new Bolinha(
149
                larguraCanvas / 2,
150
                alturaCanvas / 2,
151
                2.0,
152
                2.0,
153
                10,
154
                "rgb(0,0,0)");
155
156
        // qual bolinha está sendo arrastada?
157
       let bolinhaEmArraste = null;
158
159
       // cria um array de bolinhas
160
       let bolinhas = [ bolinha ];
161
162
       // "engine/motor" da simulação
163
       // a função passada será executada a cada
164
       // 10 milisegundos, ou seja cada segundo terá
165
       // aproximadamente 100 quadros de animação
166
       setInterval( () => {
167
168
            // limpa os desenhos anteriores
169
            context.clearRect( 0, 0, larguraCanvas, alturaCanvas );
170
171
            // realiza a movimentação e o desenho de cada
172
173
            // bolinha
            bolinhas.forEach( bolinha => {
174
                bolinha.mover();
175
                bolinha.desenhar( context );
176
           });
177
178
       }, 10);
179
180
181
        /*********
182
```

```
interação com o usuário
183
         ***********
184
185
        // quando algum botão do mouse for pressionado no <canvas>
186
        canvas.onmousedown = event => {
187
188
            // se for o botão esquerdo
189
            if ( event.button === 0 ) {
190
191
                // verifica se a posição que foi clicada
192
                // está em cima de alguma bolinha
193
                // percorre-se o array ao contrário para dar
194
                // prioridade às bolinhas que estão em cima
195
                // das outras
196
                for ( let i = bolinhas.length - 1; i >= 0; i-- ) {
197
198
                    // uma das bolinhas
199
                    let bolinha = bolinhas[i];
200
201
                    // interceptou?
202
                    if ( bolinha.intercepta( event.offsetX, event.offsetY ) ) {
203
204
205
                         // a bolinha atual está em arraste então
                        bolinha.emArraste = true;
206
207
                         // obtém a diferença da posição do clique
208
                         // com o centro da bolinha
209
210
                         // isso é importante para que se dê a ideia
                         // que a bolinha ficou "colada" no cursor
211
                         // na posição correta
212
                        dx = event.offsetX - bolinha.x;
213
                        dy = event.offsetY - bolinha.y;
214
215
                         // sabemos agora qual bolinha está em arraste
216
                         bolinhaEmArraste = bolinha;
217
218
219
                        break;
220
                    }
221
222
                }
223
224
```

```
// não há bolinha sendo arrasta
225
                 if ( bolinhaEmArraste === null ) {
226
227
                     // criamos uma nova bolinha
228
                     let novaBolinha = new Bolinha(
229
                              event.offsetX,
                              event.offsetY,
231
                              gerarValorAleatorio( -3, 3 ),
232
                              2.0,
233
                              5 + Math.random() * 10,
234
                              gerarCorAleatoria() );
235
236
                     // inserimos no array
237
                     bolinhas.push( novaBolinha );
238
239
                }
240
241
                // outro botão que não o esquerdo
            } else {
243
244
                // percorre o array e gera novas velocidades
245
                // para todas as bolinhas, dando a impressão
246
                 // "chacoalhar" o <canvas>
247
                bolinhas.forEach( bolinha => {
248
                     bolinha.gerarNovaVelocidade();
249
                });
250
251
            }
252
253
        };
254
255
        // o botão clicado no <canvas> foi solto
256
257
        canvas.onmouseup = event => {
258
            // há bolinha em arraste?
            if ( bolinhaEmArraste !== null ) {
260
261
                 // não está mais sendo arrastada!
262
                bolinhaEmArraste.emArraste = false;
263
                bolinhaEmArraste = null;
264
265
            }
266
```

```
267
        };
268
269
        // o cursor do mouse saiu do <canvas>
270
        canvas.onmouseout = event => {
271
272
            // se houver bolinha em arraste, solte-a
273
            if ( bolinhaEmArraste !== null ) {
274
                bolinhaEmArraste.emArraste = false;
275
                bolinhaEmArraste = null;
276
            }
277
278
        };
279
280
        // o cursor do mouse moveu dentro do canvas
281
        canvas.onmousemove = event => {
282
283
            // há bolinha em arraste?
284
            if ( bolinhaEmArraste !== null ) {
285
286
                // obtém as coordenadas anteriores
287
                xAntigo = bolinhaEmArraste.x;
288
289
                yAntigo = bolinhaEmArraste.y;
290
                // reposiciona a bolinha de acordo com
291
                // a difença calculada no clique/seleção
292
                bolinhaEmArraste.x = event.offsetX - dx;
293
294
                bolinhaEmArraste.y = event.offsetY - dy;
295
                // recalcula as velocidades para
296
                // quando essa bolinha for solta
297
                // dando a impressão de arremesso
298
299
                bolinhaEmArraste.velocidadeX = (bolinhaEmArraste.x - xAntigo) / 2;
                bolinhaEmArraste.velocidadeY = (bolinhaEmArraste.y - yAntigo) / 2;
300
301
            }
302
303
        };
304
305
        // esconde o menu de contexto no clique
306
        // com o botão direito
307
        canvas.oncontextmenu = event => {
308
```

```
// não realiza a ação padrão
309
            // nesse caso, não exibir o menu de contexto
310
            event.preventDefault();
311
        };
312
313
314 }
315
   // cria uma cor aleatória
316
   function gerarCorAleatoria() {
317
318
        // um valor inteiro no intervalo [0, 255]
319
        // para cada componente da tríade luminosa
320
        // vermelho (r), verde (q) a azul (b)
        let r = Math.trunc( Math.random() * 256 );
322
        let g = Math.trunc( Math.random() * 256 );
323
        let b = Math.trunc( Math.random() * 256 );
324
325
        // retorna uma string que representa tal cor
        return `rgb(${r},${g},${b})`;
327
328
329 }
330
   // gera um valor aleatório no intervalo [minimo, maximo]
331
   function gerarValorAleatorio( minimo, maximo ) {
        return minimo + Math.random() * ( maximo - minimo );
333
334 }
```

Entre as linhas 3 e 121 definimos uma classe chamada Bolinha. A ideia é que essa classe modele uma bolinha, na verdade um círculo pequeno, que estará suscetível à forças físicas que simularemos. O construtor da classe, iniciado na linha 6, possui seis parâmetros:

- x: é a posição do centro da bolinha no eixo x;
- (y): é a posição do centro da bolinha no eixo *y*;
- velocidadeX: é a quantidade de *pixels* que a bolinha vai ser movida no eixo *x* a cada passo da simulação;
- velocidadeY: é a quantidade de *pixels* que a bolinha vai ser movida no eixo *y* a cada passo da simulação;
- raio: representa o raio do círculo;
- cor : é a cor que será utilizada para desenhar a bolinha.

Ao construir a bolinha esses valores serão usados para inicializar os atributos corres-

pondentes e, além desses, outros atributos serão criados com valores fixos:

- elasticidade: representa algo como o coeficiente de elasticidade do "material" da bolinha. Na nossa simulação ele será fixo para todas as bolinhas criadas e será usado quando uma bolinha bater em alguma parede;
- (atrito): é a tentativa de simular o coeficiente de atrito do material da bolinha com o meio em que ela está imersa;
- (emArraste): indica se a bolinha está sendo arrastada na simulação, ou seja, se o usuário a "pegou" com o mouse e a está arrastando no (<canvas>).

Entre as linhas 18 e 35 é definido o método (desenhar (ctx)) da bolinha. Esse método deve receber como parâmetro um objeto do tipo CanvasRenderingContext2D que será o responsável em realizar o desenho propriamente dito. Uma analogia que podemos fazer é que esse objeto atua como uma caneta super poderosa que pode trocar de cor, de traço etc. Chamaremos esse objeto de "contexto gráfico". Na linha 22 é informado ao contexto gráfico a cor que deve ser usada, ou seja, a cor definida na construção da bolinha. Precisamos desenhar um círculo para representar nossa bolinha, mas não há um método específico para círculos no contexto gráfico da Canvas API. Para fazer isso, inicia-se um caminho na linha 25 e, na linha 29, usamos o método arc(...) que é capaz de desenhar arcos. Esse método recebe a coordenada onde o arco deve começar, no nosso caso é representada pelo centro da bolinha, ou seja, o par ordenado $\{x, y\}$ em conjunto com o raio da mesma e quantos radianos devem ser usados no início e no fim da traçagem do arco. No nosso caso iniciamos em 0 e vamos até 2π que representa uma volta completa em torno de $\{x, y\}$, fazendo assim o círculo que precisamos. Na linha 33 o contexto gráfico é informado que o caminho que foi iniciado na linha 25 deve ser fechado e preenchido com a cor definida.

Na linha 38 temos a definição do método mover() que será responsável em atualizar o estado da bolinha durante a simulação, trocando sua posição de acordo com as velocidades em x e y, detectando a colisão com as "paredes" do <canvas> a aplicando as forças que já citamos, além da gravidade, que será um valor global à simulação. Esse método será executado a cada passo da simulação. Logo veremos isso acontecendo. A implementação desse método é relativamente fácil de entender. Nas linhas 45 e 46 a posição da bolinha é incrementada com base nas velocidades. Ou seja, se a bolinha está na posição $\{50,60\}^5$ e supondo que a velocidade no momento desse incremento é de 5 em x e -2 em y, após o incremento a bolinha estará na posição $\{55,58\}$. Perceba que esse cálculo e as próximas verificações serão feitas apenas se a bolinha não estiver sendo arrastada pelo mouse, pois se o estiver, a posição da bolinha dependerá do

 $^{^5}$ Costumeiramente em APIs gráficas, o sistema de coordenadas cartesianas em um plano é centralizado no canto superior esquerdo dos componentes gráficos, ou seja, a origem de ambos os eixos, o ponto $\{0,0\}$, está situado nesse canto, com o eixo x crescendo para a direita e o eixo y crescendo para baixo, ou seja, para o eixo y temos o contrário do que estamos acostumados na matemática.

cursor e não da simulação física. Essa verificação é feita na linha 41. Entre as linhas 49 e 94 são feitas as verificações se a bolinha passou os limites/"paredes" do <anvas> e, caso tenha, precisa ser reposicionada para não "sumir", ou seja, sair dos limites do <anvas> canvas> além disso, quando a bolinha bater na parede, é necessário incidir a elasticidade do material, pois ela terá que perder momento. Por fim, nas linhas 97 e 98 as velocidades atuais são recalculadas aplicando o atrito e na linha 101 a gravidade será aplicada à velocidade do componente y.

Entre as linhas 108 e 113 é implementado o método (intercepta (x, y)) que retornará (true) caso a coordenada passada nos parâmetros interceptar a bolinha ou false caso contrário, ou seja, se está dentro do círculo que a define. Esse cálculo é relativamente simples, bastando usar o teorema de pitágoras para calcular a distância do centro da bolinha até a coordenada fornecida. Se essa distância, que é a hipotenusa do triângulo retângulo formado por esses dois pontos, considerando que os mesmos são os vértices dos ângulos agudos do triângulo, for menor ou igual ao raio do círculo, quer dizer que a coordenada está dentro do círculo.

O método [gerarNovaVelocidade()] é definido entre as linhas 116 e 119 e gerará uma nova velocidade aleatória para a bolinha quando for invocado, variando de -30 a 30.

Após a definição da classe da bolinha, temos o restante do código da simulação. Note que todo o código está comentado, por isso não ficarei esmiuçando todos os detalhes aqui.

Caso tenha interesse em explorar a Canvas API acesse o link disponível na caixa "Saiba Mais" abaixo.

(i) | Saiba Mais

A API do Canvas pode ser vista em https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API.

7.11 Requisições Assíncronas e Intercâmbio de Dados

Vamos agora lidar com o último tópico deste Capítulo em que trataremos as requisições assíncronas ao servidor. A linguagem JavaScript atualmente possui diversas funcionalidades e construções para trabalharmos com requisições assíncronas e com linhas de execução (*threads*) em segundo plano. Lidaremos com o chamado *Asynchronous JavaScript and XML* (AJAX). Esse nome não se encaixa mais muito bem com o que é feito atualmente, mas mesmo assim continuaremos a utilizá-lo, visto que

se tornou o termo técnico para designar requisições ao servidor que são feitas em segundo plano e que, ao terminarem, são processadas do lado do cliente.

A ideia se baseia em, a partir da execução de código JavaScript, podermos criar um outro canal de comunicação com o servidor, além dos já criados pelo navegador para baixar o código HTML da página, baixar imagens, sincronizar o *stream* de dados de um vídeo que está sendo transmitido pelo servidor e assistido pelos clientes e assim por diante. Essa conversa com o servidor ocorre em segundo plano, ou seja, a invocação do AJAX não é bloqueante como a chamada da função alert que faz o código "parar" de executar naquele ponto até que a função termine. Dada essa característica assíncrona é necessário que esse canal de comunicação seja tratado no futuro, após a chamada do AJAX começar, pois não sabemos quando ela terminará ou mesmo se terminará a contento.

Antigamente, antes da criação da jQuery, o preparo para a comunicação assíncrona com o servidor era um tanto quanto "bagunçado", pois isso não era padronizado entre os navegadores e cada um implementava de uma forma. Como precisamos desenvolver de forma a atender a maior quantidade de navegadores possíveis, era necessário ter um código que verificasse em qual navegador estava rodando e as vezes levando em consideração até a versão do mesmo. Enfim, quase um caos.

Depois que John Resig criou a jQuery e ela foi sendo adotada amplamente pelos desenvolvedores de aplicações Web, essa tarefa se tornou muito mais transparente, pois a biblioteca encapsulava essa questão do tratamento entre navegadores diferentes e esse foi um dos principais fatores para sua popularização. Veremos como fazer requisições AJAX com a jQuery, principalmente pelo fato já informado relacionado à software legado, mas também aprenderemos a fazer a mesma requisição usando a Fetch API que se baseia nas *Promises*, outra novidade do JavaScript "moderno". Uma *Promise* é utilizada justamente na viabilização de rotinas que necessitam de processamento assíncrono e, como o nome diz, se trata de uma promessa, algo deveria acontecer no futuro, mas que pode falhar também. Já falamos disso anteriormente quando definimos o AJAX não é?

Nas próximas caixas "Saiba Mais" você poderá encontrar mais material sobre as APIs de nível mais baixo para a execução de processamento assíncrono.

(i) | Saiba Mais

Mais sobre Promises: https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Promise>

Mais sobre a API Web Workers: https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_APII.

7.11.1 AJAX com jQuery e com Fetch API

Neste exemplo faremos a invocação do Servlet de cálculo de tabuadas de forma assíncrona, utilizando tanto a função ajax(...) da jQuery, quanto a função fetch(...) da Fetch API. Veja que na Listagem 7.16 são definidas duas funções que fazem basicamente a mesma coisa, mudando o meio de execução, pois uma usa jQuery e a outra JavaScript puro.

