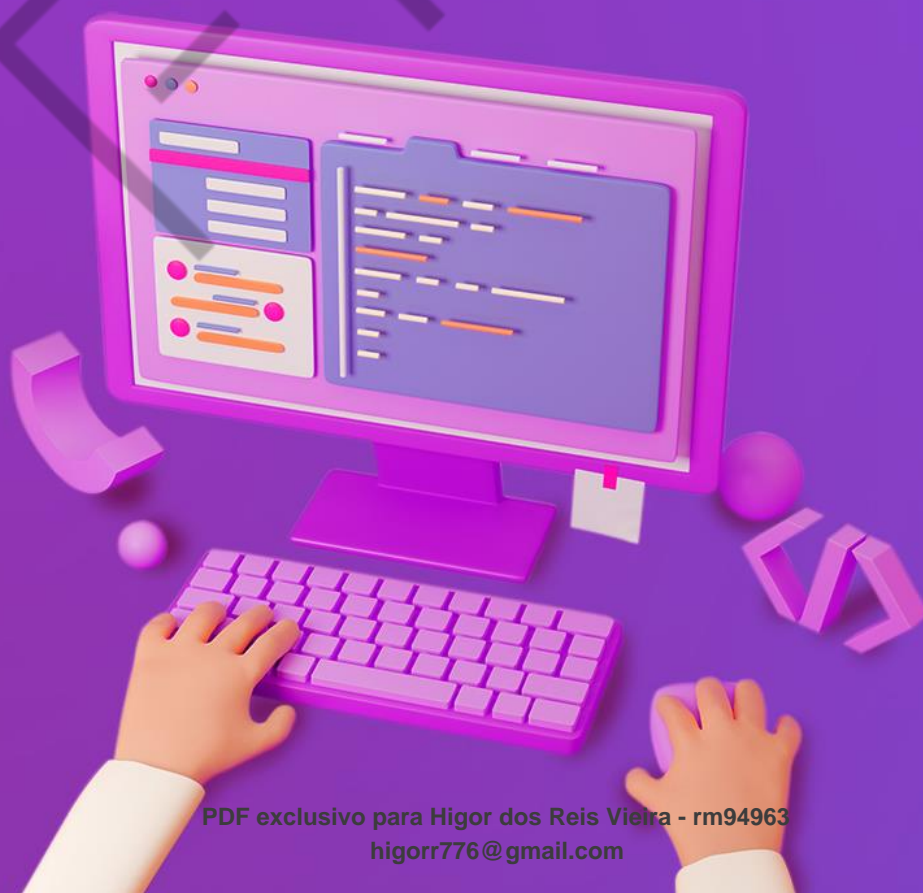


INTEGRATION

ESTE ESTUDO DE CASO

PODE TE AJUDAR!

THIAGO TOSHIYUKI I. YAMAMOTO



06

LISTA DE FIGURAS

Figura 1 – Arquitetura MVC com JSP e <i>Servlets</i>	9
Figura 2 – Criando um Projeto Java Web – Parte 1	11
Figura 3 – Criando um Projeto Java Web – Parte 2	12
Figura 4 – Criando um Projeto Java Web – Parte 3	13
Figura 5 – Estrutura do Projeto Web	14
Figura 6 – Criando a classe Produto – Parte 1.....	15
Figura 7 – Criando a classe Produto – Parte 2.....	16
Figura 8 – Criando a classe para gerenciar as conexões com o banco de dados	19
Figura 9 – Estrutura de pacotes da aplicação	21
Figura 10 – Criando uma interface para o DAO – Parte 1	22
Figura 11 – Criando uma interface para o DAO – Parte 2.....	23
Figura 12 – Criando uma Exception	24
Figura 13 – Gerando os construtores da Exception	25
Figura 14 – Criando a classe para o DAO.....	26
Figura 15 – Criando a fábrica de DAOs	39
Figura 16 – Visão geral da estrutura da aplicação	40
Figura 17 – Criando uma classe para teste.....	41
Figura 18 – Criando a página para cadastro de produto – Parte 1.....	44
Figura 19 – Criando a página para cadastro de produto – Parte 2.....	45
Figura 20 – Criando a Servlet – Parte 1	47
Figura 21 – Criando a Servlet – Parte 2	47
Figura 22 – Execução da página de cadastro no servidor – Parte 1	52
Figura 23 – Execução da página de cadastro no servidor – Parte 2	53
Figura 24 – Teste do cadastro de produto	54
Figura 25 – Teste de listagem de produtos	57
Figura 26 – Visão geral da aplicação	58
Figura 27 – Botão para editar um produto.....	58
Figura 28 – Extraindo parte de código para um método – Parte 1	64
Figura 29 – Extraindo parte de código para um método – Parte 2	65
Figura 30 – Refactoring da Servlet.....	66
Figura 31 – Teste da funcionalidade de alterar um produto – Parte 1.....	72
Figura 32 – Teste da funcionalidade de alterar um produto – Parte 2.....	72
Figura 33 – Teste da funcionalidade de excluir um produto – Parte 1	77
Figura 34 – Teste da funcionalidade de excluir um produto – Parte 2	78
Figura 35 – Teste da funcionalidade de excluir um produto – Parte 3	78
Figura 36 – Resultado da execução da classe de teste	88
Figura 37 – Opções de categorias	101
Figura 38 – Listagem de produtos com a categoria	104
Figura 39 – Listagem de produtos com a categoria	107
Figura 40 – Teste da alteração da categoria de um produto.....	109
Figura 41 – Resultado da execução do teste de criptografia	117
Figura 42 – Biblioteca de envio de e-mail do Java	122
Figura 43 – Teste de login inválido.....	129
Figura 44 – Teste de login válido	129
Figura 45 – Teste da funcionalidade de logout.....	130
Figura 46 – Criando um Filtro – Parte 1	131
Figura 47 – Criando um Filtro – Parte 2	132

Figura 48 – Teste do filtro de usuário – Parte 1	134
Figura 49 – Teste do filtro de usuário – Parte 2	134
Figura 50 – Teste da página inicial do sistema	135
Figura 51 – Erro 404, recurso não encontrado.....	136
Figura 52 – Teste da página de erro 404	137

EXEMPLO

LISTA DE CÓDIGOS-FONTE

Código-fonte 1 – SQL para criar a tabela de Produto.	10
Código-fonte 2 – SQL para criar a sequence para a tabela de Produto.	11
Código-fonte 3 – Classe Java Bean Produto.	18
Código-fonte 4 – Classe ConnectionManager.	20
Código-fonte 5 – Interface ProdutoDAO.	23
Código-fonte 6 – DBException.	26
Código-fonte 7 – Classe OracleProdutoDAO.	28
Código-fonte 8 – Método cadastrar produto.	29
Código-fonte 9 – Método atualizar produto.	30
Código-fonte 10 – Método remover produto.	31
Código-fonte 11 – Método buscar produto pelo código.	32
Código-fonte 12 – Método listar todos os produtos.	34
Código-fonte 13 – Código completo da classe OracleProdutoDAO.	38
Código-fonte 14 – Implementação da DAOFactory.	40
Código-fonte 15 – Testes do ProdutoDAO.	43
Código-fonte 16 – Página JSP de cadastro de produto.	46
Código-fonte 17 – Classe ProdutoServlet implementando a funcionalidade de cadastro.	50
Código-fonte 18 – Ajuste na página de cadastro para exibir as mensagens de sucesso e erro.	52
Código-fonte 19 – Funcionalidade de listar implementado na Servlet.	54
Código-fonte 20 – Página de listagem de produto.	56
Código-fonte 21 – Menu da aplicação.	57
Código-fonte 22 – Link para editar um produto.	60
Código-fonte 23 – Método doGet ajustado para a ação de editar.	61
Código-fonte 24 – Página de edição de produto.	63
Código-fonte 25 – Método de editar um produto.	67
Código-fonte 26 – Métodos doPost, cadastrar e editar.	70
Código-fonte 27 – Exibição das mensagens de sucesso e erro na página JSP.	70
Código-fonte 28 – Link para a listagem atualizada.	70
Código-fonte 29 – Formulário de cadastro atualizado.	71
Código-fonte 30 – Página de listagem com a opção de remoção.	75
Código-fonte 31 – Ajustes nos javascripts do projeto.	76
Código-fonte 32 – Funcionalidade de excluir na Servlet.	77
Código-fonte 33 – Implementação completa do ProdutoServlet.	82
Código-fonte 34 – SQL para criar a tabela e a sequence de categoria.	83
Código-fonte 35 – SQL para criar alterar a tabela de produto.	83
Código-fonte 36 – SQL para inserir as categorias no banco de dados.	84
Código-fonte 37 – Java Bean Categoria.	85
Código-fonte 38 – Interface CategoriaDAO.	85
Código-fonte 39 – Classe OracleCategoriaDAO.	87
Código-fonte 40 – Ajuste na classe DAOFactory.	87
Código-fonte 41 – Classe de teste do CategoriaDAO.	88
Código-fonte 42 – Classe Produto com a Categoria.	90
Código-fonte 43 – Método cadastrar produto atualizado.	91
Código-fonte 44 – Método atualizar produto atualizado.	92
Código-fonte 45 – Método buscar produto atualizado.	94

Código-fonte 46 – Método listar produto atualizado	95
Código-fonte 47 – Ajustes na classe ProdutoServlet	96
Código-fonte 48 – Método doGet da classe ProdutoServlet atualizado	97
Código-fonte 49 – Página de cadastro de produto com as opções de categoria	99
Código-fonte 50 – Método cadastrar da Servlet atualizado	100
Código-fonte 51 – Link do menu atualizado	100
Código-fonte 52 – Exibindo a categoria do produto na listagem	103
Código-fonte 53 – Método de carregar a lista de categoria na Servlet	105
Código-fonte 54 – Tela de atualização de produto atualizada	107
Código-fonte 55 – Método atualizar da Servlet atualizada	109
Código-fonte 56 – Código completo do ProdutoServlet atualizado	114
Código-fonte 57 – SQL para criar a tabela de usuário	115
Código-fonte 58 – Classe utilitária para criptografar as senhas	116
Código-fonte 59 – Classe de teste de criptografia	116
Código-fonte 60 – SQL para cadastrar um usuário na base de dados	117
Código-fonte 61 – Java bean Usuário	118
Código-fonte 62 – Interface UsuarioDAO	119
Código-fonte 63 – Classe OracleUsuarioDAO	120
Código-fonte 64 – Menu com o formulário de login	121
Código-fonte 65 – Classe para envio de e-mail	124
Código-fonte 66 – EmailException	125
Código-fonte 67 – LoginServlet	126
Código-fonte 68 – Página inicial da aplicação	127
Código-fonte 69 – Ajuste na barra de navegação para exibir o usuário ou o formulário de login	129
Código-fonte 70 – Funcionalidade de logout	130
Código-fonte 71 – Implementação do Filtro	133
Código-fonte 72 – Página inicial da aplicação	135
Código-fonte 73 – Configuração de timeout da aplicação	136
Código-fonte 74 – Configuração de página de erro	137
Código-fonte 75 – Configuração completa do web.xml	138

SUMÁRIO

1 ESTE ESTUDO DE CASO PODE TE AJUDAR!	8
1.1 Introdução	8
1.2 Arquitetura MVC	8
1.3 Banco de dados Oracle	10
1.4 Aplicação Java Web e Configurações	11
1.5 Java Bean - Produto	14
1.6 Connection manager	18
1.7 Data Access Object (DAO)	21
1.7.1 Cadastrar	28
1.7.2 Atualizar	29
1.7.3 Remover	30
1.7.4 Buscar por código	31
1.7.5 Listar	32
1.8 DAO Factory	39
1.9 Teste do DAO	41
2 CAMADA CONTROLLER E VIEW	44
2.1 Cadastrar	44
2.2 Listar	54
2.3 Editar	58
2.4 Excluir	72
2.5 Relacionamentos	82
2.6 Banco de Dados	83
2.7 Java Bean e DAO	84
2.8 View e Controller	95
2.9 Login	11
4	11
2.10 Banco de dados	11
4	11
2.11 Criptografia	11
5	11
2.12 Java Bean	11
7	11
2.13 UsuarioDAO	11
9	12
2.14 Menu com a opção de <i>login</i>	12
0	12
2.15 Serviço de envio de E-mail	12
2	12

2.16 LoginServlet	12
5	
2.17 Filtro	13
0	
2.18 Configurações da aplicação	13
5	
REFERÊNCIAS	139

EXEMPLO

1 ESTE ESTUDO DE CASO PODE TE AJUDAR!

1.1 Introdução

Chegou o momento de colocar em prática tudo o que foi abordado durante o ano! Vamos desenvolver uma solução para gerenciar produtos de uma loja. Para isso, precisamos de vários conhecimentos, como Banco de Dados, Algoritmos, Orientação a Objetos, Java, HTML, CSS, Javascript, Java Web e Engenharia de Software.