```
Listagem 7.16: Exemplo 11
   Arquivo: /js/exemplo11.js
  function executarExemplo11jQuery( event ) {
2
       let n = prompt( "Calcular a tabuada de:" );
3
4
       // prepara uma requisição assíncrona usando jQuery
5
       // para a URL "calcularTabuada", enviando o parâmetro
6
       // numero na requisição como atributo do objeto data
7
       // configurado no objeto de opções da função
       $.ajax( "calcularTabuada", {
9
10
           // objeto data
11
           data: {
12
               numero: n
13
           },
14
15
           // atributo dataType do objeto de opções que
16
           // indica qual o tipo de dado que é esperado
17
           // quando a requisição for bem suncedida.
18
           // nesse caso, um retorno de texto puro
19
           dataType: "text"
20
21
           // done associa uma função de callback para
22
           // tratar os dados da requisição caso seja
23
           // bem sucedida
24
       }).done( ( data, textStatus ) =>{
25
```

```
$( "#divExemplo11" ).html( data );
26
27
           // fail é análoga a done, com a diferença
28
           // que lida com problemas na requisição
29
       }).fail( ( jqXHR, textStatus, errorThrown ) => {
30
           alert( "Erro: " + errorThrown + "\n" +
31
                   "Status: " + textStatus );
       });
33
34
  }
35
36
   function executarExemplo11Fetch( event ) {
37
       let n = prompt( "Calcular a tabuada de:" );
39
40
       // cria um objeto do tipo URLSearchParams
41
       // que encapsula os parâmetros enviados
42
       // pela requisição assíncrona da função
43
       // fetch
       let parametros = new URLSearchParams();
45
       parametros.append( "numero", n );
46
47
       // envia uma requisição à URL "calcularTabuada" e passa
48
       // um init object com os atributos method e body
49
       fetch( "calcularTabuada", {
           method: "POST",
51
           body: parametros
52
53
           // se bem sucedido, retorna o texto da resposta
54
       }).then( response => {
           return response.text();
57
           // encadeia com o then anterior, tratando o texto
58
           // retornado
59
       }).then( text => {
60
           $( "#divExemplo11" ).html( text );
62
           // se algum problema ocorrer...
63
       }).catch( error => {
64
           alert( "Erro: " + error );
65
       });
66
67
```

68 }

As duas funções obtém um valor, que se espera que seja um número inteiro, pois não há tratamento, e então o usam como parâmetro em uma requisição ao Servlet mapeado na URL /calcularTabuada. Como o *script* executará no mesmo diretório em que o Servlet está mapeado, não há necessidade de informar o caminho da URL da requisição de forma absoluta.

Como o código está totalmente comentado não ficarei explicando-o detalhadamente, mas a ideia é que quando a requisição for enviada ela será tratada pelas funções de callback apropriadas dependendo do que acontecer. Vejam, é uma promessa do tipo "vou enviar algo para o servidor pedindo algum recurso e esperar que ele me responda algo". Essa resposta pode acontecer em um milésimo de segundo ou em alguns segundos ou mesmo nunca retornar! Então espera-se que no futuro o retorno (ou não retorno!) da requisição seja tratado. O que diferencia as chamadas assíncronas nos dois exemplos é justamente a sintaxe das funções. Ambas têm como primeiro parâmetro a URL que deve ser alcançada e como segundo um objeto de opções, que variará de acordo com a função utilizada. Na a jax (...) criaremos um objeto com as propriedades data, que contém os parâmetros que serão codificados na requisição e dataType que informa ao chamador que tipo de dado será retornado pelo servidor. Já na fetch(...) os parâmetros devem ser encapsulados em um objeto do tipo URLSearchParams e configurados na propriedade body do objeto de opções, que na Fetch API é chamado de objeto de inicialização (init object). Em ambas as funções, caso o método HTTP que deve ser utilizado não for informado, o padrão será utilizar GET.

Recomendo a leitura das respectivas documentações fornecidas nas três caixas "Saiba Mais" abaixo.

i) Saiba Mais

Documentação da função jQuery.ajax(): https://api.jquery.com/jquery.ajax/.

(i) | Saiba Mais

Documentação da Fetch API: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API>.

Usando a Fetch API: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch>.

7.11.2 AJAX com jQuery e com Fetch API Processando JSON

Nosso último exemplo é parecido com o primeiro, com a diferença que agora o servidor, ao invés de responder texto puro, nos enviará dados em JSON. Sim, JSON é texto puro também, mas como o servidor informará que o texto que está vindo é em JSON, as funções conseguirão fazer a desserialização dos dados de forma automática para que possamos tratá-los por causa do cabeçalho da resposa. Na Listagem 7.17 podem ser vistas as definições das duas funções, novamente uma usando jQuery e a outra a Fetch API.

```
Listagem 7.17: Exemplo 12
   Arquivo: /js/exemplo12.js
  function executarExemplo12jQuery( event ) {
2
      let q = prompt( "Quantidade de pessoas:" );
3
4
       $.ajax( "listarPessoas", {
5
           data: {
6
7
               quantidade: q
          },
8
           // esperando JSON no retorno
9
           dataType: "json"
10
       }).done( ( data, textStatus ) =>{
11
12
           // data já contém o objeto resultado do parse
13
           // do json retornado. isso é automático.
14
           let $div = $( "#divExemplo12" );
15
           $div.html( "" );
16
17
           data.forEach( pessoa => {
18
               $div.append(
19
                   `<div class="dadosPessoa">Pessoa:Nome: ${pessoa.nome}`+
20
                   `Data de Nascimento: ${pessoa.dataNasc}` +
21
                   Salário: R$ ${pessoa.salario}</div>`);
22
          });
23
```

```
24
       }).fail( ( jqXHR, textStatus, errorThrown ) => {
25
           alert( "Erro: " + errorThrown + "\n" +
26
                  "Status: " + textStatus );
27
       });
28
29
30
   }
31
   function executarExemplo12Fetch( event ) {
32
33
       let n = prompt( "Calcular a tabuada de:" );
34
35
       let parametros = new URLSearchParams();
36
       parametros.append( "quantidade", n );
37
38
       fetch( "listarPessoas", {
39
           method: "POST",
40
           body: parametros
41
       }).then( response => {
42
           // faz o parse do json em objeto e retorna
43
           return response.json();
44
       }).then( data => {
45
46
           let $div = $( "#divExemplo12" );
47
           $div.html( "" );
48
49
           data.forEach( pessoa => {
50
               $div.append(
51
                    `<div class="dadosPessoa">Pessoa:Nome: ${pessoa.nome}`+
52
                    `Data de Nascimento: ${pessoa.dataNasc}` +
53
                    `Salário: R$ ${pessoa.salario}</div>` );
           });
55
56
       }).catch( error => {
57
           alert( "Erro: " + error );
58
       });
59
60
61 }
```

Veja que agora os dados retornados serão processados, transformados em objetos JavaScript e então utililizados de forma transparente. Muito interessante concorda? Mais uma vez, todas as novidades estão comentadas!

7.12. RESUMO 241

7.12 Resumo

Chegamos ao fim de mais um Capítulo, onde aprendemos o básico sobre JavaScript que é a linguagem que, invariavelmente, qualquer desenvolvedor Web terá que lidar no seu trabalho. É importante que você tenha um certo domínio sobre a mesma e que, após a leitura e entendimento do Capítulo, você seja capaz de continuar seu aprendizado. No próximo Capítulo usaremos JavaScript para implementarmos um formulário de venda de produtos, com a definição da quantidade que certo produto será vendido. Usaremos também AJAX para podermos cancelar uma venda. Enfim, isso é assunto para o próximo Capítulo!

7.13 Projetos

Projeto 7.1: Implemente um cadastro simples (apenas a inserção) de dados de uma "fruta". A tabela deve ter como colunas um campo identificador (INT), um campo que armazenará o nome da fruta (VARCHAR) e um campo para armazenar a cor predominante da fruta (VARCHAR). Deve-se listar as frutas cadastrados usando AJAX, montando uma tabela dinamicamente, ao invés de processar os objetos no JSP do lado do servidor. Note que esse projeto é parecido com o Projeto 4.1 do Capítulo 4.

Projeto 7.2: Repita o Projeto 7.1, só que agora para a tabela "carro". Um carro deve ter um identificador, um nome (VARCHAR), um modelo (VARCHAR) e um ano de fabricação (INT). Note que esse projeto é parecido com o Projeto 4.2 do Capítulo 4.

Projeto 7.3: Repita o Projeto 7.1, só que agora para a tabela "produto". Um produto deve ter um identificador, uma descrição (VARCHAR) e uma quantidade em estoque (INT). Note que esse projeto é parecido com o Projeto 4.3 do Capítulo 4.

SISTEMA PARA VENDA DE PRODUTOS

"A imaginação é mais importante que o conhecimento".

Albert Einstein



ESTE Capítulo iremos construir mais um projeto completo, concluindo o aprendizado dos conhecimentos básicos para que você possa construir praticamente qualquer tipo de sistema que lide com bancos de dados relacionais, ou seja, aprenderemos a lidar com a implementação

de cadastros que têm relacionamentos muitos-para-muitos.

8.1 Introdução

Finalmente estamos aptos a construir uma aplicação Web em Java com a maioria das funcionalidades necessárias à maioria das aplicações Web que você desenvolverá na sua vida profissional. Iremos incrementar a aplicação criada no Capítulo 5 desenvolvendo mais alguns cadastros e amarrando todos eles em um cadastro de vendas de produtos. Vamos lá!

Além de clientes, cidades e estados, criaremos mais três cadastros com inserção, alteração e remoção de registros: unidades de medida, produtos e fornecedores. Com esses seis cadastros desenvolveremos a interface gráfica das vendas, onde poderemos gerar novas vendas e que, após serem feitas, poderão ser canceladas.

Na Figura 8.1 pode ser visto o DER da base de dados venda_produtos. Note que o nome da base é diferente do projeto anterior. Para esse projeto o *script* SQL para a criação da base de dados não será fornecido pois, além da estrutura, teremos algumas inserções já prontas para podermos testar. Para gerar a base, com o MariaDB/MySQL em execução, abra o modelo da base no MySQL Workbench, disponibilizado nos arquivos do Capítulo. Com o modelo aberto, clique no menu *Database* escolha a opção *Forward Engineer...* e siga o assistente. A base de dados, as tabelas e os relacionamentos serão criados, além de várias inserções para as tabelas estado, cidade, cliente, fornecedor, unidade_medida e produto que serão realizadas.

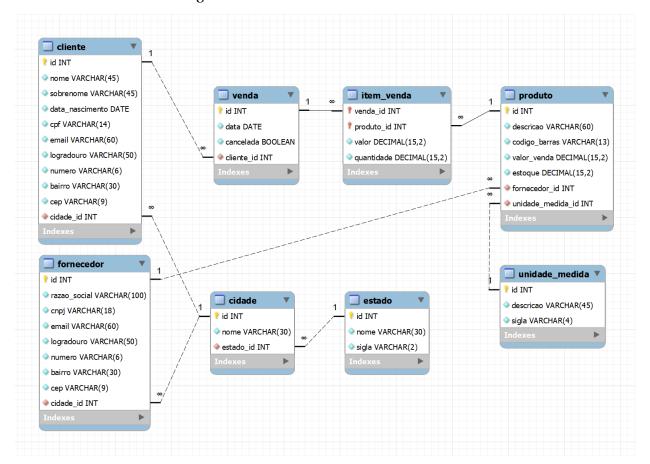


Figura 8.1: DER da base de dados

Fonte: Elaborada pelo autor

O diagrama de classes das entidades do projeto pode ser visto Figura 8.2. Veja que a classe ItemVenda é a classe que fará o papel de viabilizar o relacionamento muitospara-muitos entre produtos e vendas. Note que na UML existe a notação de classe associativa que poderia ter sido usada para representar esse relacionamento.

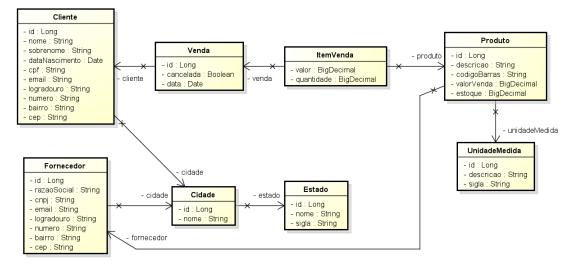


Figura 8.2: Diagrama de classes das entidades

Fonte: Elaborada pelo autor

Para que esse Capítulo não fique gigantesco, focaremos apenas nas novidades ou modificações que foram feitas em relação ao projeto anterior. Você poderá pegar o projeto pronto nos arquivos do Capítulo. A entidade Produto será usada como base para entendermos o que foi alterado e depois toda a parte da venda propriamente dita será detalhada com mais afinco.

8.2 Construindo o Sistema

Esta seção será dividida em várias subseções para que possamos organizar o que foi modificado em relação ao sistema anterior além, é claro, das funcionalidades novas. Os seis cadastros base são similares, então apenas um, como já explicado, será detalhado e o restante é por sua conta dar uma olhada, bastando consultar o código pronto no projeto disponibilizado.

8.2.1 Entidades e Validações

Começaremos discutindo nossas entidades. Todos os membros de todas elas agora serão de tipos de referência, ou seja, qualquer tipo que não seja primitivo. Os membros de tipo int, que usamos anteriormente para definir os identificadores, passarão a ser do tipo Long. Todos os tipos numéricos decimais serão do tipo BigDecimal, principalmente quando precisarmos representar valores monetários, fugindo assim da imprecisão dos tipos de ponto flutuante, pois dinheiro é algo muito importante

e precisamos tomar muito, muito, MUITO cuidado com esse tipo de dado em um sistema de verdade. Além disso, cada atributo será anotado com pelo menos uma anotação de validação, pois todos os nossos objetos das entidades serão validados, obrigatoriamente antes de serem submetidos aos seus respectivos DAOs, filtrando possíveis tentativas de adulteração dos dados submetidos através dos formulários. Na entidade Produto, apresentada na Listagem 8.1, podemos ver isso.

```
Listagem 8.1: Entidade "Produto"
   Arquivo: vendaprodutos/entidades/Produto.java
package vendaprodutos.entidades;
3 import java.math.BigDecimal;
4 import jakarta.validation.constraints.NotNull;
5 import jakarta.validation.constraints.Pattern;
6 import jakarta.validation.constraints.PositiveOrZero;
  import jakarta.validation.constraints.Size;
7
9
   * Entidade Produto.
10
11
    * @author Prof. Dr. David Buzatto
12
13
14
  public class Produto {
15
       @NotNull
16
       private Long id;
17
18
19
       @NotNull
       OSize( min = 1, max = 60)
20
       private String descricao;
21
22
       @NotNull
23
24
       QPattern( regexp = "^{d{13}}",
                 message = "deve corresponder à 999999999999" )
25
       private String codigoBarras;
26
27
       @NotNull
28
       @PositiveOrZero
29
       private BigDecimal valorVenda;
30
31
32
       @NotNull
```

```
private BigDecimal estoque;
33
34
       @NotNull
35
       private Fornecedor fornecedor;
36
37
       @NotNull
38
       private UnidadeMedida unidadeMedida;
40
       public Long getId() {
41
           return id;
42
43
44
       public void setId( Long id ) {
45
           this.id = id;
46
       }
47
48
       public String getDescricao() {
49
           return descricao;
50
       }
51
52
       public void setDescricao( String descricao ) {
           this.descricao = descricao;
54
55
56
       public String getCodigoBarras() {
           return codigoBarras;
59
60
       public void setCodigoBarras( String codigoBarras ) {
61
           this.codigoBarras = codigoBarras;
       }
63
64
       public BigDecimal getValorVenda() {
65
           return valorVenda;
66
       }
67
       public void setValorVenda( BigDecimal valorVenda ) {
           this.valorVenda = valorVenda;
70
       }
71
72
       public BigDecimal getEstoque() {
73
           return estoque;
74
```

```
}
75
76
       public void setEstoque( BigDecimal estoque ) {
77
           this.estoque = estoque;
78
       }
79
80
       public Fornecedor getFornecedor() {
81
           return fornecedor;
82
83
84
       public void setFornecedor( Fornecedor fornecedor ) {
85
86
           this.fornecedor = fornecedor;
       }
87
88
       public UnidadeMedida getUnidadeMedida() {
89
           return unidadeMedida;
90
91
92
       public void setUnidadeMedida( UnidadeMedida unidadeMedida ) {
           this.unidadeMedida = unidadeMedida;
94
       }
95
96
  }
97
```

Sempre precedendo a definição de um atributo da classe utilizaremos essas anotações. Na linha 17 o atributo identificador da entidade Produto é declarado e, logo antes, na linha 16, é usada a anotação (@NotNull) que indica que um objeto do tipo Produto, ao ser validado, não poderá ter esse atributo configurado com [null]. Veremos o mecanismo de validação já, já. O atributo descricao é anotado com [@NotNull], que já aprendemos, e OSize que, dependendo do tipo, verificará o tamanho do valor do mesmo. Para Strings, que é o nosso caso, é a quantidade de caracteres. Na linha 20 temos então que a descrição dos produtos pode ter no mínimo um caracetere e no máximo 60. Note que todas as nossas validações serão consistentes com as restrições existentes no DER do banco de dados. O código de barras de um produto terá obrigatoriamente que bater com um padrão de 13 dígitos, usando para isso a anotação (OPattern) com o atributo regexp (regular expression). Essa expressão regular diz que o padrão deve casar com a String inteira, sendo que isso é indicado pelos meta-símbolos ^ e \$ que significam, respectivamente, início e fim da String. O métasímbolo \d significa um dígito (0 a 9) e há a necessidade de se usar duas contrabarras, pois como em Java a contrabarra indica um caracetere de escape, precisamos usar

duas para escapá-la. O atributo message de <u>OPattern</u> é usado para criarmos uma mensagem personalizada para quando o produto for validado e esse atributo estiver fora do que é esperado.

A anotação (@PositiveOrZero) indica que o atributo do tipo BigDecimal deve ser zero ou qualquer valor positivo. As outras anotações que usaremos nas outras entidades são (@Positive) para valores positivos obrigatoriamente (zero não é positivo nem negativo!) e (@Email) para verificar se o valor representa um formato válido de e-mail. A implementação de referência da *Jakarta Bean Validation* faz parte do Jakarta EE 10, que estamos usando, e é fornecida pelo Hibernate Validator.

(i) | Saiba Mais

O site oficial da especificação e da implementação da API *Bean Validation* pode ser acessada pelo link https://beanvalidation.org/>

A validação dos objetos será feita sempre nos Servlets e é implementada pelo método validar, que tem sua implementação iniciada na linha 187 da classe Utils, mostrada na Listagem 8.2.

```
Listagem 8.2: Classe "Utils" com métodos estáticos utilitários Arquivo: vendaprodutos/utils/Utils.java
```

```
package vendaprodutos.utils;
3 import java.math.BigDecimal;
4 import java.sql.Date;
5 import java.sql.PreparedStatement;
6 import java.sql.ResultSet;
7 import java.sql.SQLException;
8 import java.time.LocalDate;
9 import java.time.format.DateTimeFormatter;
import java.time.format.DateTimeParseException;
import java.util.Arrays;
import java.util.LinkedHashSet;
13 import java.util.List;
14 import java.util.Set;
import jakarta.servlet.RequestDispatcher;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.validation.ConstraintViolation;
18 import jakarta.validation.Validation;
  import jakarta.validation.Validator;
```

```
20
   /**
21
    * Classe de métodos utilitários.
22
23
    * @author Prof. Dr. David Buzatto
24
25
   public abstract class Utils {
27
       // tanto o validador quanto o formatador
28
       // são thread safe
29
       // a implementação do validador normalmente tem cache
30
       private static Validator validador = Validation
31
                .buildDefaultValidatorFactory()
32
                .getValidator();
33
34
       private static DateTimeFormatter dtf = DateTimeFormatter
35
                .ofPattern( "yyyy-MM-dd" );
36
37
        * Lê um parâmetro Long do request.
39
        * Se a String for inválida, retorna null.
40
41
       public static Long getLong(
42
               HttpServletRequest request,
43
               String nomeParametro ) {
44
45
           Long v = null;
46
47
           try {
48
               v = Long.valueOf( request.getParameter( nomeParametro ) );
49
           } catch ( NumberFormatException exc ) {
50
           }
51
52
           return v;
53
54
       }
56
57
        * Lê um parâmetro BigDecimal do request.
58
        * Se a String for inválida, retorna null.
59
        */
60
       public static BigDecimal getBigDecimal(
61
```

```
HttpServletRequest request,
62
                String nomeParametro ) {
63
64
            BigDecimal v = null;
65
66
            try {
67
                v = new BigDecimal( request.getParameter( nomeParametro ) );
            } catch ( NumberFormatException exc ) {
            }
70
71
            return v;
72
73
       }
74
75
76
         * Converte uma String para Long.
77
         * Se a String for inválida, retorna null.
78
79
       public static Long getLong( String valor ) {
81
            Long v = null;
82
83
            try {
84
                v = Long.valueOf( valor );
85
            } catch ( NumberFormatException exc ) {
            }
87
88
            return v;
89
90
       }
93
         * Converte uma String para BigDecimal.
94
         * Se a String for inválida, retorna null.
95
96
       public static BigDecimal getBigDecimal( String valor ) {
            BigDecimal v = null;
99
100
            try {
101
                v = new BigDecimal( valor );
102
            } catch ( NumberFormatException exc ) {
103
```

```
}
104
105
106
            return v;
107
        }
108
109
        /*
110
         * Converte uma String no formato dd/MM/yyyy
111
         * para um java.sql.Date.
112
113
         * Se a String for inválida, retorna null.
114
115
        public static Date getDate( String data ) {
116
117
            Date d = null;
118
119
            try {
120
                 d = Date.valueOf( LocalDate.parse( data, dtf ) );
121
            } catch ( DateTimeParseException exc ) {
122
123
124
            return d;
125
126
        }
127
128
129
         * Faz a leitura da chave primária após inserção no banco.
130
         * Assume que o PreparedStatement foi configurado apropriadamente.
131
         */
132
        public static Long getChavePrimariaAposInsercao(
133
                 PreparedStatement stmt, String nomeColunaChave )
134
                 throws SQLException {
135
136
            Long pk = null;
137
138
            try ( ResultSet rsPK = stmt.getGeneratedKeys() ) {
139
                 if ( rsPK.next() ) {
140
                     pk = rsPK.getLong( nomeColunaChave );
141
                 }
142
143
144
145
            return pk;
```

```
146
        }
147
148
149
         * Realiza a validação e retorna um conjunto de violações de
150
         * restrições.
151
152
         * Não insere no retorno os campos que devem ser ignorados.
153
154
        private static Set<ConstraintViolation> validarObj(
155
                Object obj,
156
157
                String... ignorar ) {
158
            // uma lista dos campos à ignorar
159
            List<String> ignorarCampos = Arrays.<String>asList( ignorar );
160
161
            // conjunto que conterá todas as violações de restrição
162
            // que tenham caminho da propriedade igual
163
            // à alguma da lista de ignorar
164
            Set<ConstraintViolation> cvs = new LinkedHashSet<>();
165
166
            // valida e percorre todas as restrições violadas
167
168
            for ( ConstraintViolation cv : validador.validate( obj ) ) {
169
                // se a lista de campos à ignorar não tiver
170
                // o caminho da propriedade
171
                if (!ignorarCampos.contains(
172
                         cv.getPropertyPath().toString() ) ) {
173
                    cvs.add( cv );
174
                }
175
            }
176
177
178
            return cvs;
179
        }
180
181
182
         * Realiza a validação do objeto passado e lança
183
         * uma SQLException com todos os erros obtidos caso o
184
         * objeto seja inválido.
185
         */
186
        public static void validar(
187
```

```
Object obj,
188
                String... ignorar )
189
                throws SQLException {
190
191
            StringBuilder sb = new StringBuilder();
192
            Set<ConstraintViolation> cvs =
                    Utils.validarObj( obj, ignorar );
194
195
            if (!cvs.isEmpty()) {
196
197
                // cria uma mensagem com várias
198
                // tags que conterão as inconsistências
199
                // encontradas no objeto validado
200
                for ( ConstraintViolation cv : cvs ) {
201
                     sb.append( String.format(
202
                             "%s: %s",
203
                             cv.getPropertyPath(),
204
                             cv.getMessage() ) );
205
                }
206
207
                // lança a exceção com todos os erros
208
                throw new SQLException( sb.toString() );
209
210
            }
211
212
        }
213
214
215
         * Prepara o despacho para a página de erro de aplicação.
216
217
        public static RequestDispatcher prepararDespachoErro(
218
                HttpServletRequest request,
219
                String mensagem ) {
220
221
            // "Referer" é de onde veio a requisição
            request.setAttribute( "mensagemErro", mensagem );
223
            request.setAttribute( "voltarPara", request.getHeader( "Referer" ) );
224
225
            return request.getRequestDispatcher( "/erro.jsp" );
226
227
        }
228
229
```

```
230 }
```

Na nossa implementação da validação de objetos forneceremos a opção do usuário do método ignorar a validação de atributos de uma classe, caso seja necessário. Por exemplo, um Produto novo não terá identificador até que seja persistido, mas precisaremos validá-lo antes de ser enviado ao seu DAO. O método validar faz uso do método estático privado validarObj que é quem invoca de fato a infraestrutura de validação. Veja o código, está todo comentado. O método validar lançará uma exceção do tipo SQLException caso o objeto seja inválido e essa exceção terá encapsulada como mensagem a descrição de todas as inconsistências identificadas no objeto. Essa mensagem será configurada com uma série de tags (li) e será usada na página de erros da aplicação. Veremos essa questão da página de erros mais adiante no Capítulo.

Vamos ver agora o que mudou na nossa camada de persistência.

8.2.2 DAO

Temos duas novidades na nossa camada de persistência. A primeira é a implementação da interface AutoCloseable que permitirá que usemos objetos dos nossos DAOs na construção *try-with-resources* que, por sua vez, fará o fechamento automático da conexão dos DAOs ao terminarem de serem utilizados. Para isso precisaremos implementar o método close que substituirá o nosso antigo fecharConexao. A implementação do DAO genérico é mostrada na Listagem 8.3, sendo que o método close pode ser visto entre as linhas 57 e 59.

```
13
   public abstract class DAO<Tipo> implements AutoCloseable {
15
       // cada DAO terá uma conexão.
16
       private Connection conexao;
17
18
19
        * Construtor do DAO.
20
        * É nesse construtor que a conexão é criada.
21
22
        * Othrows SQLException
23
24
       public DAO() throws SQLException {
25
26
27
            * Usa-se o método getConnection() da fábrica de conexões,
28
            * para criar uma conexão para o DAO.
29
30
           conexao = ConnectionFactory.getConnection();
32
       }
33
34
35
        * Método para obter a conexão criada.
36
37
        * @return Retorna a conexão.
38
39
       public Connection getConnection() {
40
           return conexao;
41
       }
42
43
44
        * Método para fechar a conexão aberta.
45
46
        * Não precisa ser invocado explicitamente caso
47
        * o objeto do DAO tenha sido criado usando a construção
        * try-with-resources.
49
50
        * É sobrescrito da interface AutoCloseable.
51
52
        * @throws SQLException Caso ocorra algum erro
53
        * durante o fechamento da conexão.
54
```

```
*/
55
       @Override
       public void close() throws SQLException {
           conexao.close();
58
       }
59
       /**
        * Método abstrato para salvar uma instância de uma
        * entidade da base de dados.
64
        * É o "C" do CRUD.
65
        * Oparam obj Instância do objeto da entidade a ser salvo.
        st Othrows SQLException Caso ocorra algum erro durante a gravação.
69
       public abstract void salvar( Tipo obj ) throws SQLException;
70
71
       /**
72
        * Método abstrato para atualizar uma instância de uma
        * entidade da base de dados.
74
75
        * É o "U" do CRUD.
76
77
        * Oparam obj Instância do objeto da entidade a ser atualizado.
78
        * Othrows SQLException Caso ocorra algum erro durante a atualização.
        */
80
       public abstract void atualizar( Tipo obj ) throws SQLException;
81
82
       /**
83
        * Método abstrato para excluir uma instância de uma
        * entidade da base de dados.
86
        * É o "D" do CRUD.
87
88
        * Oparam obj Instância do objeto da entidade a ser salvo.
        * Othrows SQLException Caso ocorra algum erro durante a exclusão.
       public abstract void excluir( Tipo obj ) throws SQLException;
92
93
94
        * Método abstrato para obter todas as instâncias de uma
95
        * entidade da base de dados.
96
```

```
97
         * É o "R" do CRUD.
98
99
         * @return Lista de todas as instâncias da entidade.
100
         * Othrows SQLException Caso ocorra algum erro durante a consulta.
101
102
       public abstract List<Tipo> listarTodos() throws SQLException;
103
104
105
106
         * Método abstrato para obter uma instância de uma
         * entidade pesquisando pelo seu atributo identificador.
107
108
         * É o "R" do CRUD.
109
110
         * @param id Identificador da instância a ser obtida.
111
         * @return Instância relacionada ao id passado, ou null caso não seja
112
113
         * encontrada.
         * Othrows SQLException Caso ocorra algum erro durante a consulta.
114
115
       public abstract Tipo obterPorId( Long id ) throws SQLException;
116
117
118 }
```

A outra mudança que temos em nossos DAOs é que agora, toda entidade que tiver um objeto submetido ao método salvar(...), será modificada para conter o identificador que foi gerado pelo SGBD. Na Listagem 8.4 é apresentado o DAO para produtos.

```
Listagem 8.4: Código da classe "ProdutoDAO"
Arquivo: vendaprodutos/dao/ProdutoDAO.java

package vendaprodutos.dao;

import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.List;
import vendaprodutos.entidades.Cidade;
import vendaprodutos.entidades.Estado;
import vendaprodutos.entidades.Fornecedor;
```

```
import vendaprodutos.entidades.Produto;
  import vendaprodutos.entidades.UnidadeMedida;
  import vendaprodutos.utils.Utils;
14
   /**
15
    * DAO para a entidade Produto.
16
17
    * @author Prof. Dr. David Buzatto
18
19
  public class ProdutoDAO extends DAO<Produto> {
21
22
       public ProdutoDAO() throws SQLException {
       }
23
24
       @Override
25
       public void salvar( Produto obj ) throws SQLException {
26
27
28
           PreparedStatement stmt = getConnection().prepareStatement(
                   INSERT INTO
30
                   produto(
31
                        descricao,
32
                        codigo_barras,
33
                        valor_venda,
34
                        estoque,
                        fornecedor_id,
                        unidade_medida_id )
37
                    VALUES( ?, ?, ?, ?, ?, ?);
38
                    0.00
39
                   new String[]{ "insert_id" } ); // para retorno da chave
40
                                                     // primária gerada
42
           stmt.setString( 1, obj.getDescricao() );
43
           stmt.setString( 2, obj.getCodigoBarras() );
44
           stmt.setBigDecimal( 3, obj.getValorVenda() );
45
           stmt.setBigDecimal( 4, obj.getEstoque() );
           stmt.setLong( 5, obj.getFornecedor().getId() );
47
           stmt.setLong( 6, obj.getUnidadeMedida().getId() );
49
           stmt.executeUpdate();
50
           obj.setId( Utils.getChavePrimariaAposInsercao( stmt, "insert_id" ) );
51
           stmt.close();
52
```

```
53
       }
54
55
       @Override
56
       public void atualizar( Produto obj ) throws SQLException {
57
58
           PreparedStatement stmt = getConnection().prepareStatement(
60
                    UPDATE produto
61
                    SET
62
                        descricao = ?,
63
                        codigo_barras = ?,
64
                        valor_venda = ?,
65
                        estoque = ?,
66
67
                        fornecedor_id = ?,
                        unidade_medida_id = ?
68
                    WHERE
69
70
                        id = ?;
                    """);
71
72
           stmt.setString( 1, obj.getDescricao() );
73
           stmt.setString( 2, obj.getCodigoBarras() );
74
           stmt.setBigDecimal( 3, obj.getValorVenda() );
75
           stmt.setBigDecimal( 4, obj.getEstoque() );
76
           stmt.setLong( 5, obj.getFornecedor().getId() );
77
           stmt.setLong( 6, obj.getUnidadeMedida().getId() );
78
           stmt.setLong( 7, obj.getId() );
79
80
           stmt.executeUpdate();
81
           stmt.close();
82
83
       }
84
85
       @Override
86
       public void excluir( Produto obj ) throws SQLException {
87
           PreparedStatement stmt = getConnection().prepareStatement(
89
90
                    DELETE FROM produto
91
                    WHERE
92
                        id = ?;
93
                    """);
94
```

```
95
            stmt.setLong( 1, obj.getId() );
96
97
            stmt.executeUpdate();
98
            stmt.close();
99
100
        }
101
102
        @Override
103
        public List<Produto> listarTodos() throws SQLException {
104
105
106
            List<Produto> lista = new ArrayList<>();
107
            PreparedStatement stmt = getConnection().prepareStatement(
108
109
                     SELECT
110
                         p.id idProduto,
111
112
                          p.descricao descricaoProduto,
                          p.codigo_barras codigoBarrasProduto,
113
                         p.valor_venda valorVendaProduto,
114
                          p.estoque estoqueProduto,
115
                         u.id idUnidadeMedida,
116
117
                          u.descricao descricaoUnidadeMedida,
                         u.sigla siglaUnidadeMedida,
118
                          f.id idFornecedor,
119
                         f.razao_social razaoSocialFornecedor,
120
                         f.cnpj cnpjFornecedor,
121
                         f.email emailFornecedor,
122
                         f.logradouro logradouroFornecedor,
123
                          f.numero numeroFornecedor,
124
125
                         f.bairro bairroFornecedor,
                          f.cep cepFornecedor,
126
127
                          ci.id idCidade,
                          ci.nome nomeCidade,
128
                          e.id idEstado,
129
130
                          e.nome nomeEstado,
131
                          e.sigla siglaEstado
                     FROM
132
                          produto p,
133
                          unidade_medida u,
134
                          fornecedor f,
135
                          cidade ci,
136
```