Durante o curso, modelamos o banco de dados, aprendemos sobre lógica de programação e orientação a objetos, criamos as classes Java para acesso ao banco, com alguns *Design Patterns* como DAO e Factory, desenvolvemos páginas HTML com css e javascript, construímos alguns exemplos de aplicação utilizando Java Web, JSP, JSTL, EL e Servlets. Agora, vamos juntar tudo isso para desenvolver uma aplicação completa!

1.2 Arquitetura MVC

Primeiramente, vamos falar sobre a famosa arquitetura MVC. Um padrão de arquitetura de software que divide a aplicação em três camadas:

- *Model*: camada de manipulação de dados, onde ficam as regras de negócio, validações, acesso ao banco de dados ou outros repositórios.
- *View*: camada de apresentação, onde são exibidos os dados. Não precisa ser necessariamente uma interface HTML, pode ser um PDF, dados retornados em JSON ou XML, ou seja, qualquer interface da aplicação com o usuário ou outras aplicações.
- *Controller*: camada de controle, é responsável por intermediar as camadas *views* e *models*. Por exemplo, para cadastrar um cliente, o *controller* deve recuperar as informações da tela (*view*) e repassá-las para o model persistir os dados no banco de dados.

Nós já implementamos partes dessas camadas separadamente, agora chegou a hora de colocá-las todas juntas.

Lembra-se do DAO (*Data Access Object*)? Onde ele se encaixa nessa arquitetura? Se você respondeu *model*, está certo. Esse *Design Pattern* é responsável por manipular as informações da aplicação com o banco de dados. E as páginas JSP e HTML? Elas exibem as informações para o usuário. Dessa forma, fazem parte da camada de *View* da aplicação.

E as *Servlets*? O que elas fazem? Por exemplo, no cadastro, são responsáveis por pegar as informações da tela e enviar para o DAO persistir no banco. Em uma listagem, deve recuperar as informações cadastradas no banco de dados, por meio do DAO, e enviar para o JSP exibir os dados em uma tabela HTML. Esse é o papel do *Controller*.

Você deve estar se perguntando: e os *Java Beans*? Eles estão em todas as camadas da aplicação. Por exemplo, uma classe *Cliente* é utilizada no DAO, na página JSP e na *Servlet*. Observe a Figura “Arquitetura MVC com JSP e *Servlets*”, na qual apresentamos uma visão geral da arquitetura MVC com as tecnologias JSP e *Servlets*.

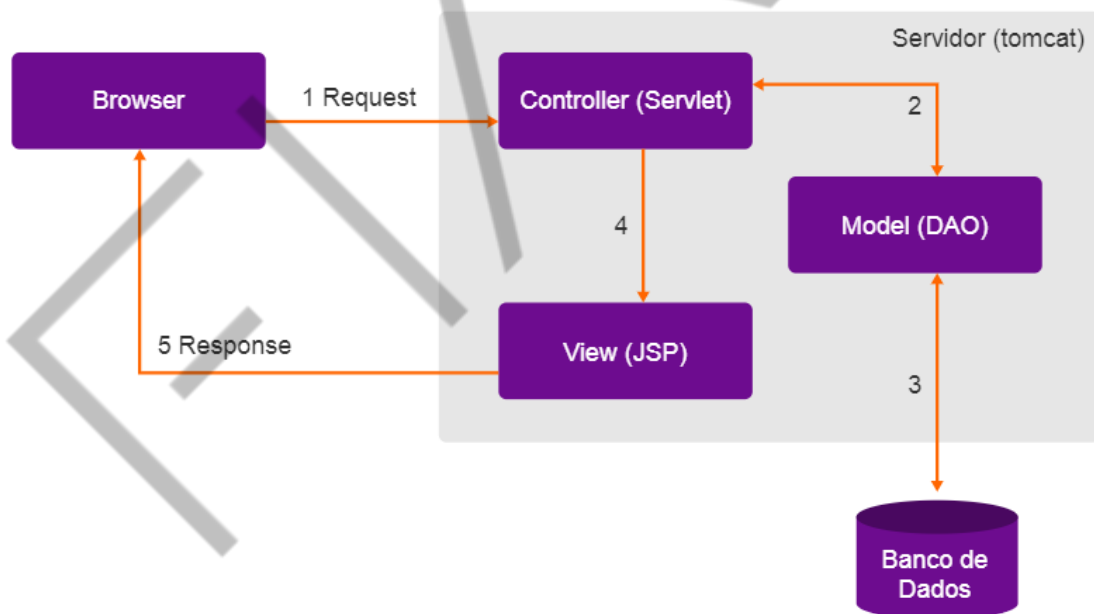


Figura 1 – Arquitetura MVC com JSP e *Servlets*
Fonte: Elaborado pelo autor (2017)

Vamos descrever, a seguir, cada interação das camadas da aplicação:

- O usuário interage com a página HTML no browser e envia uma requisição para cadastrar, listar ou pesquisar algo no sistema clicando em um link ou botão na página.

- A *Servlet* recupera as informações enviadas pelo usuário e utiliza um objeto DAO para cadastrar ou recuperar informações do banco de dados.
- O DAO acessa o banco de dados para cadastrar ou buscar informações e retorna para a *Servlet*.
- A *Servlet* adiciona informações, como uma mensagem de sucesso ou os dados retornados pelo DAO, na requisição, e encaminha para a página JSP.
- A página JSP utiliza as informações enviadas pela *Servlet* para “montar” dinamicamente a página HTML final. Esta é retornada para o usuário por meio do *response*.

Agora que entendemos um pouco sobre a arquitetura MVC, vamos começar!

1.3 Banco de dados Oracle

O primeiro passo é criar a tabela e a sequence no banco de dados. Para isso, abra o SQL Developer ou outra ferramenta para acessar o banco de dados Oracle e execute o SQL do código-fonte a seguir:

```
CREATE TABLE TB_PRODUTO (CD_PRODUTO INT PRIMARY KEY,  
NM_PRODUTO VARCHAR(100) NOT NULL, VL_PRODUTO NUMBER,  
DT_FABRICACAO DATE, QT_PRODUTO INT);
```

Código-fonte 1 – SQL para criar a tabela de Produto
Fonte: Elaborado pelo autor (2017)

Com esse SQL, estamos criando uma tabela chamada “TB_PRODUTO”, com as colunas “CD_PRODUTO”, que será a chave primária; “NM_PRODUTO”, para o nome do produto (obrigatório); “VL_PRODUTO”, que armazena o valor do produto; “DT_FABRICACAO”, para a data de fabricação; e “QT_PRODUTO”, que guarda a quantidade do produto.

Para gerar os valores para o código do produto, vamos utilizar uma sequence, que será criada por meio do Código-fonte “SQL para criar a sequence para a tabela de Produto”:

```
CREATE SEQUENCE SQ_TB_PRODUTO MINVALUE 1 START WITH 1  
INCREMENT BY 1;
```

Código-fonte 2 – SQL para criar a sequence para a tabela de Produto
Fonte: Elaborado pelo autor (2017)

Com o banco de dados pronto, vamos para a aplicação!

1.4 Aplicação Java Web e configurações

Crie um projeto Java Web. Clique no menu File > New > Dynamic Web Project, conforme a Figura “Criando um projeto Java Web – parte 1”.

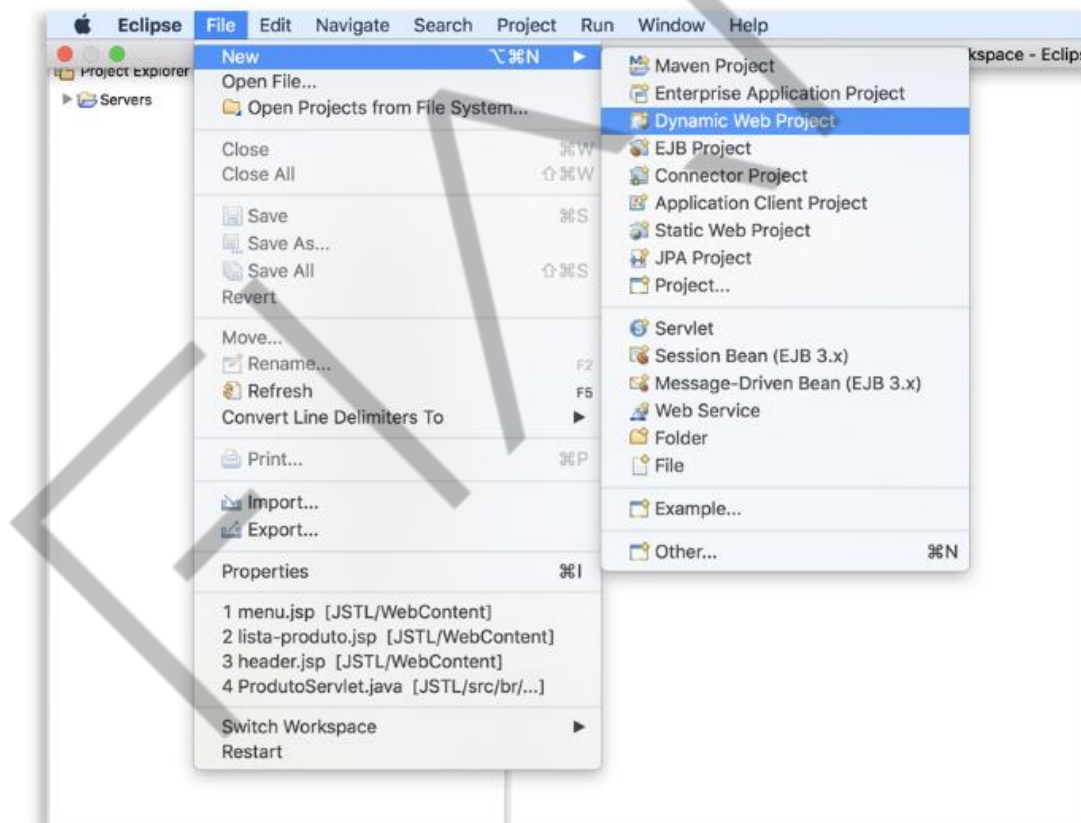


Figura 2 – Criando um projeto Java Web – parte 1
Fonte: Elaborado pelo autor (2017)

Observação: Existe um caminho alternativo, que pode ser acessado por meio do menu File > New > Project > Web > Dynamic Web Project.

Vamos dar o nome de “FiapStore” ao projeto. Não se esqueça de configurar o servidor Tomcat 10, assim como fizemos nos outros capítulos. Se você configurou o

Tomcat no eclipse, não será necessário fazê-lo novamente, pois ele já deve aparecer como uma opção para você na área de Target Runtime. Caso não tenha nenhum servidor configurado, clique em “New Runtime...”, escolha a opção Tomcat 10, clique em “Next” e depois configure o local onde o Tomcat está por meio do botão “browser”. Finalize o processo de configuração do servidor. A Figura “Criando um projeto Java Web – parte 2” apresenta as configurações iniciais para a criação do projeto.