```
estado e
137
                    WHERE
138
                         p.unidade_medida_id = u.id AND
139
                         p.fornecedor_id = f.id AND
140
                         f.cidade id = ci.id AND
141
                         ci.estado id = e.id
142
                     ORDER BY p.descricao;
143
                     """ );
144
145
            ResultSet rs = stmt.executeQuery();
146
147
            while ( rs.next() ) {
148
149
                Produto p = new Produto();
150
                UnidadeMedida u = new UnidadeMedida();
151
                Fornecedor f = new Fornecedor();
152
                Cidade ci = new Cidade();
153
154
                Estado e = new Estado();
155
                p.setId( rs.getLong( "idProduto" ) );
156
                p.setDescricao( rs.getString( "descricaoProduto" ) );
157
                p.setCodigoBarras( rs.getString( "codigoBarrasProduto" ) );
158
                p.setValorVenda( rs.getBigDecimal( "valorVendaProduto" ) );
159
                p.setEstoque( rs.getBigDecimal( "estoqueProduto" ) );
160
                p.setUnidadeMedida( u );
161
                p.setFornecedor( f );
162
163
                u.setId( rs.getLong( "idUnidadeMedida" ) );
164
                u.setDescricao( rs.getString( "descricaoUnidadeMedida" ) );
165
                u.setSigla( rs.getString( "siglaUnidadeMedida" ) );
166
167
                f.setId( rs.getLong( "idFornecedor" ) );
168
                f.setRazaoSocial( rs.getString( "razaoSocialFornecedor" ) );
169
                f.setCnpj( rs.getString( "cnpjFornecedor" ) );
170
                f.setEmail( rs.getString( "emailFornecedor" ) );
171
                f.setLogradouro( rs.getString( "logradouroFornecedor" ) );
172
                f.setNumero( rs.getString( "numeroFornecedor" ) );
173
                f.setBairro( rs.getString( "bairroFornecedor" ) );
174
                f.setCep( rs.getString( "cepFornecedor" ) );
175
                f.setCidade( ci );
176
177
                ci.setId( rs.getLong( "idCidade" ) );
178
```

```
ci.setNome( rs.getString( "nomeCidade" ) );
179
                 ci.setEstado( e );
180
181
                 e.setId( rs.getLong( "idEstado" ) );
182
                 e.setNome( rs.getString( "nomeEstado" ) );
183
                 e.setSigla( rs.getString( "siglaEstado" ) );
184
185
                 lista.add( p );
186
187
            }
188
189
            rs.close();
190
            stmt.close();
191
192
            return lista;
193
194
        }
195
196
        @Override
197
        public Produto obterPorId( Long id ) throws SQLException {
198
199
            Produto produto = null;
200
201
            PreparedStatement stmt = getConnection().prepareStatement(
202
203
                     SELECT
204
                         p.id idProduto,
205
206
                          p.descricao descricaoProduto,
                          p.codigo_barras codigoBarrasProduto,
207
                          p.valor venda valorVendaProduto,
208
209
                          p.estoque estoqueProduto,
                          u.id idUnidadeMedida,
210
211
                          u.descricao descricaoUnidadeMedida,
                         u.sigla siglaUnidadeMedida,
212
213
                          f.id idFornecedor,
214
                          f.razao_social razaoSocialFornecedor,
                          f.cnpj cnpjFornecedor,
215
                         f.email emailFornecedor,
216
                          f.logradouro logradouroFornecedor,
217
                          f.numero numeroFornecedor,
218
                          f.bairro bairroFornecedor,
219
                          f.cep cepFornecedor,
220
```

```
ci.id idCidade,
221
                         ci.nome nomeCidade,
222
                         e.id idEstado,
223
                         e.nome nomeEstado,
224
                         e.sigla siglaEstado
225
                     FROM
226
227
                         produto p,
                         unidade_medida u,
228
                         fornecedor f,
229
230
                         cidade ci,
                         estado e
231
                     WHERE
232
                         p.id = ? AND
233
                         p.unidade_medida_id = u.id AND
234
                         p.fornecedor_id = f.id AND
235
                         f.cidade_id = ci.id AND
236
                         ci.estado_id = e.id;
237
                     """);
238
239
            stmt.setLong( 1, id );
240
241
            ResultSet rs = stmt.executeQuery();
242
243
            if ( rs.next() ) {
244
245
                produto = new Produto();
246
                UnidadeMedida u = new UnidadeMedida();
247
                Fornecedor f = new Fornecedor();
248
                Cidade ci = new Cidade();
249
                Estado e = new Estado();
251
                produto.setId( rs.getLong( "idProduto" ) );
252
253
                produto.setDescricao( rs.getString( "descricaoProduto" ) );
                produto.setCodigoBarras( rs.getString( "codigoBarrasProduto" ) );
254
                produto.setValorVenda( rs.getBigDecimal( "valorVendaProduto" ) );
255
                produto.setEstoque( rs.getBigDecimal( "estoqueProduto" ) );
256
                produto.setUnidadeMedida( u );
257
                produto.setFornecedor( f );
258
259
                u.setId( rs.getLong( "idUnidadeMedida" ) );
260
                u.setDescricao( rs.getString( "descricaoUnidadeMedida" ) );
261
                u.setSigla( rs.getString( "siglaUnidadeMedida" ) );
262
```

```
263
                 f.setId( rs.getLong( "idFornecedor" ) );
264
                 f.setRazaoSocial( rs.getString( "razaoSocialFornecedor" ) );
265
                 f.setCnpj( rs.getString( "cnpjFornecedor" ) );
266
                 f.setEmail( rs.getString( "emailFornecedor" ) );
267
                 f.setLogradouro( rs.getString( "logradouroFornecedor" ) );
268
                 f.setNumero( rs.getString( "numeroFornecedor" ) );
269
                 f.setBairro( rs.getString( "bairroFornecedor" ) );
270
                 f.setCep( rs.getString( "cepFornecedor" ) );
271
                 f.setCidade( ci );
272
273
                 ci.setId( rs.getLong( "idCidade" ) );
274
                 ci.setNome( rs.getString( "nomeCidade" ) );
275
                 ci.setEstado( e );
276
277
                 e.setId( rs.getLong( "idEstado" ) );
278
                 e.setNome( rs.getString( "nomeEstado" ) );
279
                 e.setSigla( rs.getString( "siglaEstado" ) );
280
281
            }
282
283
            rs.close();
284
285
            stmt.close();
286
            return produto;
287
288
        }
289
290
291
         * Atualização do estoque para o cancelamento de vendas.
292
293
        public void atualizarEstoque( Produto obj ) throws SQLException {
294
295
            PreparedStatement stmt = getConnection().prepareStatement(
296
297
298
                     UPDATE produto
299
                     SET
                         estoque = ?
300
                     WHERE
301
302
                         id = ?;
                     """);
303
304
```

```
stmt.setBigDecimal( 1, obj.getEstoque() );
stmt.setLong( 2, obj.getId() );
stmt.executeUpdate();
stmt.close();
}
```

Veja que a criação do PreparedStatement do método salvar(...) agora usa outra versão do método prepareStatement(...) de Connection. Nessa versão, além da String contendo o código SQL que será executado, é passado um array de Strings com o nome da ou das colunas que representam as chaves primárias daquela tabela. Essa "artimanha" funciona para colunas que são auto-incrementáveis, que é o caso dos nossos identificadores. Precisaremos dessa característica no cadastro das vendas. Veremos isso mais para frente. Quando fazemos uso desse recurso, após a persistência do objeto, que gerará uma nova linha/registro/tupla na tabela, precisaremos usar o mesmo PreparedStatement para obter a ou as chaves. Isso será feito no método getChavePrimariaAposInsercao(...) da classe Utils (linha 133 da Listagem 8.2). Como consistentemente estamos usando uma coluna chamada id como chave primária auto-incrementável, usaremos esse método para todas as nossas entidades, com exceção da ItemVenda que funcionará de outra forma. Note que o nome que foi usado é insert_id, requisito necessário da versão atual do driver JDBC que estamos utilizando.

Muito bem, temos nossas entidades prontas para serem validadas e a camada de persistência atualizada. Vamos ver agora o que mudou nos nossos Servlets.

8.2.3 Servlets

Nossos Servlets continuarão a funcionar e a ter a mesma estrutura que vimos anteriormente, mas agora com algumas melhorias. Na Listagem 8.5 pode ser visto o código completo do Servlet de produtos.

```
Listagem 8.5: Código-fonte do Servlet "ProdutosServlet"
Arquivo: vendaprodutos/controladores/ProdutosServlet.java

package vendaprodutos.controladores;

2
```

```
import java.io.IOException;
  import java.math.BigDecimal;
5 import java.sql.SQLException;
6 import jakarta.servlet.RequestDispatcher;
7 import jakarta.servlet.ServletException;
  import jakarta.servlet.annotation.WebServlet;
9 import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import vendaprodutos.dao.FornecedorDAO;
  import vendaprodutos.dao.ProdutoDAO;
13
14 import vendaprodutos.dao.UnidadeMedidaDAO;
import vendaprodutos.entidades.Fornecedor;
import vendaprodutos.entidades.Produto;
  import vendaprodutos.entidades.UnidadeMedida;
17
  import vendaprodutos.utils.Utils;
19
20
  /**
   * Servlet para tratar Produtos.
21
22
    * @author Prof. Dr. David Buzatto
23
24
  @WebServlet( name = "ProdutosServlet",
25
                urlPatterns = { "/processaProdutos" } )
26
  public class ProdutosServlet extends HttpServlet {
27
28
      protected void processRequest(
29
               HttpServletRequest request,
30
               HttpServletResponse response )
31
               throws ServletException, IOException {
           String acao = request.getParameter( "acao" );
34
           RequestDispatcher disp = null;
35
36
           try ( ProdutoDAO daoProduto = new ProdutoDAO();
37
                 FornecedorDAO daoFornecedor = new FornecedorDAO();
                 UnidadeMedidaDAO daoUnidadeMedida = new UnidadeMedidaDAO() ) {
               if ( acao.equals( "inserir" ) ) {
41
42
                   String descricao = request.getParameter( "descricao" );
43
                   String codigoBarras = request.getParameter( "codigoBarras" );
44
```

```
BigDecimal valorVenda = Utils.getBigDecimal(
45
                            request, "valorVenda" );
46
                   BigDecimal estoque = Utils.getBigDecimal(
47
                            request, "estoque" );
48
                   Long idFornecedor = Utils.getLong(
49
                            request, "idFornecedor" );
50
                   Long idUnidadeMedida = Utils.getLong(
                            request, "idUnidadeMedida" );
52
53
                   Fornecedor f = daoFornecedor.obterPorId( idFornecedor );
54
                   UnidadeMedida u =
55
                    → daoUnidadeMedida.obterPorId(idUnidadeMedida);
56
                   Produto p = new Produto();
57
                   p.setDescricao( descricao );
58
                   p.setCodigoBarras( codigoBarras );
59
                   p.setValorVenda( valorVenda );
60
                   p.setEstoque( estoque );
61
                   p.setFornecedor( f );
62
                   p.setUnidadeMedida( u );
63
64
                   Utils.validar( p, "id" );
65
                   daoProduto.salvar( p );
66
                   disp = request.getRequestDispatcher(
67
                            "/formularios/produtos/listagem.jsp" );
68
69
               } else if ( acao.equals( "alterar" ) ) {
70
71
                   Long id = Utils.getLong( request, "id" );
72
                   String descricao = request.getParameter( "descricao" );
73
                   String codigoBarras = request.getParameter( "codigoBarras" );
74
                   BigDecimal valorVenda = Utils.getBigDecimal(
75
                            request, "valorVenda" );
76
                   BigDecimal estoque = Utils.getBigDecimal(
77
                            request, "estoque" );
78
                   Long idFornecedor = Utils.getLong(
79
                            request, "idFornecedor" );
80
                   Long idUnidadeMedida = Utils.getLong(
81
                            request, "idUnidadeMedida" );
82
83
                   Fornecedor f = daoFornecedor.obterPorId( idFornecedor );
84
```

```
UnidadeMedida u =
85
                         daoUnidadeMedida.obterPorId(idUnidadeMedida);
86
                    Produto p = daoProduto.obterPorId( id );
87
                    p.setDescricao( descricao );
88
                    p.setCodigoBarras( codigoBarras );
89
                    p.setValorVenda( valorVenda );
                    p.setEstoque( estoque );
91
                    p.setFornecedor( f );
92
                    p.setUnidadeMedida( u );
93
94
95
                    Utils.validar( p );
                    daoProduto.atualizar( p );
                    disp = request.getRequestDispatcher(
97
                             "/formularios/produtos/listagem.jsp" );
98
99
                } else if ( acao.equals( "excluir" ) ) {
100
101
                    Long id = Utils.getLong( request, "id" );
102
                    Produto p = daoProduto.obterPorId( id );
103
104
                    daoProduto.excluir( p );
105
                    disp = request.getRequestDispatcher(
106
                             "/formularios/produtos/listagem.jsp" );
107
108
                } else {
109
110
111
                    Long id = Utils.getLong( request, "id" );
                    Produto p = daoProduto.obterPorId( id );
112
                    request.setAttribute( "produto", p );
113
114
                    if ( acao.equals( "prepararAlteracao" ) ) {
115
116
                         disp = request.getRequestDispatcher(
                                  "/formularios/produtos/alterar.jsp" );
117
                    } else if ( acao.equals( "prepararExclusao" ) ) {
118
119
                         disp = request.getRequestDispatcher(
                                  "/formularios/produtos/excluir.jsp" );
120
                     }
121
122
                }
123
124
            } catch ( SQLException exc ) {
125
```

```
disp = Utils.prepararDespachoErro( request, exc.getMessage() );
126
            }
127
128
            if ( disp != null ) {
129
                 disp.forward( request, response );
130
            }
131
132
        }
133
134
135
        @Override
        protected void doGet(
136
137
                 HttpServletRequest request,
                 HttpServletResponse response )
138
                 throws ServletException, IOException {
139
            processRequest( request, response );
140
        }
141
142
        @Override
143
        protected void doPost(
144
                 HttpServletRequest request,
145
                 HttpServletResponse response )
146
                 throws ServletException, IOException {
147
            processRequest( request, response );
148
        }
149
150
        @Override
151
        public String getServletInfo() {
152
            return "ProdutosServlet";
153
        }
154
155
156 }
```

As ações esperadas dos formulários serão as mesmas. Entre as linhas 37 e 39 instanciamos três DAOs, um para produtos, um para fornecedores e um para unidades de medida. Precisaremos dos dois últimos para trazer do banco de dados os objetos necessários para compor o produto.

A partir da linha 43 iniciam-se as instruções para obtenção dos dados do formulário de criação/inserção de um novo produto. Veja que entre as linhas 45 e 52 usamos os métodos getBigDecimal(...) e getLong(...) para a obtenção dos dados numéricos que virão do formulário. Estamos prevendo o caso de haver alguma inserção incorreta, passando a validação do lado do cliente e, nesses métodos, que

estão definidos respectivamente nas linhas 42 e 61 da Listagem 8.2, é feita a obtenção dos valores e, caso haja algum problema, eles retornarão null ao invés de lançar exceção. Nas linhas 54 e 55 usamos os DAOs dos fornecedores e unidades de medida para a obtenção dos objetos que contém os ids que vieram do formulário. Entre as linhas 57 e 63 configuramos todos os atributos do novo produto que será inserido e, na linha 65, antes de "salvarmos" o objeto no banco de dados, realizamos a validação. Note que estamos ignorando o atributo id, pois um novo produto ainda não o tem. O método de validação lançará uma SQLException caso haja algum erro na validação e, caso tudo esteja correto, o produto será salvo na linha 66 e na linha 67 preparamos o despacho para voltar à página de listagem. Outra coisa, após a execução do método salvar(...) do DAO de produtos, o produto terá seu identificador configurado. Nesse caso não fará muita diferença, mas precisaremos desse comportamento da obtenção do id gerado pelo auto-incremento quando formos lidar com as vendas.

O restante dos tratamentos das ações dos formulários continuam praticamente as mesmas. Como nossos DAOs foram instanciados usando um *try-with-resources*, não precisamos fechar as conexões dos DAOs de forma explícita, simplificando um pouco nosso código. Caso haja qualquer problema de validação ou mesmo em nível do SGBD, a SQLException lançada será capturada no catch da linha 125. Ali usamos mais um método da classe Utils, definido na linha 218 da Listagem 8.2, em que preparamos o redirecionamento para a página de erros do nosso sistema. Note que na linha 223 da classe Utils configuramos um atributo do *request* com o nome de mensagemErro, em que a mensagem da exceção será usada e na linha 224 configuramos o segundo parâmetro, chamado voltarPara, em que obtemos de qual recurso que veio a requisição para o Servlet, usando para isso o cabeçalho Referer da requisição, permitindo que criemos o *link* apropriado para que possamos voltar à página de onde o erro foi gerado.

Por falar em erros, na Listagem 8.6 pode ser visto o arquivo JSP que tratará os erros de validação e de persistência do sistema.