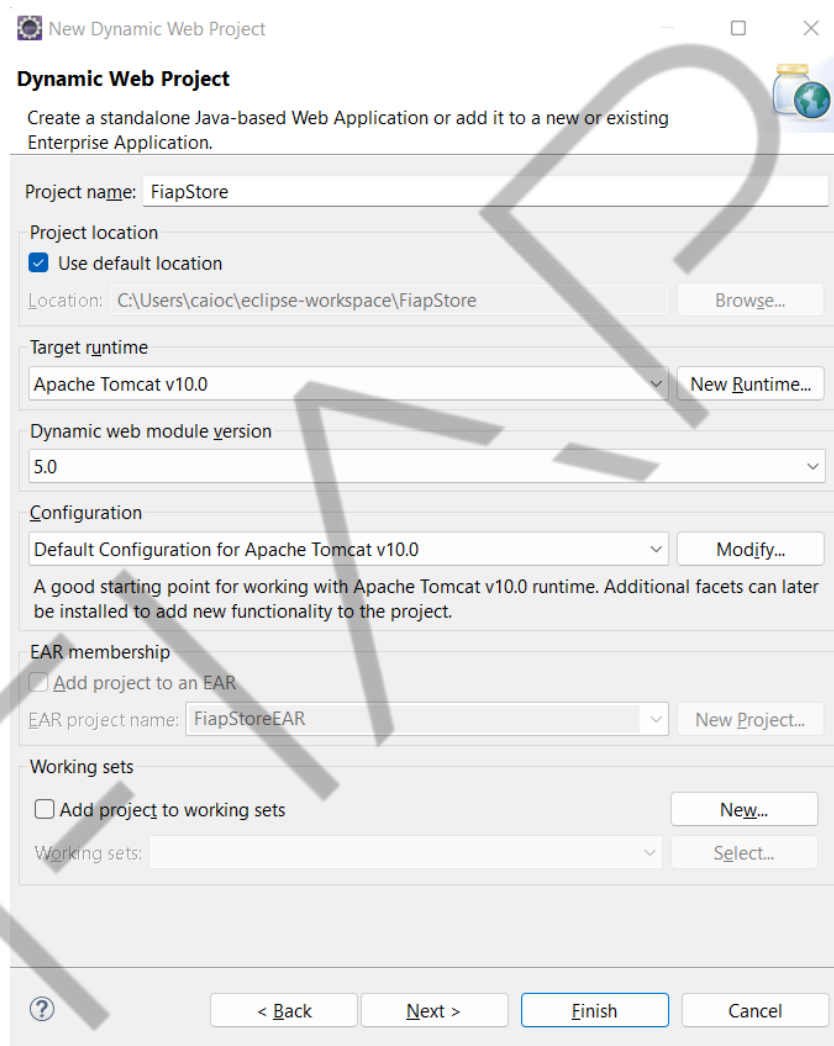


Figura 3 – Criando um projeto Java Web – parte 2
Fonte: Elaborado pelo autor (2022)

Para finalizar, clique em “Next” e “Next” até chegar à última tela. Marque a opção para criar o web.xml e finalize o processo (Figura “Criando um projeto Java Web – parte 3”).

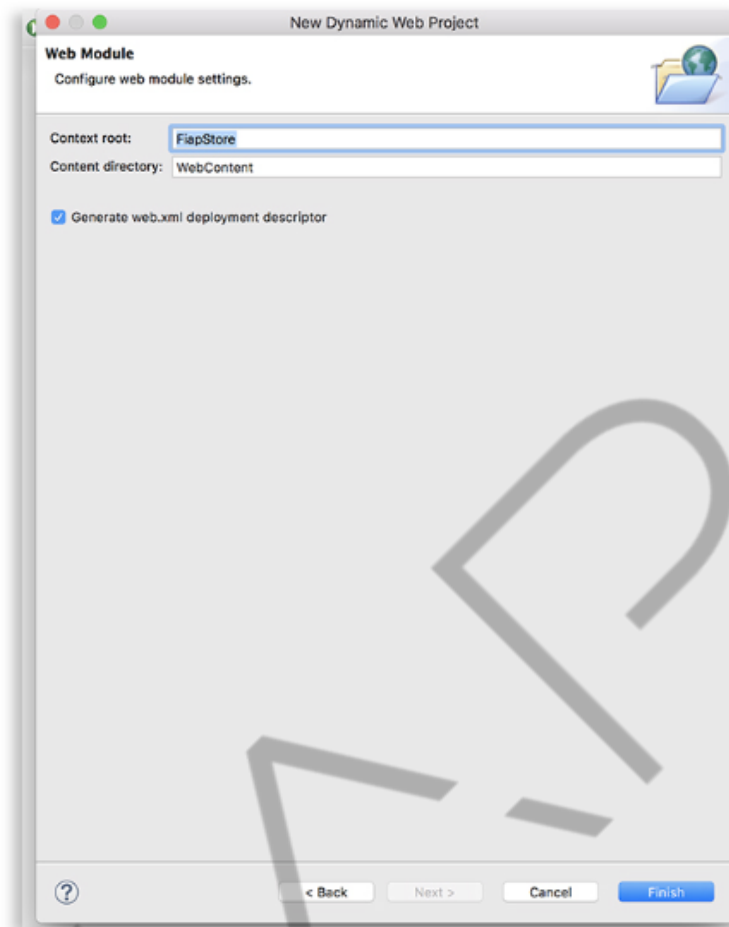


Figura 4 – Criando um projeto Java Web – parte 3
Fonte: Elaborado pelo autor (2017)

Depois de criar o projeto, vamos configurá-lo! Para criar essa aplicação, utilizaremos o banco de dados Oracle, as bibliotecas de tags JSTL e, para o *layout*, o Bootstrap.

Dessa forma, copie o jQuery e o Bootstrap para a nossa aplicação. Assim como fizemos no capítulo de JSP, crie um diretório chamado “resources” dentro da pasta “WebContent”. Copie as pastas “css” e “js” para esse diretório. Copie também as bibliotecas (jars) do driver do Oracle e das tags JSTL. Lembre-se de que o diretório correto para essas bibliotecas é “WebContent/WEB-INF/lib”.

Para finalizar, vamos criar alguns arquivos JSPs para implementar o que será comum a todas as outras páginas, como o menu, o *header* com os links do css e o *footer* com os javascripts, da mesma forma que fizemos no capítulo de JSP. A configuração final é apresentada na Figura “Estrutura do projeto Web”.

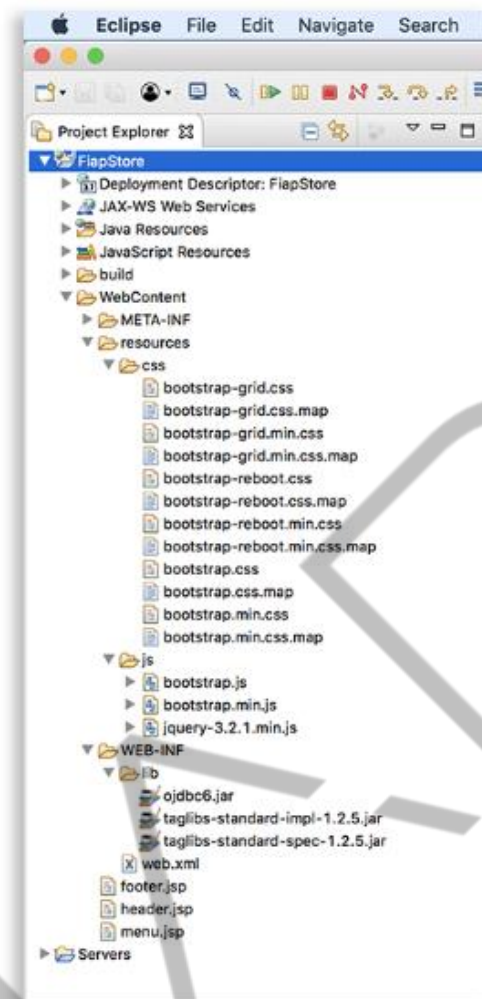


Figura 5 – Estrutura do projeto Web
Fonte: Elaborado pelo autor (2017)

Agora, sim, estamos prontos para a implementação!

1.5 Java Bean – Produto

Primeiramente, vamos criar uma classe Java Bean para o produto. Nessa classe, vamos definir os atributos que serão correspondentes às colunas do banco de dados. Para isso, crie uma nova classe Java. Clique com o botão direito do mouse na pasta “src” e escolha “New” > “Class”, conforme a Figura “Criando a classe Produto – parte 1”.

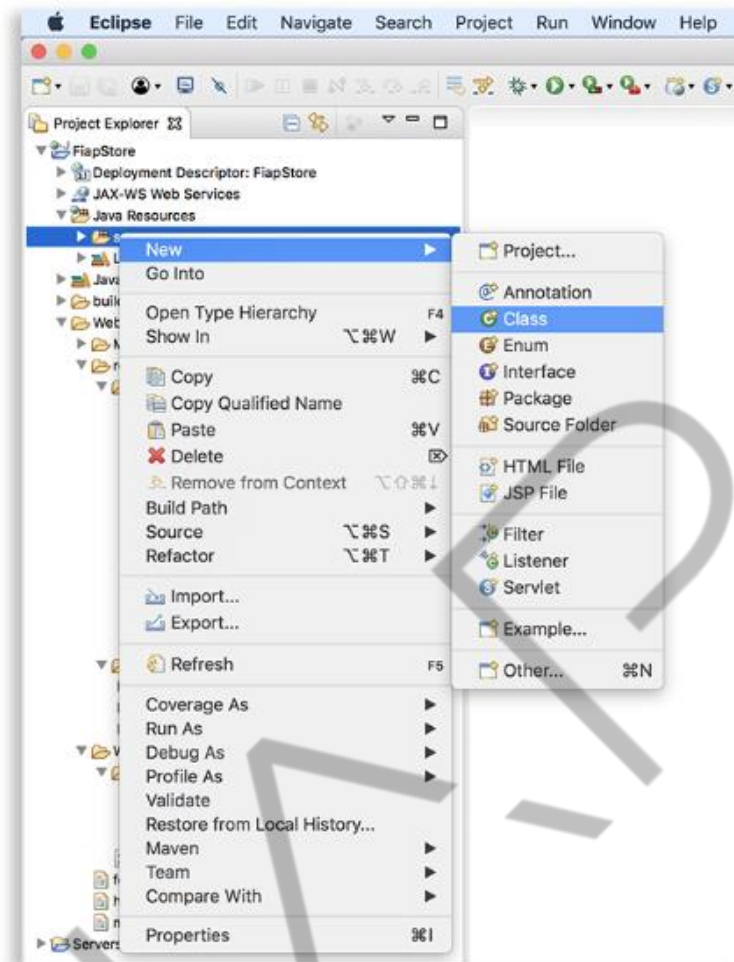


Figura 6 – Criando a classe Produto – parte 1
Fonte: Elaborado pelo autor (2017)

Utilize o pacote “br.com.fiap.store.bean” para os Java Beans. Dê o nome “Produto” para a classe e finalize o processo (Figura “Criando a classe Produto – parte 2”).