```
<title>Erro(s)!</title>
8
       <meta charset="UTF-8">
9
       <meta name="viewport"</pre>
10
              content="width=device-width, initial-scale=1.0">
11
       <link rel="stylesheet"</pre>
12
              href="${cp}/css/estilos.css"/>
13
     </head>
14
15
     <body>
16
17
       <h1>Erro(s)!</h1>
18
19
       <div id="divErros">
20
         <u1>
21
            ${requestScope.mensagemErro}
22
         23
       </div>
24
25
       <a href="${requestScope.voltarPara}">Voltar</a>
26
27
     </body>
28
29
   </html>
30
```

Com o conhecimento que você já adquiriu com o que estamos trabalhando você será capaz de entender o que essa página faz.

8.2.4 Cadastro de Vendas

Agora vamos detalhar todo o cadastro de vendas do sistema onde, de certa forma, todos os cadastros básicos convergirão. Começaremos com o lado do servidor, o chamado *back-end*.

Back-End

Na Listagem 8.7 pode ser vista a entidade Venda. Uma venda só poderá ser cadastrada e, caso necessário, ser cancelada. Estamos tentando simular um sistema real com anotações fiscais e esse tipo de coisa precisa acontecer, ou seja, uma venda nunca deve ser excluída! Para o tratamento do cancelamento temos o atributo cancelada.

```
Listagem 8.7: Entidade "Venda"
  Arquivo: vendaprodutos/entidades/Venda.java
package vendaprodutos.entidades;
3 import java.sql.Date;
4 import jakarta.validation.constraints.NotNull;
6 /**
   * Entidade Venda.
7
   * @author Prof. Dr. David Buzatto
public class Venda {
13
       @NotNull
      private Long id;
14
15
      @NotNull
16
17
      private Boolean cancelada;
18
      @NotNull
19
      private Date data;
21
      @NotNull
22
      private Cliente cliente;
23
24
      public Long getId() {
           return id;
26
       }
27
28
       public void setId( Long id ) {
29
           this.id = id;
30
       }
32
      public Boolean getCancelada() {
33
           return cancelada;
34
       }
35
      public void setCancelada( Boolean cancelada ) {
           this.cancelada = cancelada;
38
39
       }
```

```
40
       public Date getData() {
41
           return data;
42
43
44
       public void setData( Date data ) {
45
           this.data = data;
       }
47
48
       public Cliente getCliente() {
49
           return cliente;
50
51
       public void setCliente( Cliente cliente ) {
           this.cliente = cliente;
54
       }
55
56
57 }
```

Na Listagem 8.8 a entidade ItemVenda é apresentada.

```
Listagem 8.8: Entidade "ItemVenda"
  Arquivo: vendaprodutos/entidades/ItemVenda.java
package vendaprodutos.entidades;
3 import java.math.BigDecimal;
4 import jakarta.validation.constraints.NotNull;
5 import jakarta.validation.constraints.Positive;
6 import jakarta.validation.constraints.PositiveOrZero;
7
8
   * Entidade ItemVenda.
9
10
   * @author Prof. Dr. David Buzatto
11
12
public class ItemVenda {
14
      @NotNull
15
      private Venda venda;
16
17
```

```
@NotNull
18
       private Produto produto;
19
20
       @NotNull
21
       @PositiveOrZero
22
       private BigDecimal valor;
23
       @NotNull
25
       @Positive
26
       private BigDecimal quantidade;
27
28
       public Venda getVenda() {
29
           return venda;
       }
31
32
       public void setVenda( Venda venda ) {
33
           this.venda = venda;
34
35
       public Produto getProduto() {
37
           return produto;
38
39
40
       public void setProduto( Produto produto ) {
41
           this.produto = produto;
42
       }
43
44
       public BigDecimal getValor() {
45
           return valor;
46
       }
47
       public void setValor( BigDecimal valor ) {
49
           this.valor = valor;
50
       }
51
       public BigDecimal getQuantidade() {
           return quantidade;
       }
55
56
       public void setQuantidade( BigDecimal quantidade ) {
57
           this.quantidade = quantidade;
58
       }
59
```

```
60 61 }
```

Cada venda pode conter um ou mais produtos que, por sua vez, podem ter sido vendidos em uma ou mais vendas, sendo assim, precisamos ter uma entidade que associa as outras duas. O valor do item da venda é o valor do produto naquele momento em que foi vendido, visto que o valor de venda de um produto pode variar com o tempo, mas o valor usado no momento da venda deve ser mantido! Além disso, todo item da venda tem uma quantidade associada, pois podemos ter comprado, por exemplo, duas caixas de ovos ou um quilo e meio de peito de frango.

O código do DAO que trata as vendas é apresentado na Listagem 8.9. O método atualizar(...) será usado para cancelar uma venda. Além disso, não há razão para excluirmos vendas realizadas, sendo assim, o corpo do método está vazio.

```
Listagem 8.9: Código da classe "VendaDAO"
  Arquivo: vendaprodutos/dao/VendaDAO.java
  package vendaprodutos.dao;
2
3 import java.sql.PreparedStatement;
4 import java.sql.ResultSet;
5 import java.sql.SQLException;
6 import java.util.ArrayList;
7 import java.util.List;
8 import vendaprodutos.entidades.Cidade;
9 import vendaprodutos.entidades.Cliente;
import vendaprodutos.entidades.Estado;
  import vendaprodutos.entidades.Venda;
  import vendaprodutos.utils.Utils;
13
14
   * DAO para a entidade Venda.
15
16
    * @author Prof. Dr. David Buzatto
17
18
  public class VendaDAO extends DAO<Venda> {
19
20
      public VendaDAO() throws SQLException {
21
      }
22
23
```

```
@Override
24
       public void salvar( Venda obj ) throws SQLException {
25
26
           PreparedStatement stmt = getConnection().prepareStatement(
27
28
                    INSERT INTO
29
                    venda(
                        data,
31
                        cancelada,
32
                        cliente_id )
33
                    VALUES( ?, ?, ?);
34
35
                    new String[]{ "insert_id" } );
37
           stmt.setDate( 1, obj.getData() );
38
           stmt.setBoolean( 2, obj.getCancelada() );
39
           stmt.setLong( 3, obj.getCliente().getId() );
40
41
           stmt.executeUpdate();
           obj.setId( Utils.getChavePrimariaAposInsercao( stmt, "insert_id" ) );
43
           stmt.close();
44
45
       }
46
47
       @Override
       public void atualizar( Venda obj ) throws SQLException {
49
50
           // a atualização das vendas será a de cancelamento
51
52
           PreparedStatement stmt = getConnection().prepareStatement(
                    UPDATE venda
55
                    SET
56
                        data = ?,
57
                        cancelada = ?,
58
                        cliente_id = ?
                    WHERE
                        id = ?;
61
                    """):
62
63
           stmt.setDate( 1, obj.getData() );
64
           stmt.setBoolean( 2, obj.getCancelada() );
65
```

```
stmt.setLong( 3, obj.getCliente().getId() );
66
            stmt.setLong( 4, obj.getId() );
67
68
            stmt.executeUpdate();
69
            stmt.close();
70
71
        }
72
73
        @Override
74
        public void excluir( Venda obj ) throws SQLException {
75
            // vendas não são excluídas!
76
77
78
        @Override
79
        public List<Venda> listarTodos() throws SQLException {
80
81
            List<Venda> lista = new ArrayList<>();
82
83
            PreparedStatement stmt = getConnection().prepareStatement(
85
                     SELECT
86
                         v.id idVenda,
87
                         v.data dataVenda,
88
                         v.cancelada vendaCancelada,
89
                         c.id idCliente,
90
                         c.nome nomeCliente,
91
                         c.sobrenome sobrenomeCliente,
92
                         c.data_nascimento dataNascimentoCliente,
93
                         c.cpf cpfCliente,
94
                         c.email emailCliente,
95
                         c.logradouro logradouroCliente,
96
                         c.numero numeroCliente,
97
                         c.bairro bairroCliente,
98
                         c.cep cepCliente,
99
                         ci.id idCidade,
100
101
                         ci.nome nomeCidade,
                         e.id idEstado,
102
                         e.nome nomeEstado,
103
                         e.sigla siglaEstado
104
                     FROM
105
                         venda v,
106
                         cliente c,
107
```

```
108
                         cidade ci,
                         estado e
109
                     WHERE
110
                         v.cliente_id = c.id AND
111
                         c.cidade id = ci.id AND
112
                         ci.estado id = e.id
113
                     ORDER BY v.data DESC, c.nome;
114
                     """);
115
116
            ResultSet rs = stmt.executeQuery();
117
118
119
            while ( rs.next() ) {
120
                Venda v = new Venda();
121
                Cliente c = new Cliente();
122
                Cidade ci = new Cidade();
123
                Estado e = new Estado();
124
125
                v.setId( rs.getLong( "idVenda" ) );
126
                v.setData( rs.getDate( "dataVenda" ) );
127
                v.setCancelada( rs.getBoolean( "vendaCancelada" ) );
128
                v.setCliente( c );
129
130
                c.setId( rs.getLong( "idCliente" ) );
131
                c.setNome( rs.getString( "nomeCliente" ) );
132
                c.setSobrenome( rs.getString( "sobrenomeCliente" ) );
133
                c.setDataNascimento( rs.getDate( "dataNascimentoCliente" ) );
134
                c.setCpf( rs.getString( "cpfCliente" ) );
135
                c.setEmail( rs.getString( "emailCliente" ) );
136
                c.setLogradouro( rs.getString( "logradouroCliente" ) );
137
                c.setNumero( rs.getString( "numeroCliente" ) );
138
                c.setBairro( rs.getString( "bairroCliente" ) );
139
                c.setCep( rs.getString( "cepCliente" ) );
140
                c.setCidade( ci );
141
142
                ci.setId( rs.getLong( "idCidade" ) );
                ci.setNome( rs.getString( "nomeCidade" ) );
144
                ci.setEstado( e );
145
146
                e.setId( rs.getLong( "idEstado" ) );
147
                e.setNome( rs.getString( "nomeEstado" ) );
148
                e.setSigla( rs.getString( "siglaEstado" ) );
149
```

```
150
                 lista.add( v );
151
152
            }
153
154
            rs.close();
155
            stmt.close();
156
157
            return lista;
158
159
        }
160
161
        @Override
162
        public Venda obterPorId( Long id ) throws SQLException {
163
164
            Venda venda = null;
165
166
            PreparedStatement stmt = getConnection().prepareStatement(
167
                      0.00
168
                     SELECT
169
                          v.id idVenda,
170
                          v.data dataVenda,
171
172
                          v.cancelada vendaCancelada,
                          c.id idCliente,
173
                          c.nome nomeCliente,
174
                          c.sobrenome sobrenomeCliente,
175
                          c.data_nascimento dataNascimentoCliente,
176
177
                          c.cpf cpfCliente,
                          c.email emailCliente,
178
                          c.logradouro logradouroCliente,
179
                          c.numero numeroCliente,
180
                          c.bairro bairroCliente,
181
182
                          c.cep cepCliente,
                          ci.id idCidade,
183
                          ci.nome nomeCidade,
184
185
                          e.id idEstado,
                          e.nome nomeEstado,
186
                          e.sigla siglaEstado
187
                     FROM
188
                          venda v,
189
                          cliente c,
190
                          cidade ci,
191
```

```
192
                         estado e
                     WHERE
193
                         v.id = ? AND
194
                         v.cliente_id = c.id AND
195
                         c.cidade id = ci.id AND
196
                         ci.estado id = e.id;
197
                     """ );
198
199
            stmt.setLong( 1, id );
200
201
            ResultSet rs = stmt.executeQuery();
202
203
            if ( rs.next() ) {
204
205
                venda = new Venda();
206
                Cliente c = new Cliente();
207
                Cidade ci = new Cidade();
208
209
                Estado e = new Estado();
210
                venda.setId( rs.getLong( "idVenda" ) );
211
                venda.setData( rs.getDate( "dataVenda" ) );
212
                venda.setCancelada( rs.getBoolean( "vendaCancelada" ) );
213
214
                venda.setCliente( c );
215
                c.setId( rs.getLong( "idCliente" ) );
216
                c.setNome( rs.getString( "nomeCliente" ) );
217
                c.setSobrenome( rs.getString( "sobrenomeCliente" ) );
218
                c.setDataNascimento( rs.getDate( "dataNascimentoCliente" ) );
219
                c.setCpf( rs.getString( "cpfCliente" ) );
220
                c.setEmail( rs.getString( "emailCliente" ) );
221
                c.setLogradouro( rs.getString( "logradouroCliente" ) );
222
                c.setNumero( rs.getString( "numeroCliente" ) );
223
                c.setBairro( rs.getString( "bairroCliente" ) );
224
                c.setCep( rs.getString( "cepCliente" ) );
225
                c.setCidade( ci );
226
227
                ci.setId( rs.getLong( "idCidade" ) );
228
                ci.setNome( rs.getString( "nomeCidade" ) );
229
                ci.setEstado( e );
230
231
                e.setId( rs.getLong( "idEstado" ) );
232
                e.setNome( rs.getString( "nomeEstado" ) );
233
```

```
e.setSigla( rs.getString( "siglaEstado" ) );
234
235
             }
236
237
             rs.close();
238
             stmt.close();
240
241
             return venda;
242
        }
243
244
245 }
```

Já o ItemVendaDAO é exibido na Listagem 8.10.

```
Listagem 8.10: Código da classe "ItemVendaDAO"
   Arquivo: vendaprodutos/dao/ItemVendaDAO.java
package vendaprodutos.dao;
2
3 import java.sql.PreparedStatement;
4 import java.sql.ResultSet;
5 import java.sql.SQLException;
6 import java.util.ArrayList;
7 import java.util.List;
8 import vendaprodutos.entidades.ItemVenda;
  import vendaprodutos.entidades.Produto;
10
11
12
   * DAO para a entidade ItemVenda.
13
   * @author Prof. Dr. David Buzatto
14
   */
15
  public class ItemVendaDAO extends DAO<ItemVenda> {
16
17
18
       public ItemVendaDAO() throws SQLException {
19
20
       @Override
21
       public void salvar( ItemVenda obj ) throws SQLException {
22
23
```

```
PreparedStatement stmt = getConnection().prepareStatement(
24
25
                   INSERT INTO
26
                    item_venda( venda_id, produto_id, valor, quantidade )
27
                   VALUES( ?, ?, ?, ?);
28
                    """);
29
           stmt.setLong( 1, obj.getVenda().getId() );
31
           stmt.setLong( 2, obj.getProduto().getId() );
32
           stmt.setBigDecimal( 3, obj.getValor() );
33
           stmt.setBigDecimal( 4, obj.getQuantidade() );
34
35
           stmt.executeUpdate();
           stmt.close();
37
38
       }
39
40
       @Override
41
       public void atualizar( ItemVenda obj ) throws SQLException {
           // não faz sentido na nossa implementação,
43
           // pois não é possível atualizar um item
44
           // da venda armazenado
45
       }
46
47
       @Override
       public void excluir( ItemVenda obj ) throws SQLException {
49
           // não faz sentido na nossa implementação,
50
           // pois não é possível excluir um item
51
           // da venda armazenado
52
       }
54
       @Override
55
       public List<ItemVenda> listarTodos() throws SQLException {
56
57
           // nesse caso, não há sentido haver uma listagem por todos
           // os itens de venda, visto que essa entidade é uma entidade
           // de ligação e não faremos atualização em vendas
           // já realizadas, a não ser o cancelamento, mas isso trataremos
61
           // usando o método obterPorIdVenda implementado abaixo.
62
           return null;
63
64
       }
65
```

```
66
       @Override
67
       public ItemVenda obterPorId( Long id ) throws SQLException {
68
69
            // o identificador dessa entidade é composto!
70
            // precisamos ter um método especializado se fosse
71
            // necessário.
72
            return null;
73
74
       }
75
76
        /**
77
         * Obtenção de todos os itens de venda por um identificador da venda.
78
         * Esse método será utilizado para o ajuste do estoque das vendas
79
         * que forem canceladas. Apenas os valores necessários serão obtidos.
80
81
       public List<ItemVenda> obterPorIdVenda(Long idVenda) throws SQLException {
82
83
            List<ItemVenda> itensVenda = new ArrayList<>();
84
85
            PreparedStatement stmt = getConnection().prepareStatement(
86
87
                    SELECT
88
                         iv.quantidade quantidadeItemVenda,
89
                        p.id idProduto,
90
                        p.estoque estoqueProduto
91
                    FROM
92
                        item_venda iv,
93
                        produto p
94
                    WHERE iv.produto id = p.id AND
95
                          iv.venda_id = ?;
96
                    """);
97
98
            stmt.setLong( 1, idVenda );
99
100
101
            ResultSet rs = stmt.executeQuery();
102
            while ( rs.next() ) {
103
104
                ItemVenda iv = new ItemVenda();
105
                Produto p = new Produto();
106
107
```

```
iv.setQuantidade( rs.getBigDecimal( "quantidadeItemVenda" ) );
108
                 iv.setProduto( p );
109
110
                 p.setId( rs.getLong( "idProduto" ) );
111
                 p.setEstoque( rs.getBigDecimal( "estoqueProduto" ) );
112
113
                 itensVenda.add( iv );
114
115
            }
116
117
             rs.close();
118
119
             stmt.close();
120
            return itensVenda;
121
122
        }
123
124
125
   }
```

Nesse DAO temos a implementação do método salvar(...), mas a atualização, a exclusão, a obtenção de todos os itens de venda e a obtenção por id não são implementadas, visto que nunca mexeremos numa venda após ser feita, mas como poderemos cancelar uma venda, implementamos um método adicional no ItemVendaDAO chamado de obterPorIdVenda, definido a partir da linha 82. Esse método retornará todos os itens de uma determinada venda para que, ao haver a necessidade de cancelála, possamos atualizar o estoque dos produtos, pois se uma venda foi cancelada, os produtos deverão ser devolvidos à loja e voltarão a compor o estoque.

Vamos agora fechar a implementação do nosso *back-end* detalhando o Servlet das vendas. Na Listagem 8.11 pode ser visto o código completo do mesmo.