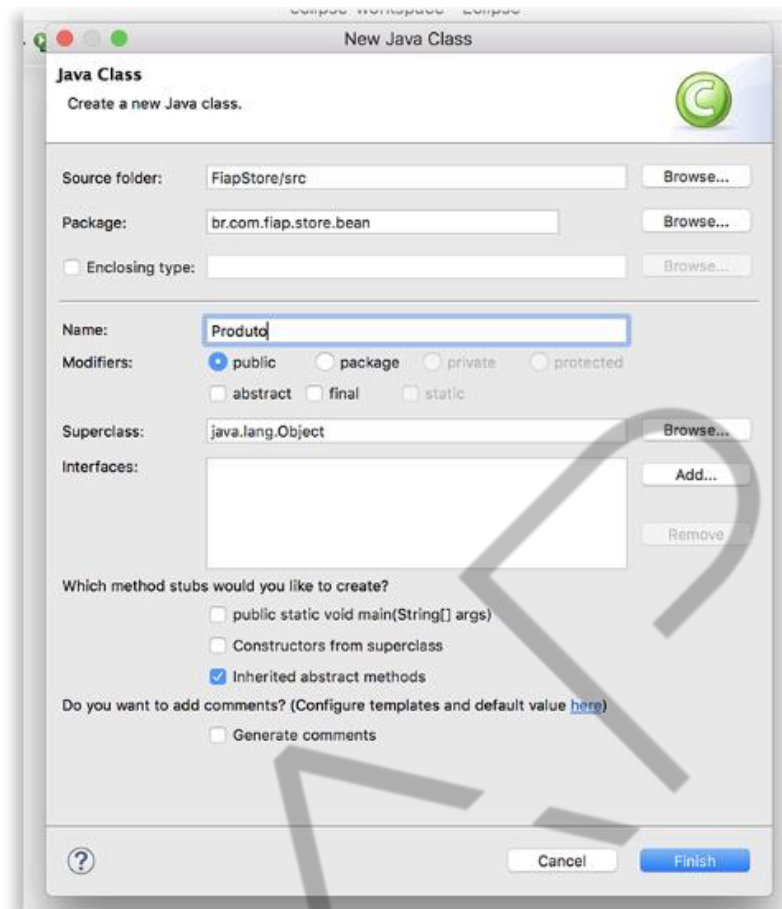


Figura 7 – Criando a classe Produto – parte 2

Fonte: Elaborado pelo autor (2017)

Codifique a classe e adicione os atributos com o modificador de acesso privado, para o encapsulamento. Crie também os métodos *gets* e *sets* para todos os atributos e dois construtores, um que recebe como parâmetro valores para todos os atributos do Produto e outro que não recebe nenhum parâmetro (Construtor padrão). O Código-fonte “Classe Java Bean Produto” apresenta a codificação completa da classe Produto:

```
package br.com.fiap.store.bean;

import java.util.Calendar;

public class Produto {

    private int codigo;

    private String nome;

    private double valor;
```



```
private Calendar dataFabricacao;

private int quantidade;

public Produto() {
    super();
}

public Produto(int codigo, String nome, double valor,
Calendar dataFabricacao, int quantidade) {
    super();
    this.codigo = codigo;
    this.nome = nome;
    this.valor = valor;
    this.dataFabricacao = dataFabricacao;
    this.quantidade = quantidade;
}

public int getCodigo() {
    return codigo;
}

public void setCodigo(int codigo) {
    this.codigo = codigo;
}

public String getNome() {
    return nome;
}

public void setNome(String nome) {
    this.nome = nome;
}

public double getValor() {
    return valor;
}

public void setValor(double valor) {
    this.valor = valor;
}

public Calendar getDataFabricacao() {
```

```
        return dataFabricacao;
    }

    public void setDataFabricacao(Calendar
dataFabricacao) {
        this.dataFabricacao = dataFabricacao;
    }

    public int getQuantidade() {
        return quantidade;
    }

    public void setQuantidade(int quantidade) {
        this.quantidade = quantidade;
    }
}
```

Código-fonte 3 – Classe Java Bean Produto
Fonte: Elaborado pelo autor (2017)

Observe que utilizamos o *Calendar* para a data de fabricação. Provavelmente, você deve saber que existem outras opções para manipular data no Java, como o *Date* e o *LocalDate*. Escolhemos o *Calendar*, pois a nova API de data do Java está disponível a partir da versão 8, e isso pode ser um problema se você trabalhar com sistemas que não possuem essa versão, o que é comum no mercado. Essa classe *Produto* será utilizada em todas as camadas da nossa aplicação. Será utilizada na *View*, no *Model* e no *Controller*.

Camada *Model*

O próximo passo será focar no *Model* ou *backend* da aplicação.

1.6 Connection manager

Crie uma nova classe para gerenciar as conexões com o banco de dados, o nome será “*ConnectionManager*” e o pacote será “*br.com.fiap.store.singleton*” (Figura “Criando a classe para gerenciar as conexões com o banco de dados”).

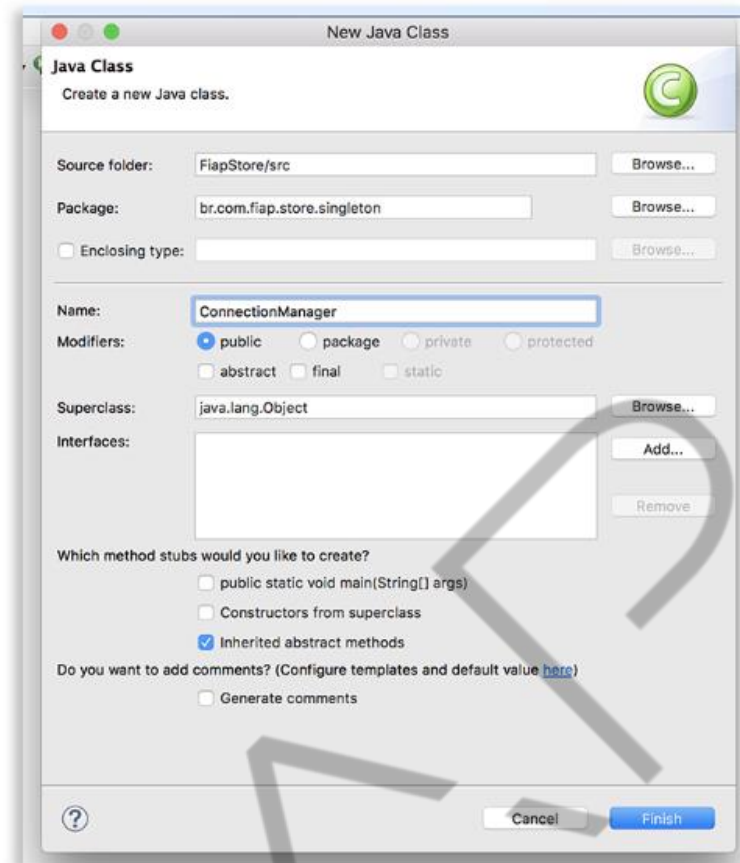


Figura 8 – Criando a classe para gerenciar as conexões com o banco de dados
Fonte: Elaborado pelo autor (2017)

A classe que cria as conexões com o banco de dados segue o padrão de projetos Singleton. Só para relembrar, esse padrão define que deve existir somente uma instância (objeto) de uma determinada classe. Em nossa aplicação, só existirá uma instância da classe `ConnectionManager`, pois não é necessário ter vários objetos desse tipo. Porém, os objetos de conexão que ele cria podem (devem) possuir várias.

Normalmente, uma classe que implementa o padrão Singleton possui um atributo privado e estático, um construtor privado e um método estático `getInstance`, que retorna à instância do objeto e é do mesmo tipo do atributo privado. É no método `getInstance` que a “mágica” acontece, pois o método faz a validação se existe uma instância no atributo, caso não exista, ele instancia, e caso exista, ele simplesmente retorna o objeto existente. É por isso que garante a existência de uma única instância da classe. O código completo do `ConnectionManager` pode ser visto no código-fonte a seguir.

```
package br.com.fiap.store.singleton;

import java.sql.Connection;
```

```
import java.sql.DriverManager;

public class ConnectionManager {

    private static ConnectionManager connectionManager;

    private ConnectionManager() {
    }

    public static ConnectionManager getInstance() {
        if (connectionManager == null) {
            connectionManager = new
ConnectionManager();
        }
        return connectionManager;
    }

    public Connection getConnection() {
        Connection connection = null;
        try {

            Class.forName("oracle.jdbc.driver.OracleDriver");

            connection =
DriverManager.getConnection("jdbc:oracle:thin:@oracle.fiap.co
m.br:1521:ORCL", "USUARIO",
                            "SENHA");
        } catch (Exception e) {
            e.printStackTrace();
        }
        return connection;
    }

}
```

Código-fonte 4 – Classe ConnectionManager

Fonte: Elaborado pelo autor (2017)

Ajuste o código inserindo o seu usuário e a senha do banco de dados. Na Figura “Estrutura de pacotes da aplicação”, podemos ver a estrutura do projeto com a classe Produto e a ConnectionManager.

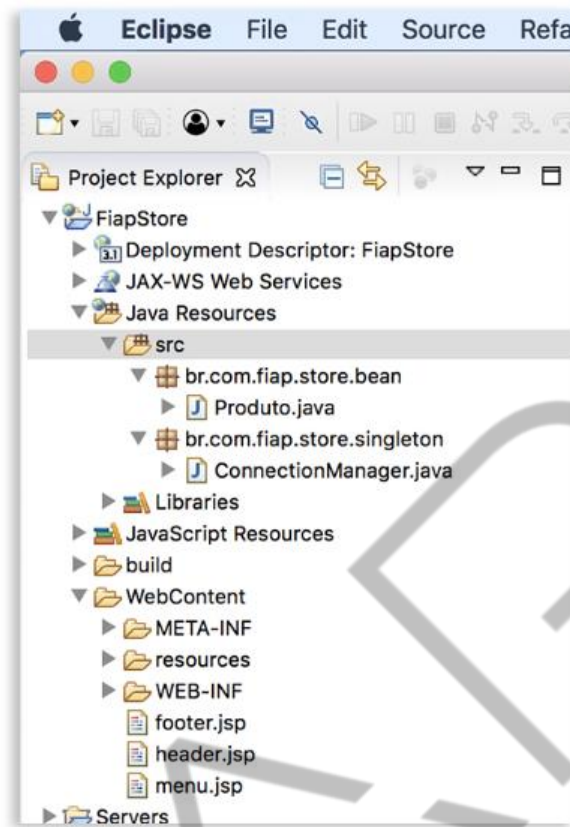


Figura 9 – Estrutura de pacotes da aplicação
Fonte: Elaborado pelo autor (2017)

1.7 Data Access Object (DAO)

Continuando a implementação, chegou o momento de criar o *Data Access Object* (DAO), ou seja, a camada responsável por manipular as informações da base de dados. Para isso, vamos criar o DAO para o Produto, começando pela interface. Clique com o botão direito do *mouse* no diretório “src” e escolha “New” > “Interface” (Figura “Criando uma interface para o DAO – parte 1”).

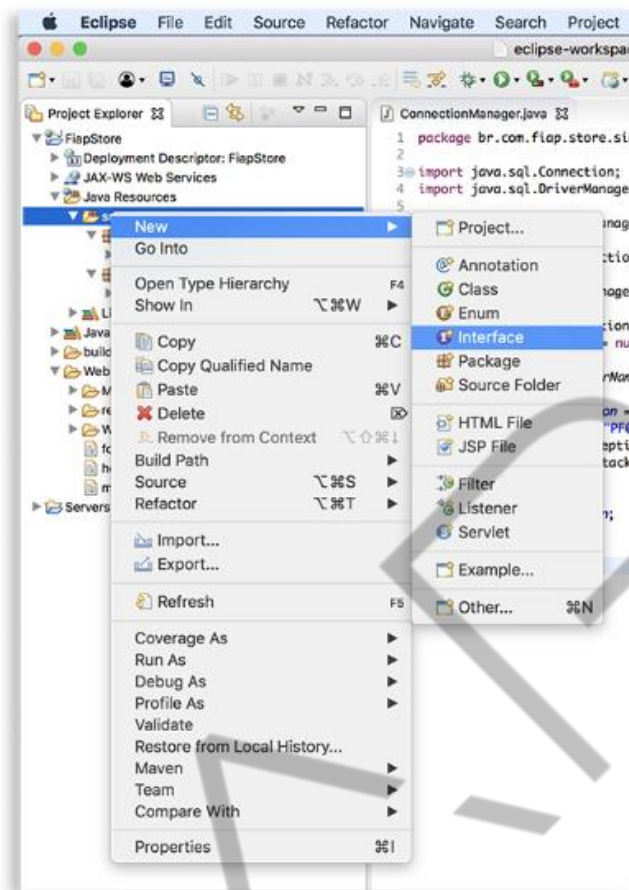


Figura 10 – Criando uma interface para o DAO – parte 1
Fonte: Elaborado pelo autor (2017)

No próximo passo, devemos definir o pacote “br.com.fiap.store.dao” e o nome da interface “ProdutoDAO” (Figura “Criando uma interface para o DAO – parte 2”).

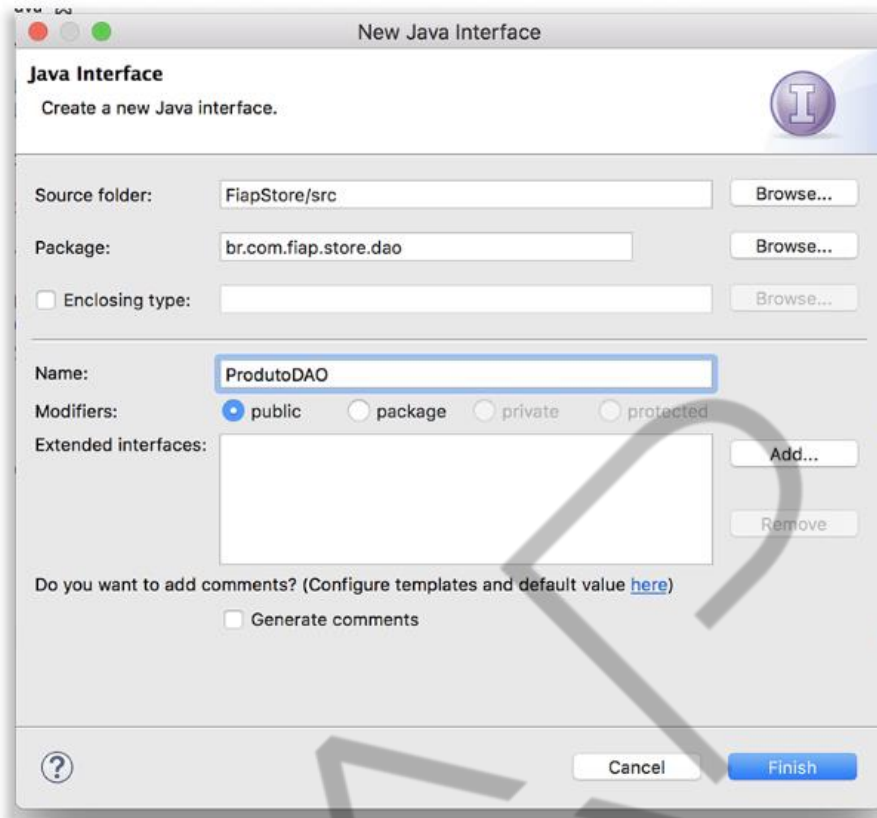


Figura 11 – Criando uma interface para o DAO – parte 2

Fonte: Elaborado pelo autor (2017)

Na interface, definimos as assinaturas dos métodos que o DAO deve implementar. Os métodos serão os básicos para um CRUD (Create, Read, Update e Delete), ou seja, para realizar um cadastro, busca e listagem, atualização e remoção.

```
package br.com.fiap.store.dao;

import java.util.List;
import br.com.fiap.store.bean.Produto;
import br.com.fiap.store.exception.DBException;

public interface ProdutoDAO {

    void cadastrar(Produto produto) throws DBException;
    void atualizar(Produto produto) throws DBException;
    void remover(int codigo) throws DBException;
    Produto buscar(int id);
    List<Produto> listar();
}
```

Código-fonte 5 – Interface ProdutoDAO

Fonte: Elaborado pelo autor (2017)

Observe que não é necessário utilizar o modificador de acesso “public”, já que os métodos definidos na interface são públicos por padrão. Outro detalhe são as exceções. Os métodos cadastrar, atualizar e remover lançam a exceção DBException. Caso algo dê errado, isso será útil para notificar o usuário se a operação foi realizada com sucesso ou não. Sem a exception, isso não seria possível.

Mas de onde veio a DBException? É uma exceção que criaremos. Ter exceções customizadas na aplicação é uma boa prática, pois assim podemos tratá-las de forma específica também. Uma exception nada mais é do que uma classe Java que herda de Exception. Assim, vamos criar uma classe com o pacote “br.com.fiap.store.exception”, denominada “DBException”, conforme mostra a Figura “Criando uma Exception”.

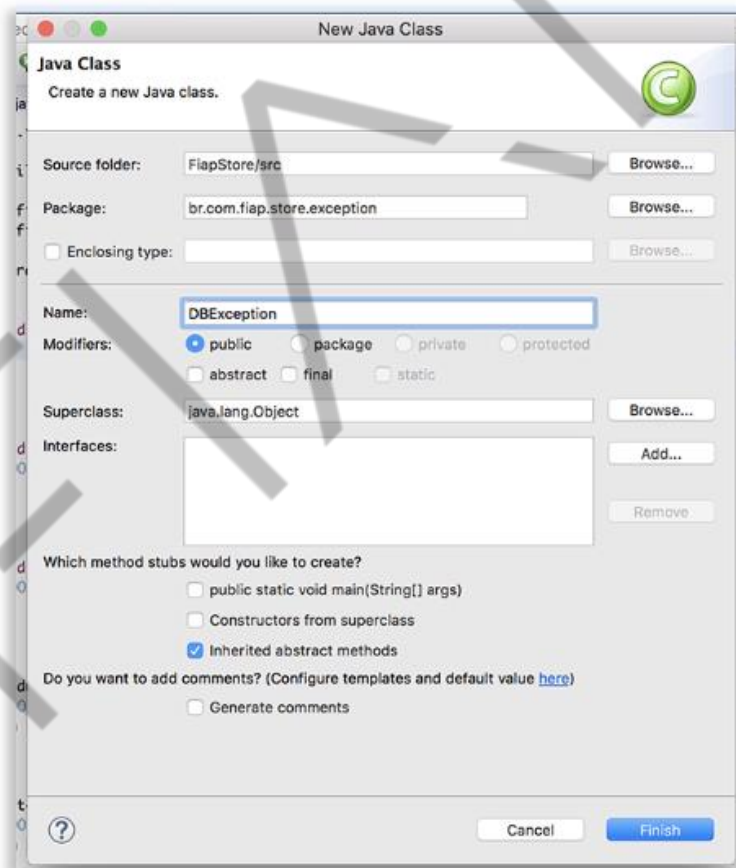


Figura 12 – Criando uma Exception
Fonte: Elaborado pelo autor (2017)

Faça a herança de Exception e crie os construtores da superclasse (Figura “Gerando os construtores da Exception”), para ser possível instanciar a Exception com uma mensagem, por exemplo.

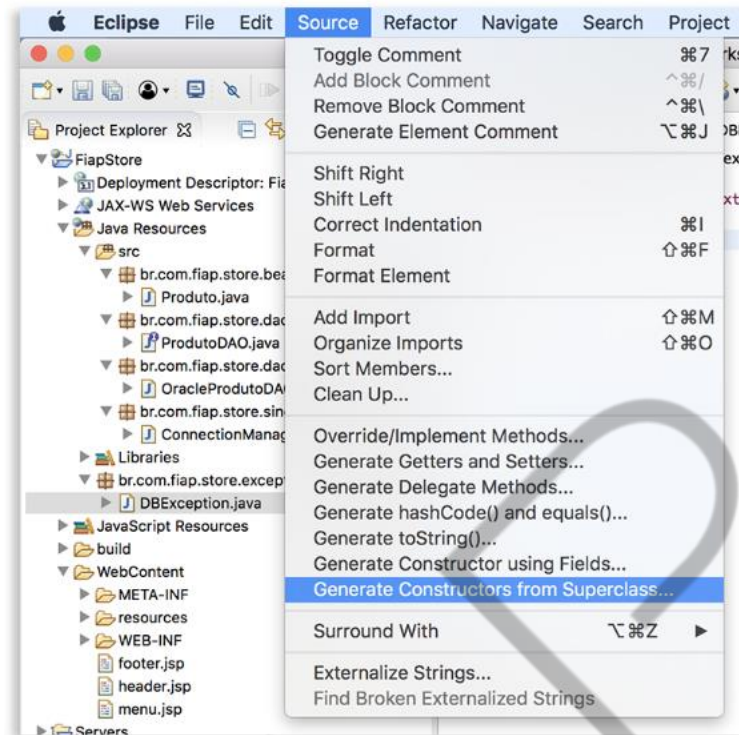


Figura 13 – Gerando os construtores da Exception
Fonte: Elaborado pelo autor (2017)

O resultado da implementação da exceção é apresentado no código-fonte a seguir:

```
package br.com.fiap.store.exception;

public class DBException extends Exception {

    public DBException() {
        super();
        // TODO Auto-generated constructor stub
    }

    public DBException(String message, Throwable cause,
        boolean enableSuppression, boolean writableStackTrace) {
        super(message, cause, enableSuppression,
        writableStackTrace);
        // TODO Auto-generated constructor stub
    }

    public DBException(String message, Throwable cause)
    {
        super(message, cause);
        // TODO Auto-generated constructor stub
    }
}
```

```
}

public DBException(String message) {
    super(message);
    // TODO Auto-generated constructor stub
}

public DBException(Throwable cause) {
    super(cause);
    // TODO Auto-generated constructor stub
}
}
```

Código-fonte 6 – DBException
Fonte: Elaborado pelo autor (2017)

Para finalizar o DAO, crie a classe que implementará a interface “ProdutoDAO”. Como estamos utilizando o banco de dados Oracle, o nome da classe será “OracleProdutoDAO”, e vamos separar os pacotes da interface da classe, por isso, defina o pacote como “br.com.fiap.store.dao.impl”, o impl é de implementação (Figura “Criando a classe para o DAO”).

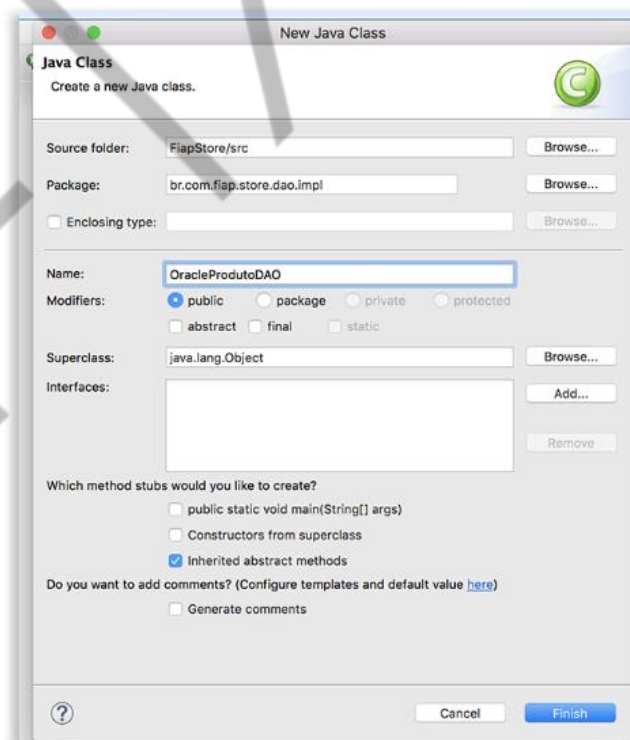


Figura 14 – Criando a classe para o DAO
Fonte: Elaborado pelo autor (2017)

Essa classe deve implementar a interface. Provavelmente, quando fizer o *implements*, a classe começará a dar erro, pois ela precisa implementar os métodos definidos pela interface. Para corrigir o erro, vá com o mouse em cima do erro e escolha a opção de implementar os métodos da interface. É possível, também, ir com o cursor até o erro e utilizar o atalho “CTRL”+ 1 e escolher a mesma opção.

Essa classe deve ter um atributo do tipo Connection, para que os métodos possam utilizá-la. Não é necessário fazer o *get* e *set* desse atributo, pois ele será utilizado somente de dentro da classe OracleProdutoDAO, a implementação parcial pode ser vista no código-fonte a seguir.

```
public class OracleProdutoDAO implements ProdutoDAO {  
  
    private Connection conexao;  
  
    @Override  
    public void cadastrar(Produto produto) throws  
    DBException {  
  
    }  
  
    @Override  
    public void atualizar(Produto produto) throws  
    DBException {  
  
    }  
  
    @Override  
    public void remover(int codigo) throws DBException {  
  
    }  
  
    @Override  
    public Produto buscar(int id) {  
  
    }  
  
    @Override  
    public List<Produto> listar() {  
  
    }  
}
```

```
}
```

Código-fonte 7 – Classe OracleProdutoDAO
Fonte: Elaborado pelo autor (2017)

Agora vamos focar na implementação de cada um dos métodos. Começando pelo cadastrar. Esse método recebe o objeto produto que deve ser persistido no banco de dados.