```
Listagem 8.11: Código-fonte do Servlet "VendasServlet"
Arquivo: vendaprodutos/controladores/VendasServlet.java

package vendaprodutos.controladores;

import java.io.IOException;
import java.io.PrintWriter;
import java.io.StringReader;
import java.math.BigDecimal;
import java.sql.Date;
```

```
import java.sql.SQLException;
9 import java.time.LocalDate;
import jakarta.json.Json;
import jakarta.json.JsonArray;
import jakarta.json.JsonObject;
import jakarta.json.JsonReader;
import jakarta.json.JsonValue;
import jakarta.servlet.RequestDispatcher;
import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
20 import jakarta.servlet.http.HttpServletResponse;
21 import vendaprodutos.dao.ClienteDAO;
22 import vendaprodutos.dao.ItemVendaDAO;
23 import vendaprodutos.dao.ProdutoDAO;
24 import vendaprodutos.dao.VendaDAO;
25 import vendaprodutos.entidades.Cliente;
26 import vendaprodutos.entidades.ItemVenda;
27 import vendaprodutos.entidades.Produto;
  import vendaprodutos.entidades.Venda;
  import vendaprodutos.utils.Utils;
29
30
  /**
31
   * Servlet para tratar Vendas.
32
33
   * @author Prof. Dr. David Buzatto
34
35
  @WebServlet( name = "VendasServlet",
36
                urlPatterns = { "/processaVendas" } )
37
  public class VendasServlet extends HttpServlet {
39
       protected void processRequest(
40
               HttpServletRequest request,
41
               HttpServletResponse response )
42
               throws ServletException, IOException {
43
44
           String acao = request.getParameter( "acao" );
45
           RequestDispatcher disp = null;
46
47
           try ( VendaDAO daoVenda = new VendaDAO();
48
                 ClienteDAO daoCliente = new ClienteDAO();
49
```

```
ItemVendaDAO daoItemVenda = new ItemVendaDAO();
                 ProdutoDAO daoProduto = new ProdutoDAO() ) {
51
52
               if ( acao.equals( "inserir" ) ) {
53
54
                   Long idCliente = Utils.getLong( request, "idCliente" );
                   String itensVenda = request.getParameter( "itensVenda" );
57
                   // cria um leitor de json para processar os
58
                   // itens da venda
59
                   JsonReader jsr = Json.createReader(
60
                           new StringReader( itensVenda ) );
61
                   // faz a leitura/parse
                   JsonArray jsaItensVenda = jsr.readArray();
63
64
                   Cliente c = daoCliente.obterPorId( idCliente );
65
66
                   Venda v = new Venda();
67
                   v.setData( Date.valueOf( LocalDate.now() ) );
                   v.setCancelada( false );
69
                   v.setCliente( c );
70
71
                   Utils.validar( v, "id" );
72
                   daoVenda.salvar( v );
73
                   // itera pelos itens da venda genéricos
75
                   for ( JsonValue jsv : jsaItensVenda ) {
76
77
                        // sabemos que cada item é um objeto
78
                        JsonObject jso = jsv.asJsonObject();
                        // extraímos os atributos
81
                       Long idProduto = Utils.getLong(
82
                                jso.getString( "idProduto" ) );
83
                        BigDecimal quantidade = Utils.getBigDecimal(
                                jso.getString( "quantidade" ) );
                        // obtém o produto e atualiza o estoque
87
                        Produto p = daoProduto.obterPorId( idProduto );
88
                        p.setEstoque( p.getEstoque().subtract( quantidade ) );
89
90
                        // cria um item da venda
91
```

```
ItemVenda iv = new ItemVenda();
92
                         iv.setVenda( v );
93
                         iv.setProduto( p );
94
                         iv.setValor( p.getValorVenda() );
95
                         iv.setQuantidade( quantidade );
96
97
                         // não validaremos o produto, pois
                         // permitiremos estoque negativo na venda
99
                         daoProduto.atualizar( p );
100
                         daoItemVenda.salvar( iv );
101
102
                    }
103
104
                     disp = request.getRequestDispatcher(
105
                             "/formularios/vendas/listagem.jsp" );
106
107
                } else if ( acao.equals( "cancelar" ) ) {
108
109
                    Long id = Utils.getLong( request, "id" );
110
111
                    Venda v = daoVenda.obterPorId( id );
112
                    v.setCancelada( true );
113
114
                     daoVenda.atualizar( v );
115
                     for ( ItemVenda iv : daoItemVenda.obterPorIdVenda( id ) ) {
116
                         Produto p = iv.getProduto();
117
                         p.setEstoque( p.getEstoque().add( iv.getQuantidade() ) );
118
                         daoProduto.atualizarEstoque( p );
119
                    }
120
                    response.setContentType( "application/json; charset=UTF-8" );
122
123
124
                     JsonObject jo = Json.createObjectBuilder()
                             .add( "status", "ok" )
125
                             .build();
126
127
                     try ( PrintWriter out = response.getWriter() ) {
128
                         out.print( jo );
129
                     }
130
131
                }
132
133
```

```
} catch ( SQLException exc ) {
134
                 disp = Utils.prepararDespachoErro( request, exc.getMessage() );
135
            }
136
137
            if ( disp != null ) {
138
                 disp.forward( request, response );
139
            }
140
141
        }
142
143
        @Override
144
145
        protected void doGet(
                 HttpServletRequest request,
146
                 HttpServletResponse response )
147
                 throws ServletException, IOException {
148
            processRequest( request, response );
149
        }
150
151
        @Override
152
        protected void doPost(
153
                 HttpServletRequest request,
154
                 HttpServletResponse response )
155
                 throws ServletException, IOException {
156
            processRequest( request, response );
157
        }
158
159
        @Override
160
        public String getServletInfo() {
161
            return "VendasServlet";
162
        }
163
164
   }
165
```

Esse Servlet tratará dois tipos de ação. Uma de inserção, entre as linhas 53 e 107 e uma de cancelamento, entre as linhas 108 e 130. A inserção de uma venda envolve a associação do cliente que está comprando do estabelecimento, a data da mesma e todos os itens que a compõe. Veja que a variável itensVenda é uma String que conterá dados codificados em JSON que virão do cliente. Precisaremos processar esse JSON do lado do servidor parar criar um objeto genérico que conterá o ou os identificadores do produtos e a ou as respectivas quantidades vendidas. O processo de construção desse JSON será visto quando formos tratar sobre o lado do cliente. Veremos isso logo.

Veja que o processamento do JSON dos itens da venda é feito inicialmente criando um objeto do tipo JsonReader nas linhas 60 e 61. Posteriormente, na linha 63 o processo de *parsing* do JSON é feito, atribuindo o resultado à variável <code>jsaItensVenda</code> do tipo JsonArray. Após salvarmos a venda na linha 73, iteraremos sobre <code>jsaItensVenda</code> para extrairmos cada objeto genérico com os dados que precisamos para "amarrar" os produtos vendidos na venda realizada. Essa iteração ocorre entre as linhas 76 e 103, onde inicialmente obtemos o objeto genérico atual (linha 79), extraímos os dados necessários entre as linhas 82 e 85 e na linha 88 consultamos o produto e atualizamos o seu estoque (só no objeto por enquanto), visto que como estamos vendendo, precisamos subtrair a quantidade do estoque. Entre as linhas 92 e 96 criamos o item da venda, na linha 100 atualizamos o produto por causa do estoque (agora no banco de dados) e na linha 101 salvamos o item da venda. Não precisamos validar os itens de venda nem os produtos, visto que todo esse processamento será feito internamente.

O processo de cancelamento, tratado entre as linhas 108 e 130, atualizará a venda, marcando-a como cancelada e atualizará os estoques dos produtos associados. Note que não criaremos um RequestDispatcher, pois a ação de cancelamento será executada via AJAX. Não trataremos situações em que possam haver problemas no SGBD, pois após o cancelamento de uma venda retornaremos um JSON sinalizando que a requisição foi bem sucedida, mas em um sistema mais robusto teríamos que pensar nisso também. Uma outra coisa a se pensar, mas que não foi endereçada na nossa implementação, seria o caso de haver algum erro durante a inserção dos itens de venda ou na atualização dos produtos no cancelamento. Isso seria resolvido configurando o driver do SGBD para que as transações fossem gerenciadas manualmente e iniciadas antes das operações dos DAOs e finalizadas explicitamente com um *commit* caso tudo ocorresse como esperado ou canceladas (*rollback*) na detecção de algum problema.

Algo importante que não lidamos na nossa implementação é o caso de algum problema acontecer durante o processo de atualização vários tantos registros na base de dados. Imagine se o primeiro item de venda foi processado corretamente, mas no segundo, por algum motivo, aconteceu algum erro como o servidor perdeu a comunicação com o SGBD. Se isso acontecer, teremos dados corrompidos, pois o que deveria ter sido feito completamente foi feito parcialmente, concorda? Por exemplo, no caso de inserção/cadastro, o ideal seria iniciarmos uma transação antes de salvar a venda e dar um "commit" nessa transação antes encaminharmos a requisição para a página de listagem, ou dar um "rollback" em caso de algum problema acontecer, desfazendo o que foi feito desde o início da transação, evitando problemas na estrutura da base de dados que foi modificada. Novamente, para simplificarmos um pouco mais a implementação isso não foi endereçado. Trataremos esse tipo de situação no Capítulo de persistência, tudo bem?

Vamos agora tratar o lado do cliente.

Front-End

Começaremos conferindo como a listagem das vendas foi implementada. Basicamente ela consiste na mesma estrutura dos outros cadastros, ou seja, uma em que cada linha há um registro da tabela do banco de dados em questão. Estamos fazendo a construção dessa tabela a partir das classes de serviços, você se lembra? A novidade aqui é que uma venda não pode ser excluída ou atualizada, com a exceção de que ela pode ser cancelada. O cancelamento será feito através de uma chamada em AJAX. Para implementarmos isso, veja o <c:choose> construído entre as linhas 60 e 69 da Listagem 8.12.

```
Listagem 8.12: Código da listagem de Vendas
   Arquivo: /formularios/vendas/listagem.jsp
  %@page contentType="text/html" pageEncoding="UTF-8"%>
   %@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
3 <%@taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
4 <c:set var="cp" value="${pageContext.request.contextPath}"/>
5 <!DOCTYPE html>
6
   <html>
7
     <head>
8
9
       <title>Vendas Cadastradas</title>
10
11
       <meta charset="UTF-8">
12
       <meta name="viewport"
13
             content="width=device-width, initial-scale=1.0">
14
15
       <link rel="stylesheet"</pre>
16
             href="${cp}/css/estilos.css"/>
17
18
       <script src="${cp}/js/libs/jquery/jquery.min.js"></script>
19
       <script src="${cp}/js/formularios/vendas/listagem.js"></script>
20
21
     </head>
22
23
     <body>
24
25
       <h1>Vendas Cadastradas</h1>
26
```

```
27
      >
28
        <a href="${cp}/formularios/vendas/novo.jsp">
29
         Nova Venda
30
        </a>
31
      32
33
      34
        <thead>
35
          36
           Id
37
           Data
38
           Cliente
39
           Cancelar
40
          41
        </thead>
42
        43
44
          <jsp:useBean
45
             id="servicos"
46
             scope="page"
47
             class="vendaprodutos.servicos.VendaServices"/>
48
49
          <c:forEach items="${servicos.todos}" var="venda">
50
           51
             ${venda.id}
52
             >
53
               <fmt:formatDate
54
                 pattern="dd/MM/yyyy"
55
                 value="${venda.data}"/>
56
             57
             ${venda.cliente.nome} ${venda.cliente.sobrenome}
58
             59
               <c:choose>
60
                 <c:when test="${venda.cancelada}">
61
62
                   Cancelada
                 </c:when>
63
                 <c:otherwise>
64
                   <a href="#" data-id="${venda.id}"
65

    onclick="cancelarVenda(event,'${cp}')">

                    Cancelar
66
                   </a>
67
```

```
</c:otherwise>
68
                </c:choose>
69
              70
            71
          </c:forEach>
72
        73
74
      75
76
      >
77
        <a href="${cp}/formularios/vendas/novo.jsp">
78
79
          Nova Venda
        </a>
      81
82
      <a href="${cp}/index.jsp">Tela Principal</a>
83
84
85
    </body>
86
  </html>
87
```

Caso uma venda esteja cancelada, a palavra "Cancelada" aparecerá na coluna de cancelamento. Caso contrário, entre as linhas 65 e 67, será gerado um *link* apontando para "#" no atributo href, sendo que o sinal "#" sozinho indica que é uma âncora para lugar nenhum, ou seja, quando o link for clicado, ele não fará nada. Outras tags poderiam ter sido usadas aqui, mas para manter a consistência na GUI em relação aos outros cadastros, optei por usar hyperlinks mesmo. Nosso link terá mais dois atributos definidos. Um você já conhece, que é o onclick, apontando para a função cancelar Venda (...) O outro é o atributo data. Esse atributo é interessante pois podemos armazenar dados nas tags! Para usá-lo, começamos com a palavra data, inserimos um traço e depois um identificador qualquer, sem espaços ou letras maiúsculas. Caso queira um identificador com nome composto, ou seja, com mais de uma palavra, separe-as por traços. No nosso caso, o identificador é id. A ideia é que esse *link* tenha um dado associado que é o identificador da venda. A função cancelarVenda(...) usará esse dado para saber qual venda deve ser cancelada. Note que também poderíamos ter passado o identificador da venda como argumento para a função, mas eu quis usar o atributo data para vocês saberem que ele existe.

(i) | Saiba Mais

Mais sobre os atributos data: https://developer.mozilla.org/en-US/docs/Learn/HTML/Howto/Use_data_attributes>

Vamos dar uma olhada no arquivo de *script* em que a função cancelar Venda (...) está implementada. Esse arquivo pode ser visto na Listagem 8.13.