1.7.1 Cadastrar

Declaramos um PreparedStatement para definir a query e os parâmetros que serão executadas no banco de dados. Obtemos uma conexão com o banco de dados utilizando o ConnectionManager e definimos o SQL de insert no banco.

Com a conexão, criamos o objeto PreparedStatement e configuramos os parâmetros (valores) que serão adicionados em cada coluna no banco de dados. Por fim, executamos a query e fechamos a conexão. Caso algum erro aconteça, lançamos a exceção DBException, para informar que algo deu errado a quem chamou o método. A implementação do método é apresentada no código-fonte a seguir.

```
@Override
public void cadastrar(Produto produto) throws
DBException {
    PreparedStatement stmt = null;

    try {
        conexao
        ConnectionManager.getInstance().getConnection();
        String sql = "INSERT INTO TB_PRODUTO
(CD_PRODUTO, NM_PRODUTO, QT_PRODUTO, VL_PRODUTO,
DT_FABRICACAO) VALUES (SQ_TB_PRODUTO.NEXTVAL, ?, ?, ?, ?)";

        stmt = conexao.prepareStatement(sql);
        stmt.setString(1, produto.getNome());
        stmt.setInt(2, produto.getQuantidade());
        stmt.setDouble(3, produto.getValor());
        java.sql.Date data = new
        java.sql.Date(produto.getDataFabricacao().getTimeInMillis());
        stmt.setDate(4, data);
```

```

        stmt.executeUpdate();
    } catch (SQLException e) {
        e.printStackTrace();
        throw new DBException("Erro ao
cadastradar.");
    } finally {
        try {
            stmt.close();
            conexao.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}

```

Código-fonte 8 – Método cadastrar produto
Fonte: Elaborado pelo autor (2017)

1.7.2 Atualizar

O próximo método é o atualizar, a implementação é muito parecida com a de cadastro, com a diferença do SQL: UPDATE em vez do INSERT, e deve passar também o código na cláusula WHERE, em vez da SEQUENCE gerar um novo valor.

O código da implementação é apresentado no código-fonte a seguir.

```

@Override
public void atualizar(Produto produto) throws
DBException {
    PreparedStatement stmt = null;

    try {
        conexao
        ConnectionManager.getInstance().getConnection();
        String sql = "UPDATE TB_PRODUTO SET
NM_PRODUTO = ?, QT_PRODUTO = ?, VL_PRODUTO = ?, DT_FABRICACAO
= ? WHERE CD_PRODUTO = ?";
        stmt = conexao.prepareStatement(sql);
        stmt.setString(1, produto.getNome());
        stmt.setInt(2, produto.getQuantidade());
        stmt.setDouble(3, produto.getValor());
        java.sql.Date data = new
java.sql.Date(produto.getDataFabricacao().getTimeInMillis());
    }
}

```

```
        stmt.setDate(4, data);
        stmt.setInt(5, produto.getCodigo());

        stmt.executeUpdate();
    } catch (SQLException e) {
        e.printStackTrace();
        throw new DBException("Erro ao atualizar.");
    } finally {
        try {
            stmt.close();
            conexao.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Código-fonte 9 – Método atualizar produto
Fonte: Elaborado pelo autor (2017)

1.7.3 Remover

A funcionalidade de remover recebe o código do produto que será excluído. Novamente, declaramos uma variável do tipo PreparedStatement para criar o SQL de DELETE e o parâmetro código; para a cláusula WHERE, nunca se esqueça de colocar o filtro. Caso contrário, poderá apagar todos os registros.

Depois de configurar o PreparedStatement, é o momento de executar e fechar a conexão. Caso aconteça algum erro, uma exceção será lançada. A implementação completa desse método pode ser vista no Código-fonte “Método remover produto”.

```
@Override
public void remover(int codigo) throws DBException {
    PreparedStatement stmt = null;

    try {
        conexao =
        ConnectionManager.getInstance().getConnection();
        String sql = "DELETE FROM TB_PRODUTO WHERE
        CD_PRODUTO = ?";
        stmt = conexao.prepareStatement(sql);
    }
```

```
        stmt.setInt(1, codigo);
        stmt.executeUpdate();
    } catch (SQLException e) {
        e.printStackTrace();
        throw new DBException("Erro ao remover.");
    } finally {
        try {
            stmt.close();
            conexao.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Código-fonte 10 – Método remover produto
Fonte: Elaborado pelo autor (2017)

1.7.4 Buscar por código

Agora só faltam as buscas. A primeira implementação será a busca pelo código do produto. Esse método recebe o código que será pesquisado para colocar no SQL de SELECT. Após executar a query, validamos se encontrou algum resultado com o método next() do ResultSet. Caso exista, recuperamos os valores de cada coluna e adicionamos nos respectivos atributos do objeto Produto. Caso não encontre nada, retornamos um valor vazio (*null*). O Código-fonte “Método buscar produto pelo código” exibe a implementação do método de buscar por código.

```
@Override
public Produto buscar(int id) {
    Produto produto = null;
    PreparedStatement stmt = null;
    ResultSet rs = null;
    try {
        conexao
        ConnectionManager.getInstance().getConnection();
        stmt = conexao.prepareStatement("SELECT *
        FROM TB_PRODUTO WHERE CD_PRODUTO = ?");
        stmt.setInt(1, id);
        rs = stmt.executeQuery();
    }
```

```
        if (rs.next()) {
            int codigo = rs.getInt("CD_PRODUTO");
            String nome =
rs.getString("NM_PRODUTO");
            int qtd = rs.getInt("QT_PRODUTO");
            double valor =
rs.getDouble("VL_PRODUTO");
            java.sql.Date data =
rs.getDate("DT_FABRICACAO");
            Calendar dataFabricacao =
Calendar.getInstance();

            dataFabricacao.setTimeInMillis(data.getTime());

            produto = new Produto(codigo, nome,
valor, dataFabricacao, qtd);
        }

    } catch (SQLException e) {
        e.printStackTrace();
    } finally {
        try {
            stmt.close();
            rs.close();
            conexao.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    return produto;
}
```

Código-fonte 11 – Método buscar produto pelo código
Fonte: Elaborado pelo autor (2017)

1.7.5 Listar

O último método que será implementado é o listar. Esse método retornará todos os produtos cadastrados no banco de dados, por isso não precisa receber nenhum parâmetro. Após executar a query, utilizamos um loop para ler todos os registros encontrados. A cada iteração, recuperamos todos os valores das colunas e adicionamos no objeto produto, e este é adicionado à lista de Produtos no final da

iteração. Depois de ler todos os registros, a conexão é fechada e a lista é retornada. Observe a implementação no Código-fonte “Método listar todos os produtos”.

```
@Override
public List<Produto> listar() {
    List<Produto> lista = new ArrayList<Produto>();
    PreparedStatement stmt = null;
    ResultSet rs = null;
    try {
        conexao =
        ConnectionManager.getInstance().getConnection();
        stmt = conexao.prepareStatement("SELECT *
        FROM TB_PRODUTO");
        rs = stmt.executeQuery();

        //Percorre todos os registros encontrados
        while (rs.next()) {
            int codigo = rs.getInt("CD_PRODUTO");
            String nome =
            rs.getString("NM_PRODUTO");
            int qtd = rs.getInt("QT_PRODUTO");
            double valor =
            rs.getDouble("VL_PRODUTO");
            java.sql.Date data =
            rs.getDate("DT_FABRICACAO");
            Calendar dataFabricacao =
            Calendar.getInstance();

            dataFabricacao.setTimeInMillis(data.getTime());

            Produto produto = new Produto(codigo,
            nome, valor, dataFabricacao, qtd);
            lista.add(produto);
        }
    } catch (SQLException e) {
        e.printStackTrace();
    } finally {
        try {
            stmt.close();
            rs.close();
            conexao.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

```
    }  
    return lista;  
}
```

Código-fonte 12 – Método listar todos os produtos

Fonte: Elaborado pelo autor (2017)

Dessa forma, fechamos a implementação do DAO! O código completo, com todos os *imports* e métodos, é apresentado no código-fonte a seguir.

```
package br.com.fiap.store.dao.impl;  
  
import java.sql.Connection;  
import java.sql.PreparedStatement;  
import java.sql.ResultSet;  
import java.sql.SQLException;  
import java.util.ArrayList;  
import java.util.Calendar;  
import java.util.List;  
  
import br.com.fiap.store.bean.Produto;  
import br.com.fiap.store.dao.ProdutoDAO;  
import br.com.fiap.store.exception.DBException;  
import br.com.fiap.store.singleton.ConnectionManager;  
  
public class OracleProdutoDAO implements ProdutoDAO {  
  
    private Connection conexao;  
  
    @Override  
    public void cadastrar(Produto produto) throws  
DBException {  
        PreparedStatement stmt = null;  
  
        try {  
            conexao =  
ConnectionManager.getInstance().getConnection();  
            String sql = "INSERT INTO TB_PRODUTO  
(CD_PRODUTO, NM_PRODUTO, QT_PRODUTO, VL_PRODUTO,  
DT_FABRICACAO) VALUES (SQ_TB_PRODUTO.NEXTVAL, ?, ?, ?, ?)";  
            stmt = conexao.prepareStatement(sql);  
            stmt.setString(1, produto.getNome());  
            stmt.setInt(2, produto.getQuantidade());  
            stmt.setDouble(3, produto.getValor());  
        }  
    }  
}
```

```

        java.sql.Date data = new
        java.sql.Date(produto.getDataFabricacao().getTimeInMillis());
        stmt.setDate(4, data);

        stmt.executeUpdate();
    } catch (SQLException e) {
        e.printStackTrace();
        throw new DBException("Erro ao
        cadastrar.");
    } finally {
        try {
            stmt.close();
            conexao.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}

@Override
public void atualizar(Produto produto) throws
DBException {
    PreparedStatement stmt = null;

    try {
        conexao =
        ConnectionManager.getInstance().getConnection();
        String sql = "UPDATE TB_PRODUTO SET
        NM_PRODUTO = ?, QT_PRODUTO = ?, VL_PRODUTO = ?, DT_FABRICACAO
        = ? WHERE CD_PRODUTO = ?";
        stmt = conexao.prepareStatement(sql);
        stmt.setString(1, produto.getNome());
        stmt.setInt(2, produto.getQuantidade());
        stmt.setDouble(3, produto.getValor());
        java.sql.Date data = new
        java.sql.Date(produto.getDataFabricacao().getTimeInMillis());
        stmt.setDate(4, data);
        stmt.setInt(5, produto.getCodigo());

        stmt.executeUpdate();
    } catch (SQLException e) {
        e.printStackTrace();
        throw new DBException("Erro ao
        atualizar.");
    }
}

```

```
        } finally {
            try {
                stmt.close();
                conexao.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }

    }

    @Override
    public void remover(int codigo) throws DBException {
        PreparedStatement stmt = null;

        try {
            conexao =
            ConnectionManager.getInstance().getConnection();
            String sql = "DELETE FROM TB_PRODUTO WHERE
            CD_PRODUTO = ?";

            stmt = conexao.prepareStatement(sql);
            stmt.setInt(1, codigo);
            stmt.executeUpdate();
        } catch (SQLException e) {
            e.printStackTrace();
            throw new DBException("Erro ao remover.");
        } finally {
            try {
                stmt.close();
                conexao.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }

    @Override
    public Produto buscar(int id) {
        Produto produto = null;
        PreparedStatement stmt = null;
        ResultSet rs = null;
        try {
```

```

        conexao
        ConnectionManager.getInstance().getConnection();
        stmt = conexao.prepareStatement("SELECT *
FROM TB_PRODUTO WHERE CD_PRODUTO = ?");
        stmt.setInt(1, id);
        rs = stmt.executeQuery();

        if (rs.next()){
            int codigo = rs.getInt("CD_PRODUTO");
            String nome
rs.getString("NM_PRODUTO");
            int qtd = rs.getInt("QT_PRODUTO");
            double valor
rs.getDouble("VL_PRODUTO");
            java.sql.Date data
rs.getDate("DT_FABRICACAO");
            Calendar dataFabricacao
Calendar.getInstance();

            dataFabricacao.setTimeInMillis(data.getTime());

            produto = new Produto(codigo, nome,
valor, dataFabricacao, qtd);
        }

    } catch (SQLException e) {
        e.printStackTrace();
    } finally {
        try {
            stmt.close();
            rs.close();
            conexao.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    return produto;
}

@Override
public List<Produto> listar() {
    List<Produto> lista = new ArrayList<Produto>();
    PreparedStatement stmt = null;
    ResultSet rs = null;

```

```
        try {
            conexao =
            ConnectionManager.getInstance().getConnection();
            stmt = conexao.prepareStatement("SELECT *
            FROM TB_PRODUTO");
            rs = stmt.executeQuery();

            //Percorre todos os registros encontrados
            while (rs.next()) {
                int codigo = rs.getInt("CD_PRODUTO");
                String nome =
                rs.getString("NM_PRODUTO");
                int qtd = rs.getInt("QT_PRODUTO");
                double valor =
                rs.getDouble("VL_PRODUTO");
                java.sql.Date data =
                rs.getDate("DT_FABRICACAO");
                Calendar dataFabricacao =
                Calendar.getInstance();

                dataFabricacao.setTimeInMillis(data.getTime());

                Produto produto = new Produto(codigo,
                nome, valor, dataFabricacao, qtd);
                lista.add(produto);
            }
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            try {
                stmt.close();
                rs.close();
                conexao.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
        return lista;
    }
}
```

Código-fonte 13 – Código completo da classe OracleProdutoDAO
Fonte: Elaborado pelo autor (2017)

1.8 DAO Factory

Para finalizar o *backend*, crie a Fábrica de DAOs, uma classe responsável por instanciar os DAOs de nossa aplicação, outro padrão de projetos que foi abordado anteriormente. Para isso, crie uma nova classe no pacote “br.com.fiap.store.factory”, chamada “DAOFactory” (Figura “Criando a fábrica de DAOs”).

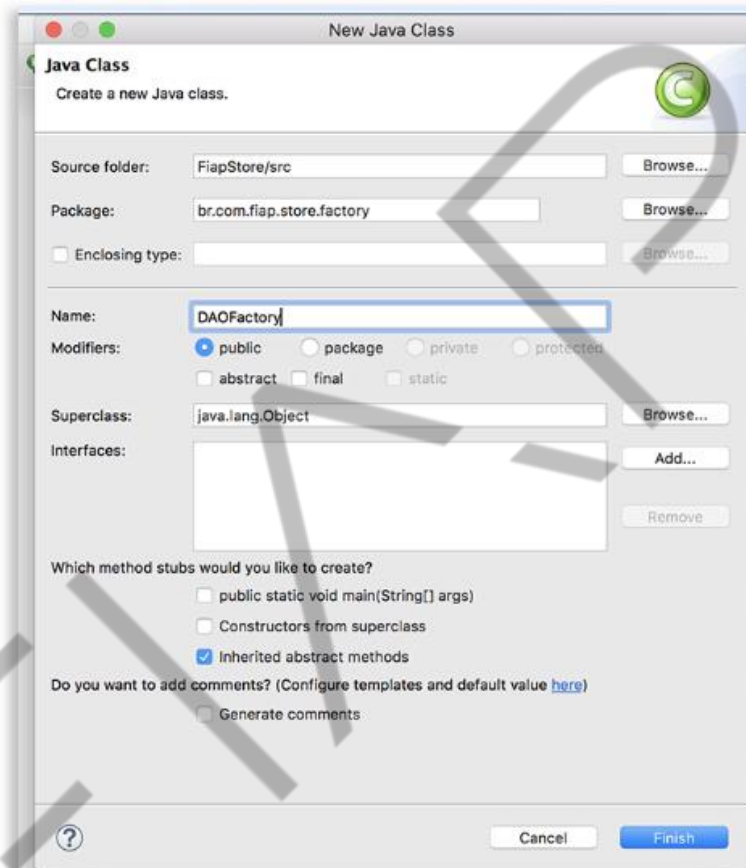


Figura 15 – Criando a fábrica de DAOs
Fonte: Elaborado pelo autor (2017)

A implementação dessa classe será bem simples, somente com um método, já que só temos um DAO no sistema. Esse método deve retornar um objeto do tipo ProdutoDAO. Segue a implementação da Factory no Código-fonte “Implementação da DAOFactory”.

```
package br.com.fiap.store.factory;

import br.com.fiap.store.dao.ProdutoDAO;
import br.com.fiap.store.dao.impl.OracleProdutoDAO;
```

```
public class DAOFactory {  
  
    public static ProdutoDAO getProdutoDAO() {  
        return new OracleProdutoDAO();  
    }  
  
}
```

Código-fonte 14 – Implementação da DAOFactory

Fonte: Elaborado pelo autor (2017)

Com isso, finalizamos a implementação do *backend* da nossa aplicação. Dentro da arquitetura MVC, a camada de modelo (Model) está pronta. A Figura “Visão geral da estrutura da aplicação” apresenta a estrutura de classes completa do *backend*.

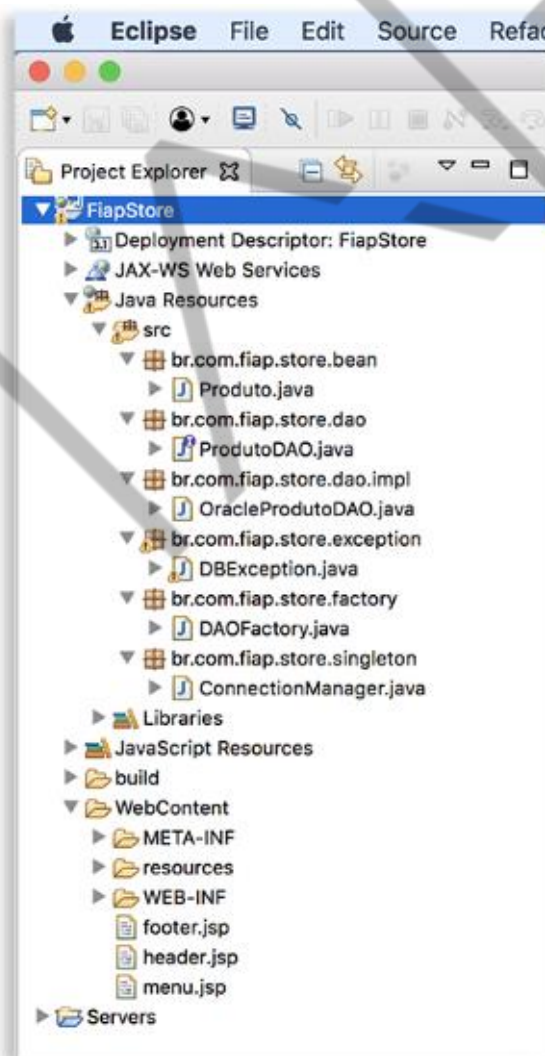


Figura 16 – Visão geral da estrutura da aplicação

Fonte: Elaborado pelo autor (2017)

1.9 Teste do DAO

Mas, até agora, você sentiu falta de alguma coisa?

Os testes! É sempre indicado testar tudo o mais rápido possível. A cada pequena implementação, realize os testes. Nunca deixe para testar tudo no final, pois fica mais difícil encontrar o problema. Existem alguns *frameworks* em Java, como o JUnit, que o ajudam na realização dos testes.

Em nosso caso, podemos implementar um método do DAO e testá-lo, antes de implementar o próximo método. Crie uma classe com o método main para realizar os testes. Nesse método, vamos testar todas as funcionalidades do DAO, porém, podemos criar também uma classe para cada funcionalidade do DAO.

Crie uma nova classe no pacote “br.com.fiap.store.teste”, chamada “ProdutoDAOTeste” (Figura “Criando uma classe para teste”).

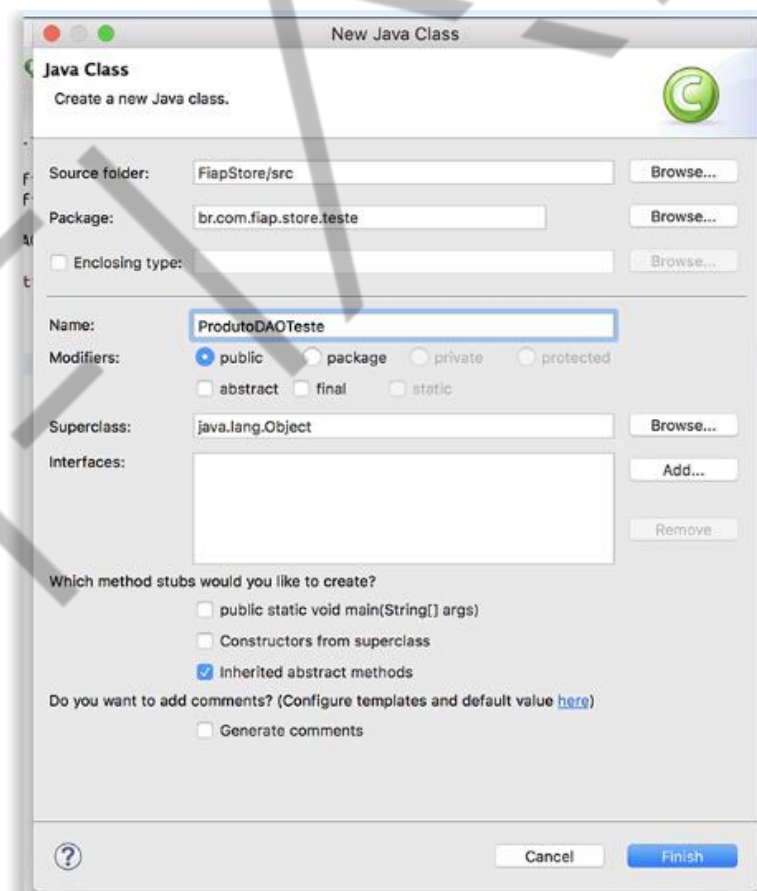


Figura 17 – Criando uma classe para teste
Fonte: Elaborado pelo autor (2017)

Implemente o método main e, dentro dele, crie um objeto do tipo ProdutoDAO utilizando a DAOFactory. Depois, utilize as funcionalidades do CRUD para testar o DAO. Um exemplo de teste é apresentado no Código-fonte “Testes do ProdutoDAO”.