```
Listagem 8.13: Script para cancelamento de Vendas
   Arquivo: /js/formularios/vendas/listagem.js
   function cancelarVenda( event, cp ) {
2
       if ( confirm( "Deseja mesmo cancelar essa venda?" ) ) {
3
4
           let id = event.target.dataset.id;
5
6
           let parametros = new URLSearchParams();
7
           parametros.append( "acao", "cancelar" );
8
           parametros.append( "id", id );
9
10
           fetch( `${cp}/processaVendas`, {
11
               method: "POST",
12
               body: parametros
13
           }).then( response => {
14
               return response.json();
15
           }).then( data => {
16
17
               if ( data.status === "ok" ) {
18
                    event.target.parentElement.innerHTML = "Cancelada";
19
               } else {
20
                   alert( "Ocorreu um erro na sua requisição!" );
21
               }
22
23
           }).catch( error => {
24
               alert( "Erro: " + error );
25
           });
26
27
           // jQuery
28
           /*$.ajax( `${cp}/processaVendas`, {
29
               data: {
30
                   acao: "cancelar",
31
```

```
id: id
32
                },
33
                dataType: "json"
34
            }).done( ( data, textStatus ) =>{
35
36
                if ( data.status ) {
37
                    $( event.target ).parent().html( "Cancelada" );
                } else {
39
                    alert( "Ocorreu um erro na sua requisição!" );
40
                7
41
42
           }).fail( ( jqXHR, textStatus, errorThrown ) => {
43
                alert( "Erro: " + errorThrown + "\n" +
44
                        "Status: " + textStatus );
45
           }):*/
46
47
       }
48
49
   }
50
```

Dentro da função temos a implementação da requisição AJAX feita tanto usando JavaScript puro, empregando a Fetch API entre as linhas 7 e 26, quanto usando jQuery, implementação essa que está comentada e é encontrada entre as linhas 29 e 46. Na linha 5 obtemos o identificador da venda através do atributo dataset, que é quem referencia os atributos data definidos no código HTML. Veja que acessamos o atributo target do evento, que representa quem disparou o evento, no caso o *link* que foi clicado, acessamos o atributo dataset do *link* e, por sua vez, o atributo id do dataset, que é quem está armazenando o dado que queremos, ou seja, o identificador da venda armazenado no banco de dados. Novamente, como já foi dito, poderíamos alternativamente ter passado o identificador da venda em um parâmetro dessa função.

Nossa requisição AJAX aponta para o Servlet de vendas, mapeado na URL

/processaVendas. Note que passamos o contexto da aplicação para a função, evitando problemas de referências quebradas. Veja que quando a requisição for feita, a ação "cancelar" será processada no Servlet, atualizando a coluna do registro que indica que a venda está cancelada, além de retornar ao estoque do ou dos produtos vendidos as suas respectivas quantidades. Note que em caso de sucesso, o JSON trazido do servidor será processado na linha 15 e então o objeto criado, chamado de data, será utilziado para verificar o status, que no caso só poderá ser "ok", e o conteúdo da célula da tabela onde a tag <a> existia (parentElement) será atualizado para "Cancelada", não permitindo mais ser clicado, pois a partir desse momento não haverá mais um

link naquela posição.

Agora trataremos o nosso formulário de cadastro de uma nova venda. Esse formulário é um pouco mais elaborado que os anteriores, sendo que podemos ver seu código completo na Listagem 8.14.

```
Listagem 8.14: Formulário de cadastro de novas Vendas
   Arquivo: /formularios/vendas/novo.jsp
1 < 00 page contentType="text/html" pageEncoding="UTF-8"%>
2 < 0 cm/jsp/jstl/core %>
3 <%@taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
4 <c:set var="cp" value="${pageContext.request.contextPath}"/>
5 <!DOCTYPE html>
6
  <html>
7
8
    <head>
9
       <title>Nova Venda</title>
10
11
       <meta charset="UTF-8">
12
       <meta name="viewport"</pre>
13
             content="width=device-width, initial-scale=1.0">
14
15
16
       <link rel="stylesheet"</pre>
             href="${cp}/css/estilos.css"/>
17
18
       <script src="${cp}/js/libs/jquery/jquery.min.js"></script>
19
       <script src="${cp}/js/libs/decimal.js/decimal.min.js"></script>
20
       <script src="${cp}/js/formularios/vendas/novo.js"></script>
21
22
     </head>
23
24
     <body>
25
26
       <h1>Nova Venda</h1>
27
28
       <form id="formNovaVenda" method="post" action="${cp}/processaVendas">
29
30
         <input name="acao" type="hidden" value="inserir"/>
31
         <input id="hiddenItensVenda" name="itensVenda" type="hidden"/>
32
33
```

```
<div id="divCliente">
34
           <jsp:useBean
35
               id="servicosC"
36
               scope="page"
37
               class="vendaprodutos.servicos.ClienteServices"/>
38
39
           Cliente:
           <br/>
41
           <select id="selectCliente" name="idCliente" required>
42
             <c:forEach items="${servicosC.todos}" var="cliente">
43
               <option value="${cliente.id}">
44
                 ${cliente.nome} ${cliente.sobrenome}
45
               </option>
             </c:forEach>
47
           </select>
48
         </div>
49
50
         <div id="divItensVenda">
51
           53
               54
55
                 <jsp:useBean
56
                      id="servicosP"
57
                      scope="page"
                      class="vendaprodutos.servicos.ProdutoServices"/>
59
60
                 >
61
                   Produto:
62
                    <br/>
                    <select id="selectProduto">
                      <c:forEach items="${servicosP.todos}" var="produto">
65
66
                        <fmt:formatNumber
67
                            pattern="#.##"
                            minIntegerDigits="1"
                            minFractionDigits="2"
70
                            maxFractionDigits="2"
71
                            var="valorVenda"
72
                            scope="page"
73
                            value="${produto.valorVenda}"/>
74
75
```

```
<option value="${produto.id}"</pre>
76
                               data-valor="${valorVenda}"
77
                               data-descricao="${produto.descricao}">
78
                         ${produto.descricao}
79
                         (R$ ${valorVenda}
80
81
                         ${produto.unidadeMedida.sigla})
82
                       </option>
83
                     </c:forEach>
84
                   </select>
85
                 86
87
                 >
88
                 Quantidade:
89
                 <br/>
90
                 <input id="txtQuantidade"</pre>
91
                        type="number"
92
                        size="3"
93
                        placeholder="9,99"
94
                        step="0.01"
95
                        min="0"/>
96
                 97
98
               99
               100
                 <input id="btnInserir" type="button" value="&#x2795;"/>
101
                 <input id="btnRemover" type="button" value="&#x2796;"/>
102
103
                 <input id="btnLimpar" type="button" value="&#x274C;"/>
               104
               105
                 Itens da Venda:
106
                 <br/>
107
108
                 <select id="selectItensVenda" size="10" multiple>
                 </select>
109
                 <br/>
110
                 <div>
111
                   <div id="divTotal">Total: R$ 0,00</div>
112
                 </div>
113
               114
             115
116
             117
```

```
118
              <
119
              120
               <input id="btnSalvar" type="submit" value="Salvar"/>
121
122
            123
          </div>
125
126
        <a href="${cp}/formularios/vendas/listagem.jsp">
127
          Voltar
128
        </a>
129
130
      </form>
131
132
    </body>
133
134
   </html>
135
```

Graficamente, o código acima gerará algo como apresentado na Figura 8.3.

Figura 8.3: Formulário para Novas Vendas



Fonte: Elaborada pelo autor

Você já conhece praticamente todo o código utilizado na Listagem 8.14, sendo assim focaremos na implementação da funcionalidade apresentado na Listagem 8.15.

```
Listagem 8.15: Script para tratamento de novas Vendas
   Arquivo: /js/formularios/vendas/novo.js
   /**
1
   * Implementação das funções do formulário de venda.
2
3
4
   // Document ready (quando o documento estiver pronto)
   $(() => {
       // array para armazenar os itens da venda
8
       let itensVenda = [];
9
10
       // formatadores
11
       let fmtMoeda = new Intl.NumberFormat(
12
           "pt-BR", {
13
               style: "currency",
14
               currency: "BRL"
15
           }
16
       );
17
18
       let fmtNumero = new Intl.NumberFormat(
19
           "pt-BR", {
20
21
               minimumFractionDigits: 2,
               maximumFractionDigits: 2
22
           }
23
       );
24
25
       // ao clicar no botão inserir
26
       $( "#btnInserir" ).on( "click", event => {
27
28
           let $selectProduto = $( "#selectProduto" );
29
           let $txtQuantidade = $( "#txtQuantidade" );
30
31
           let idProduto = $selectProduto.val();
32
           let valorVenda = $selectProduto.find( ":selected" ).data( "valor"
33
            → ).toString();
           let descricao = $selectProduto.find( ":selected" ).data("descricao");
34
           let quantidade = null;
35
```

```
36
           // se o valor da venda tem vírgula, troca por ponto
37
           if ( valorVenda.includes( "," ) ) {
               valorVenda = valorVenda.replace( ",", "." );
39
           }
40
41
           try {
               quantidade = new Decimal( $txtQuantidade.val() );
43
           } catch ( e ) {
44
45
46
           if ( quantidade !== null && quantidade.greaterThan( 0 ) ) {
47
               // há um item da venda iqual?
               let itemIgual = null;
50
               itensVenda.some( item => {
51
                    if ( item.idProduto === idProduto ) {
52
                        itemIgual = item;
53
                        return true; // para a iteração
                    }
55
               });
56
57
               // se há item igual, atualiza
58
               if ( itemIgual !== null ) {
59
                    // soma a quantidade
61
                    itemIgual.quantidade = itemIgual
62
                            .quantidade
63
                             .plus( quantidade );
64
                    // caso contrário, cria um novo item
               } else {
67
                    itensVenda.push({
68
                        idProduto: idProduto,
69
                        valorVenda: valorVenda,
70
                        descricao: descricao,
                        quantidade: quantidade
72
                    });
73
               }
74
75
               atualizarGUI();
76
               $txtQuantidade.val( "" );
77
```

```
78
            } else {
79
                 alert( "Forneça uma quantidade maior que zero!" );
80
81
82
        });
83
84
        // ao clicar no botão remover
85
        $( "#btnRemover" ).on( "click", event => {
86
87
            // retorna um array com os values de todos os itens
88
            // (option) selecionados
89
            let selecao = $( "#selectItensVenda" ).val();
90
91
            // se não selecionou nenhum
92
            if ( selecao.length === 0 ) {
93
                alert( "Selecione um item da venda para remover!" );
94
95
                //se há seleção
            } else if ( confirm( "Deseja remover o(s) item(ns) da venda
97

    selecionado(s)?" ) ) {
98
                 // itera pela seleção
99
                for ( let i = 0; i < selecao.length; i++ ) {</pre>
100
101
                     // busca sequencial nos itens de venda
102
                     for ( let j = 0; j < itensVenda.length; <math>j++ ) {
103
104
                         let item = itensVenda[j];
105
                         // encontrou?
107
                         if ( selecao[i] === item.idProduto ) {
108
109
                              // remove da posição j
110
                              itensVenda.splice( j, 1 );
111
                              break;
112
113
                         }
114
115
                     }
116
117
                }
118
```

```
119
                 // remonta a lista
120
                 atualizarGUI();
121
122
            }
123
124
        });
125
126
        // ao clicar no botão limpar
127
        $( "#btnLimpar" ).on( "click", event => {
128
            if ( confirm( "Deseja remover todos os itens da venda?" ) ) {
129
                 itensVenda = [];
130
                 atualizarGUI();
131
            }
132
        });
133
134
        // submissão da venda
135
        $( "#formNovaVenda" ).on( "submit", event => {
136
137
            if ( $( "#selectItensVenda > option" ).length === 0 ) {
138
                 alert( "Uma venda precisa conter pelo menos um item!" );
139
                 return false;
140
            }
141
142
143
            return true;
144
        });
145
146
        // evita que ao teclar enter dentro do campo
147
        // de texto o formulário seja submetido
148
        $( "#txtQuantidade" ).on( "keydown", event => {
149
            if ( event.keyCode === 13 ) {
150
151
                 event.preventDefault();
            }
152
        });
153
154
        // constrói as opções do <select> (lista) de itens de venda;
155
        // atualiza o valor total da venda;
156
        // e prepara os dados para envio
157
        let atualizarGUI = () => {
158
159
            let $select = $( "#selectItensVenda" );
160
```

```
161
            let total = new Decimal( 0 );
162
            $select.html( "" );
163
164
            itensVenda.forEach( item => {
165
166
                let valorItem = new Decimal( item.valorVenda )
167
                                       .times( item.quantidade );
168
169
                 $opt = $( "<option></option>" ).
170
                         html( `${item.descricao} - ` +
171
                         `${fmtMoeda.format( item.valorVenda )} x ` +
172
                         `${fmtNumero.format( item.quantidade )} = ` +
173
                         `${fmtMoeda.format( valorItem )}` ).
174
                         val( `${item.idProduto}` );
175
176
                 $select.append( $opt );
177
                 total = total.plus( valorItem );
178
179
            });
180
181
            $( "#divTotal" ).html( "Total: " + fmtMoeda.format( total ) );
182
            $( "#hiddenItensVenda" ).val( JSON.stringify( itensVenda ) );
183
184
        };
185
186
187 });
```

Neste formulário optei por fazer o registro de todos os eventos programaticamente. Na linha 6 usamos um dialeto padrão da jQuery em que registramos uma função, passada como argumento para a função \$(...), como ouvinte do "evento" onready. Esse evento na verdade é uma construção da biblioteca jQuery que define que quando o documento está pronto, ou seja, todo o HTML já passou pelo processo de *parsing* pelo navegador, a função será executada, o que é diferente do evento onload da *tag* \$\left\(\text{body} \right\) que é disparado quando todo o HTML e todos os recursos como imagens, arquivos externos etc. forem carregados. Note que essa função contém todas as outras que vamos usar.

Na linha 9 criamos um array que armazenará todos os itens da venda que iremos montar durante o uso do formulário. Entre as linhas 12 e 24 criamos dois formatadores de números que serão usados para formatar apropriadamente valores monetários e valores numéricos em ponto flutuante.

Entre as linhas 27 e 83 temos o evento onclick do botão inserir que, na Figura 8.3, pode ser visto entre a caixa de seleção dos produtos e a lista de itens da venda. O valor desse botão é "➕" que representa o emoji com sinal de mais/adição. O código em hexadecimal desse símbolo é 2795, sendo que o prefixo #x indica que é um valor em Unicode codificado em hexadecimal que deve ser processado pelo navegador e convertido em um caractere especial. Essa indicação de codificação de caractere é interpretada pelo navegador ao se iniciar tal String com & e terminá-la com ponto e vírgula. Note que o mesmo é aplicado aos botões com o sinal de menos/subtração (botão remover) e com o símbolo de um X vermelho (botão limpar).

(i) | Saiba Mais

A lista completa dos emojis do Unicode pode ser vista nesse *link*: <https://unicode.org/emoji/charts/full-emoji-list.html>

Para inserir um item de venda precisamos do produto e da quantidade, sendo assim obtemos os controles que detêm esses valores nas linhas 29 e 30. Entre as linhas 32 e 45 extraímos todos os dados que precisaremos para criar um objeto do item da venda. Veja que na linha 43 é criado um objeto do tipo Decimal. Esse tipo não é nativo do JavaScript, mas sim implementado na biblioteca decimal. js inserida no projeto via CDNJS. Essa biblioteca implementa tipos decimais de precisão arbitrária, assim como o tipo BigDecimal do Java. Como estamos lidando com quantidades de produtos do lado do cliente e não queremos arriscar ter algum tipo de erro ou problema relativo à precisão de números em ponto flutuante, vamos usar essa biblioteca. Entre as linhas 38 e 40 verificamos se o valor da venda, que é tratado como String (linha 33), contém uma vírgula separador decimal, o que provavelmente deve ser o seu caso, e caso seja, trocamos por um ponto para podermos criar um objeto do tipo Decimal quando for necessário.

Se a quantidade informada for um número maior que zero, entraremos no [if] da linha 47, ou seja, se uma quantidade válida foi informada o item da venda será criado ou a quantidade de um item existente será incrementada. Não faz sentido ter valor negativo em quantidades para o nosso problema, certo?

A primeira coisa que será feita é verificar se já existe um item de venda para o produto que está se tentando inserir na lista. Nosso banco de dados não permite que possa existir mais de um item de venda para uma mesma venda e um mesmo produto. Veja que no banco de dados a chave primária da tabela item_venda é composta pelas duas chaves estrangeiras, uma de venda e uma de produto. Uma tarefa desse Capítulo será tornar isso possível. Dada essa restrição, se já houver um item de venda com o produto que se está tentando inserir, o item de venda que foi inserido posteriormente será atualizado, ou seja, sua quantidade será incrementada com a nova quantidade

que se está tentando inserir. Para essa verificação, iteraremos sobre os itens da venda e, caso o identificador do produto do elemento atual for igual ao identificador do produto que se está tentando inserir na lista, a variável itemIgual receberá esse item e a iteração parará, por causa do retorno true do *callback* de some(...).

Na linha 59, caso o item da venda seja diferente de nulo, ou seja, foi encontrado um item da venda com o mesmo produto, atualiza-se a quantidade desse item de venda. Veja que não usamos simplesmente o operador de adição, visto que o valor do atributo quantidade é do tipo Decimal, havendo a necessidade de usar o método plus na quantidade do item, passando a quantidade a ser somada e, além disso, o retorno do método é atribuído à quantidade do item, visto que os objetos do tipo Decimal são imutáveis.

Se o produto que se está inserindo no novo item de venda não existe na lista, um novo item da lista será criado e inserido no array entre as linhas 68 e 73. Na linha 76 a lista de itens de venda é montada na GUI, baseando-se nos dados contidos no array itensVenda, além de outras atualizações na interface gráfica e na linha 77 o *input* da quantidade é resetado. A linha 80 contém um alerta que será mostrado caso a quantidade fornecida na inserção seja inválida.

Entre as linhas 86 e 125 temos a implementação da remoção de itens da lista. Perceba que esta lista pode ter mais de um item selecionado ao se clicar no botão de remoção, então precisamos tratar isso. Veja na linha 108 da Listagem 8.14 que o select de itens da venda pode ter seleção múltipla, pois contém o atributo booleano multiple. Essa seleção é feita na GUI mantendo pressionada a tecla CTRL do teclado ao se clicar nos itens.

Na linha 90 é obtido um array com todas as opções que estão selecionadas na lista. Se o tamanho desse array for zero, significa que não há itens selecionados, então uma mensagem é exibida (linha 94). Caso exista pelo menos um item, um diálogo de confirmação perguntará ao usuário se ele quer realmente remover os itens da venda selecionados. Perceba que não estamos conversando com o banco de dados! Caso o usuário escolha que os itens devem ser removidos, primeiramente precisamos iterarar pelos valores dos elementos selecionados (linha 100) e assim, para cada um desses valores, varrer o array de itens de venda (linha 103), procurando sequencialmente pelo produto e, caso encontrado, removendo esse item do array na linha 111. Ao terminar esse processo, a função atualizarGUI() é invocada, assim como na linha 76, para remontar a lista de itens da venda e atualizar a GUI, baseando-se no array de itens de venda.

Entre as linhas 128 e 133 temos a implementação do botão de limpar que, ao ser clicado, limpa a lista de itens de venda, ou seja, remove todos os itens de uma vez. Para isso, basta-se criar um novo array vazio para os itens da venda e, novamente,

8.3. RESUMO 307

atualizar a GUI. Já falaremos dessa montagem da lista!

Entre as linhas 136 e 145 implementamos a submissão do formulário que só pode ser feita se houver pelo menos um item de venda criado. Para isso, verificamos quantos elementos do tipo <option> existem dentro do <select> de itens da venda. Se houver pelo menos um, o formulário será submetido, visto o retorno true na linha 143. Caso contrário, dentro do if de verificação dessa quantidade, será retornado o valor false.

Outra coisa que trataremos é remover o comportamento padrão de submissão do formulário ao se teclar <ENTER> em um campo de texto. Como temos o campo de texto das quantidades, registramos o evento onkeydown nele e verificamos se a tecla que foi pressionada foi o <ENTER> que tem código 13. Se for o caso, informamos que o comportamento padrão do evento deve ser descartado, evitando a submissão do formulário.

A última coisa que precisamos conferir é a tal da função que monta os itens da venda na GUI e a atualiza. Ela está definida entre as linhas 158 e 185. Primeiramente obtémse o componente dos itens de venda da linha 160 e na linha 161 cria-se um acumulador para armazenar o valor total da venda. Limpa-se a lista na linha 163 e itera-se sobre os itens da venda entre as linhas 165 e 180. Nessa iteração, cada item de venda é usado para criar uma nova opção para o (select) dos itens de venda. Na linha 167 calcula-se o valor do item que é composto do valor do produto multiplicado pela quantidade, entre as linhas 170 e 175 é criado um novo item da lista, na linha 177 esse item é inserido e, na linha 178, o totalizador da venda é incrementado. Ao terminar esse processo, a lista já estará atualizada com os itens que refletem o array de itens de venda, faltando atualizar o total da venda, o que acontece na linha 182 e, na linha 183, o array de itens de venda é serializado em JSON para ser enviado na submissão do formulário usando o campo escondido chamado hiddenItensVenda. Veja que a serialização em JSON carregará dados que não são necessários como a descrição do produto e o valor da venda. Poderíamos enviar uma forma mais enxuta desses dados mantendo um array para os dados completos e um com somente para o que é necessário para o Servlet atuar, mas deixaremos dessa forma para não complicar mais do que o necessário nesse momento.

Pronto, terminamos!

8.3 Resumo

Neste Capítulo construímos uma aplicação Web completa para a venda de produtos. Utilizamos para isso tudo que aprendemos até agora. Com isso você já é capaz de implementar a maioria dos tipos de cadastros que aparecerão em sistemas do mundo real. Parabéns! No próximo Capítulo vai ser a sua vez de desenvolver, reescrevendo e melhorando o sistema de locação de DVDs.

8.4 Projetos

Projeto 8.1: Modifique a implementação do projeto desenvolvido neste Capítulo para permitir que em uma venda possa existir mais de um item de venda igual. Na implementação atual isso não é permitido, pois cada registro da tabela item_venda tem como chave primária a composição das duas chaves estrangeiras que a relaciona com as tabelas produto e venda. Será necessário modificar o DER e alguns detalhes da implementação do projeto, tanto do lado do servidor, quanto do lado do cliente.

8.5 Desafios

Desafio 8.1: Que tal implementar a paginação das listagens dos cadastros? Imagine que numa situação real você terá centenas de produtos cadastrados, concorda? Ver todos esses produtos de uma só vez na interface gráfica pode ser um grande problema, certo? Você acabou de ser contratato em uma empresa que desenvolveu o sistema deste Capítulo e seu chefe lhe deu a tarefa de implementar a tal da paginação. Você já deve ter visto em sistemas reais, veja o destaque em laranja na Figura 8.4. Você deve tomar todas as decisões necessárias, como quantos registros serão mostrados por página e deverá procurar como fazer esse tipo de limitação no banco de dados. Boa sorte!

8.5. DESAFIOS 309

Curso: SBV.IEC.IFI.2009...- IECNICO EM INFORMATICA PARA INTERNET (Campus Sao J Situação: Evasão
E-mail Acadêmico:

Matrícula: Matríc

Figura 8.4: Paginação em um sistema acadêmico

Fonte: Elaborada pelo autor

Desafio 8.2: Outra funcionalidade importante em sistemas reais é a emissão de relatórios ou de consultas personalizadas para a exibição de dados. Por exemplo, quero ter uma consulta de produtos em que eu insira parte da descrição de um produto em uma caixa de texto e, ao clicar no botão "Consultar", é mostrado na GUI do sistema todos os produtos que contenham na sua descrição o valor fornecido como uma substring. Pense só se o sistema tem centenas de produtos cadastrados e você precisa fazer uma venda! Imagine ficar rolando uma caixa de seleção por milhares de registro... Pois é, seu chefe gostou do seu trabalho na tarefa anterior e teu a missão de implementar essa funcionalidade no sistema. Ele quer que ao invés de ser exibida uma caixa de seleção com todos os produtos no formulário da venda, que haja um campo de texto onde o usuário fornecerá um valor que pode tanto ser o identificador do produto quanto seu código de barras e, ao teclar <ENTER> naquele campo, seja montada uma lista com os produtos obtidos, algo como mostrado na Figura 8.5. Use sua imaginação e criatividade para implementar tal funcionalidade!

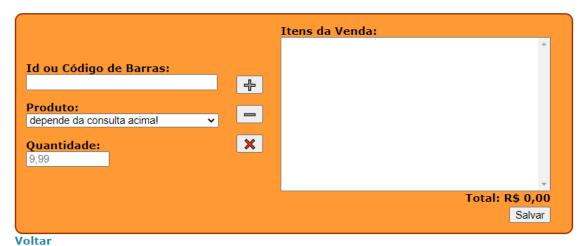


Figura 8.5: Consulta de produtos no formulário de vendas

Fonte: Elaborada pelo autor

Desafio 8.3: Seu chefe adorou sua solução para o desafio anterior e agora quer mais um "botãozinho" no sistema. Entendedores entenderão hahaha! Ele quer que você implemente a emissão de relatórios para o sistema. Neste primeiro momento ele quer dois relatórios, um que mostre a quantidade vendida de todos os produtos em um período de tempo, ou seja, para cada produto, definindo-se uma data inicial e uma data final em um formulário, realizar a consulta no banco de dados e mostrar a quantidade que foi vendida de cada um desses produtos. O outro relatório é um relatório que mostre o montante total das vendas em um período, ou seja, ele quer saber quanto de dinheiro que entrou em um período de tempo. Você não precisa usar nada além do que já sabe, mas se quiser empregar o uso de alguma *engine* de relatórios como o iReport¹, fique à vontade. Fácil? Difícil? Seu emprego está em jogo! Mãos à obra!

¹<https://community.jaspersoft.com/project/ireport-designer>

SEGUNDO PROJETO: SISTEMA PARA LOCAÇÃO DE MÍDIAS

"Com organização e tempo, acha-se o segredo de fazer tudo e bem feito".

Pitágoras



ESTE Capítulo final aplicaremos o conhecimento adquirido até o momento na construção de uma aplicação Web em Java totalmente funcional, terminando o desenvolvimento do sistema de locação de DVDs começado no Capítulo 6, alterando diversas coisas que já foram feitas,

inserindo a possibilidade do cadastro de mídias de vários tipos e, enfim, a locação de uma ou mais mídias por cada cliente. Sim, eu sei que locadora de DVDs, BluRays etc. não são mais tão populares, mas acredito que você saiba o que são e é um bom exemplo para podermos colocar o que sabemos em prática.

9.1 Introdução

Assim como no Capítulo 6, neste Capítulo será apresentada uma série de requisitos, de forma muito simplificada, que deve ser usada para criar uma aplicação Web da mesma forma que fizemos no Capítulo 8. Novamente, tudo que será requisitado estará

baseado no que já aprendemos, sendo assim, todas as funcionalidades requeridas poderão ser implementadas com recursos já vistos no Capítulo 8, mas isso não implica que você não terá que se virar de alguma forma para resolver algum problema. Agora você está apto a desenvolver um sistema completo de locação de mídias usando tudo que aprendeu até o momento. Vamos lá?

9.2 Apresentação dos Requisitos

Você foi contratado para criar um sistema para controle de cadastro de mídias. Esse sistema irá manter o cadastro dessas mídias e permitirá que clientes as aluguem. O DER do banco de dados pode ser visto na Figura 9.1. Para gerar a base, com o MariaDB/MySQL em execução, abra o modelo da base no MySQL Workbench, disponibilizado nos arquivos do Capítulo. Com o modelo aberto, clique no menu *Database* escolha a opção *Forward Engineer...* e siga o assistente. A base de dados, as tabelas e os relacionamentos serão criados, além de várias inserções em todas as tabelas, com exceção das tabelas de locação e dos itens da locação, que serão realizadas.

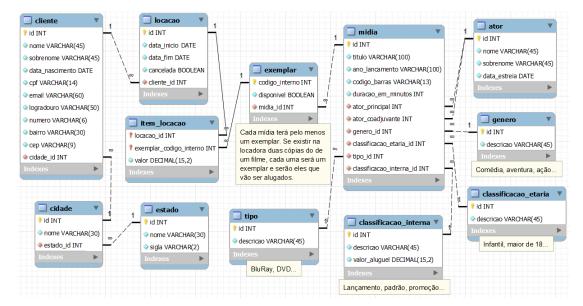


Figura 9.1: DER da base de dados

Fonte: Elaborada pelo autor

As entidades e seus atributos você poderá ver no diagrama apresentado na Figura 9.1. Cada um dos cadastros base, ou seja, cadastro de mídias e seus exemplares, atores, gêneros, classificações etárias, tipos, classificações internas, clientes, cidades e estados, deve conter as funcionalidades de criar, alterar e excluir um determinado registro. A página principal da aplicação deve conter um *link* para cada tipo de cadastro. Cada

mídia pode ter um ou mais exemplares. O valor da locação virá da classificação interna de uma mídia, por exemplo, quando um filme ou jogo é lançamento, a locação é mais cara, concorda? Abaixo de cada nova tabela há uma caixa de texto com alguns exemplos do que poderia haver naquele cadastro para que você possa se guiar. O processo de locação é análogo ao processo de venda de produtos do Capítulo 8 e aqui a locação é por exemplar, então a tabela de itens de locação já está correta para a modelagem da solução desse problema, pois não será possível alugar o mesmo exemplar duas vezes na mesma locação.

9.3 Desenvolvimento do Projeto

O projeto Web que deverá ser criado deve ter o nome de "LocacaoMidias". Configure o projeto para conter as bibliotecas internamente. O pacote de código-fonte base do projeto deve ter o nome de "locacaomidias". A estrutura do projeto deve ser igual à estrutura do projeto criado no Capítulo 8, sendo que, obviamente, o nome das classes e suas respectivas implementações serão diferentes do projeto daquele Capítulo. A base de dados com as tabelas das entidades obtidas a partir da análise dos requisitos na seção anterior deve ter o nome de "locacao_midias". Não se esqueça de que cada entidade deverá conter um identificador. As páginas da aplicação deverão ter sua aparência configurada usando uma ou mais folhas de estilos, da mesma forma que fizemos no projeto do Capítulo 8. Personalize os estilos, mudando as cores etc., além de usar imagens ou quaisquer outros artificios que julgar interessante para mudar a aparência da sua aplicação. Ainda, crie um menu legal para o usuário escolher qual cadastro quer utilizar. Note que há um projeto com a implementação inicial nos arquivos disponibilizados para este Capítulo e você pode usá-lo como ponto de partida. Nessa implementação cabe a você terminar de desenvolver as funcionalidades: cadastro de atores, cadastro de mídias, cadastro de exemplares e, a principal, que é de cadastro/realização de locações. Você pode, é claro, implementar tudo do zero.

9.4 Resumo

Neste Capítulo foi requisitado que você implementasse uma aplicação Web em Java para gerenciar a locação de mídias como BluRays e DVDs, por isso, não há atividades a serem realizadas.

Parte III

Finalizando...

CAPÍTULO

CONCLUSÃO

"O que vale na vida não é o ponto de partida e sim a caminhada. Caminhando e semeando, no fim terás o que colher".

Cora Coralina



SSIM finalizamos nosso livro! Vale a pena salientar que o que vimos durante o mesmo corresponde a uma pequena parte das funcionalidades disponíveis na linguagem de programação Java e no "Mundo" do desenvolvimento para Web, sendo que, para que você possa esgotar o assunto,

será necessário o estudo de outras fontes, além de anos de prática com a linguagem em um ambiente de desenvolvimento real.

Caso queira se especializar mais na linguagem Java, recomendo as obras:

- DEITEL, P. M.; DEITEL, H. M. **Java: como programar**. 10. ed. São Paulo: Pearson, 2017. 934 p.
- SIERRA, K. et al. **Use a Cabeça: Java**. 3. ed. Rio de Janeiro: Alta Books, 2024. 702 p.
- HORSTMANN, C. **Core Java: Fundamentals, Volume 1**. 13. ed. New York: Oracle Press, 2024. 840 p.
- HORSTMANN, C. **Core Java: Advanced Features, Volume 2**. 13. ed. New York: Oracle Press, 2024. 992 p.

Em relação às tecnologias Web atuais, existem diversas alternativas para a resolução

dos mais variados tipos de problemas. No ecossistema da plataforma Java, talvez um dos *frameworks back-end* mais famosos e utilizados seja o Spring Boot¹ que integra todas as soluções do *framework* Spring de forma simplificada. Do ponto de vista de *frameworks front-end*, os mais famosos atualmente são o Angular², o React³ e o Vue⁴.

Espero que este livro tenha sido útil!

Um grande abraço a todos!

Até mais!

¹<https://spring.io/projects/spring-boot>

²<https://angular.dev/>

³<https://react.dev/>

⁴<https://vuejs.org/>

BIBLIOGRAFIA

ALEXANDER, C.; ISHIKAWA, S.; SILVERSTEIN, M. **A Pattern Language: towns, buildings, construction**. New York: Oxford University Press, 1977. 1171 p.

DEITEL, P. M.; DEITEL, H. M. **Java: como programar**. 10. ed. São Paulo: Pearson, 2017. 934 p.

GAMMA, E. et al. **Padrões de Projeto: Soluções Reutilizáveis de Software Orientado a Objetos**. Porto Alegre: Bookman, 2000. 364 p.

HORSTMANN, C. Core Java: Advanced Features, Volume 2. 13. ed. New York: Oracle Press, 2024. 992 p.

HORSTMANN, C. **Core Java: Fundamentals, Volume 1**. 13. ed. New York: Oracle Press, 2024. 840 p.

SIERRA, K. et al. **Use a Cabeça: Java**. 3. ed. Rio de Janeiro: Alta Books, 2024. 702 p.