```
package br.com.fiap.store.teste;

import java.util.Calendar;
import java.util.List;

import br.com.fiap.store.bean.Produto;
import br.com.fiap.store.dao.ProdutoDAO;
import br.com.fiap.store.exception.DBException;
import br.com.fiap.store.factory.DAOFactory;

public class ProdutoDAOTeste {

    public static void main(String[] args) {
        ProdutoDAO dao = DAOFactory.getProdutoDAO();

        //Cadastrar um produto
        Produto produto = new
        Produto(0, "Caderno", 20, Calendar.getInstance(), 100);
        try {
            dao.cadastrar(produto);
            System.out.println("Produto cadastrado.");
        } catch (DBException e) {
            e.printStackTrace();
        }

        //Buscar um produto pelo código e atualizar
        produto = dao.buscar(1);
        produto.setNome("Caderno capa dura");
        produto.setValor(30);
        try {
            dao.atualizar(produto);
            System.out.println("Produto atualizado.");
        } catch (DBException e) {
            e.printStackTrace();
        }

        //Listar os produtos
        List<Produto> lista = dao.listar();
        for (Produto item : lista) {
```

```
        System.out.println(item.getNome() + " " +
item.getQuantidade() + " " + item.getValor());
    }

    //Remover um produto
    try {
        dao.remover(1);
        System.out.println("Produto removido.");
    } catch (DBException e) {
        e.printStackTrace();
    }
}
}
```

Código-fonte 15 – Testes do ProdutoDAO

Fonte: Elaborado pelo autor (2017)

Para testar, execute como Java Application e analise o resultado no console do eclipse.

2 CAMADA CONTROLLER E VIEW

Com o *backend* pronto, chegou o momento de implementar a interface com o usuário, ou seja, as páginas e o *controller*, a camada responsável por recuperar as informações inseridas pelo usuário (*view*) e enviar para a camada *model*.

2.1 Cadastrar

Comece pela tela, crie o JSP para cadastrar um produto. Clique com o botão direito do mouse na pasta “WebContent” e escolha New > JSP File (Figura “Criando a página para cadastro de produto – parte 1”).

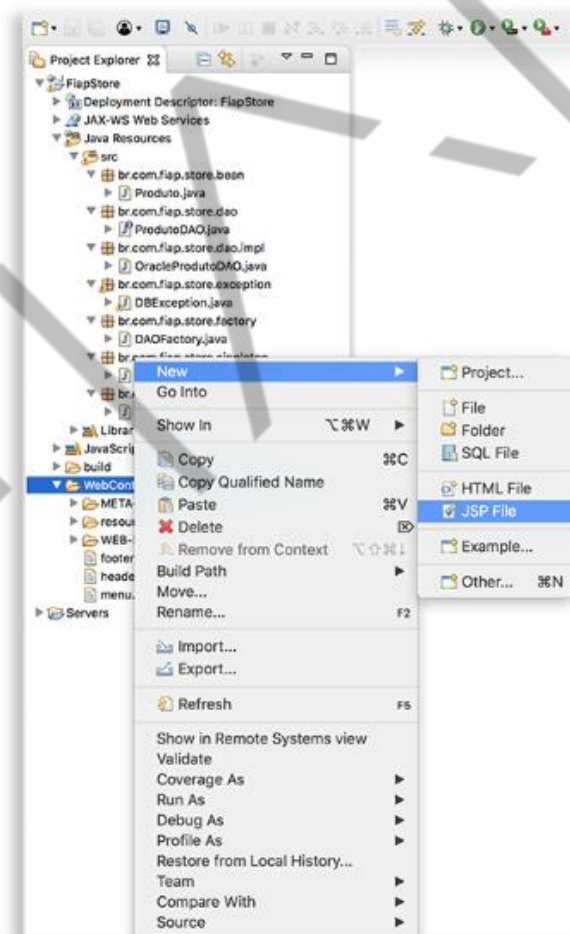


Figura 18 – Criando a página para cadastro de produto – parte 1
Fonte: Elaborado pelo autor (2017)

Dê o nome “cadastro-produto” e finalize o processo clicando no botão “Finish”, conforme a Figura “Criando a página para cadastro de produto – parte 2”.

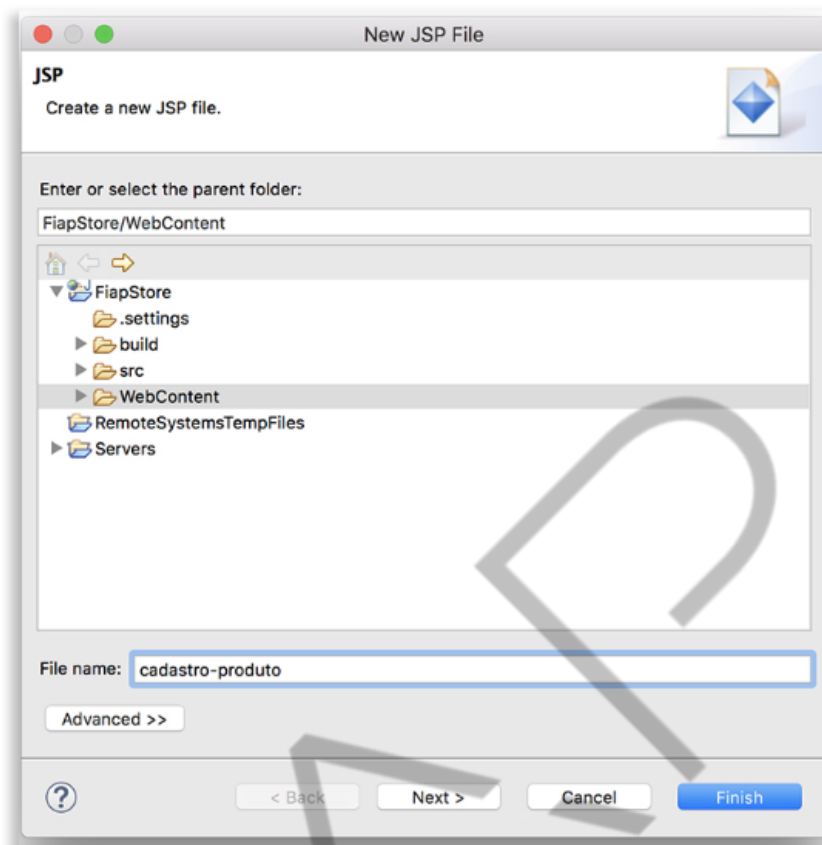


Figura 19 – Criando a página para cadastro de produto – parte 2
Fonte: Elaborado pelo autor (2017)

Utilize os JSPs do menu, header e footer, fazendo os includes na página de cadastro. Crie o formulário com o método Post e *action* "produto". Utilize o Bootstrap para estilizar os campos do formulário. Atente-se aos nomes que utilizar, pois eles serão usados no momento de recuperar as informações na Servlet. No label, o atributo **for** identifica o campo do formulário que ele descreve. Por isso deve ter o mesmo valor do **id** do campo. Analise o código-fonte a seguir, que implementa a página de cadastro de produto.

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01
Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1">
<title>Cadastro de Produto</title>
```

```
<%@ include file="header.jsp" %>
</head>
<body>
<%@ include file="menu.jsp" %>
<div class="container">
  <h1>Cadastro de Produto</h1>
  <form action="produto" method="post">
    <div class="form-group">
      <label for="id-nome">Nome</label>
      <input type="text" name="nome" id="id-nome" class="form-control">
    </div>
    <div class="form-group">
      <label for="id-valor">Valor</label>
      <input type="text" name="valor" id="id-valor" class="form-control">
    </div>
    <div class="form-group">
      <label for="id-quantidade">Quantidade</label>
      <input type="text" name="quantidade" id="id-quantidade" class="form-control">
    </div>
    <div class="form-group">
      <label for="id-fabricacao">Data de Fabricação</label>
      <input type="text" name="fabricacao" id="id-fabricacao" class="form-control">
    </div>
    <input type="submit" value="Salvar" class="btn btn-primary">
  </form>
</div>
<%@ include file="footer.jsp" %>
</body>
</html>
```

Código-fonte 16 – Página JSP de cadastro de produto
Fonte: Elaborado pelo autor (2017)

Com a primeira *view* finalizada, vamos implementar o *controller*. Crie uma *Servlet*, para isso, clique com o botão direito do mouse na pasta “src” e escolha New > *Servlet* (Figura “Criando a *Servlet* – parte 1”).

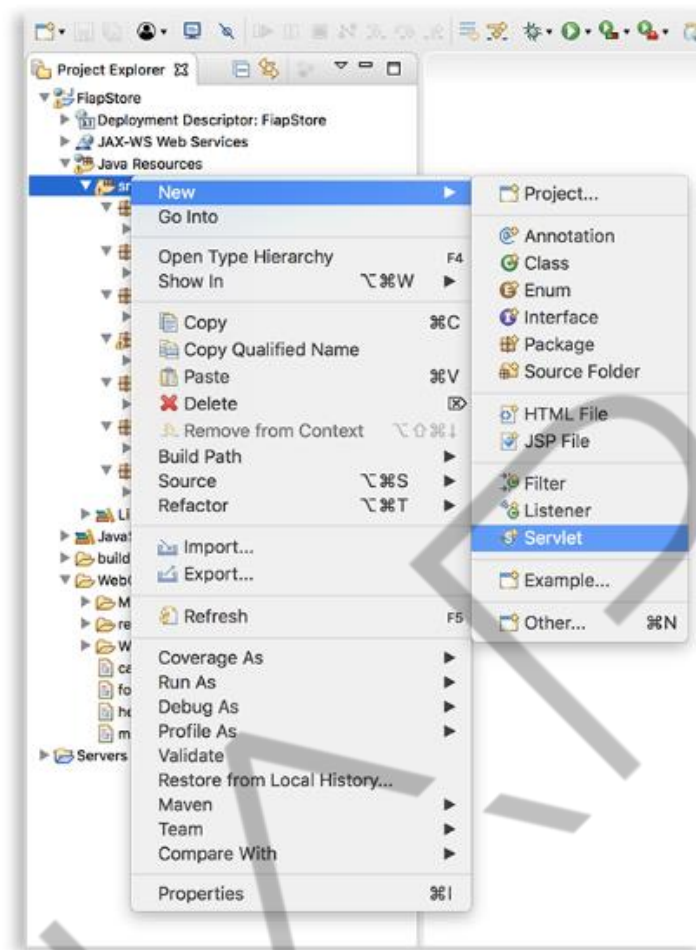


Figura 20 – Criando a Servlet – parte 1
Fonte: Elaborado pelo autor (2017)

Configure o pacote para “br.com.fiap.store.controller”, e denomine a Servlet “ProdutoServlet” (Figura “Criando a Servlet – parte 2”).

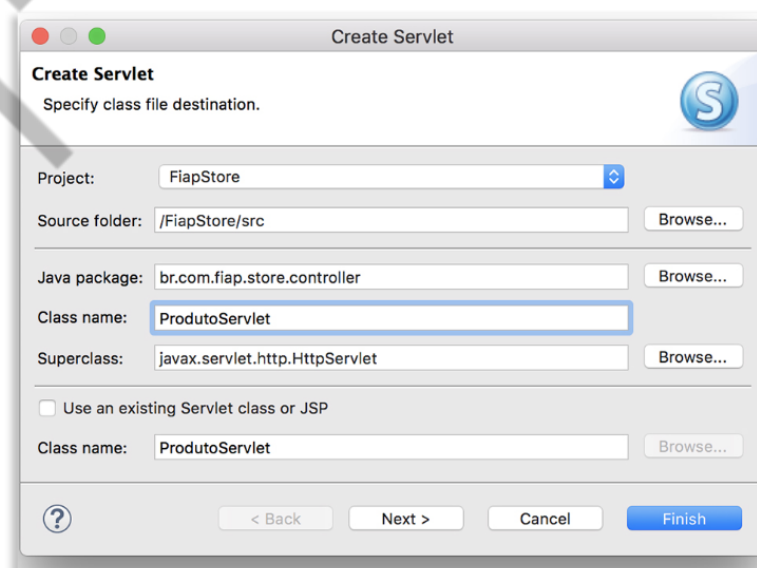


Figura 21 – Criando a Servlet – parte 2
Fonte: Elaborado pelo autor (2017)

O primeiro ajuste que devemos realizar é o mapeamento da Servlet, na anotação `@WebServlet`, ajuste o valor para `"/produto"`, igual ao valor da action do formulário. Dessa forma, sempre utilizaremos a URL `"produto"` para enviar as requisições a essa Servlet.

A Servlet precisa do ProdutoDAO para realizar as operações com o banco de dados, assim, declare um atributo privado do tipo ProdutoDAO. Não é preciso ter `get` e `set`, pois essa classe utilizará somente o DAO. Para inicializar o DAO, utilize o método do ciclo de vida da Servlet **`init()`**, esse método é invocado pelo servidor no momento que a Servlet é criada.

Para tratar a requisição enviada pelo formulário, utilize o método **`doPost`**. Recupere todos os parâmetros do formulário, lembre-se de que os parâmetros são recuperados por meio do método **`getParameter("")`** da request e recebem o nome (name) do campo do formulário. Esse método sempre retorna uma String, por isso precisamos realizar as conversões para números e data. Para a conversão de data, é necessário um objeto especial e especializado em formatar datas. Vamos utilizar o `SimpleDateFormat`, quando instanciamos essa classe, podemos informar qual será o padrão de data esperado: o caractere `"d"` representa o dia, `"M"` o mês e `"y"` o ano, assim, se utilizamos o valor `"dd/MM/yyyy"`, definimos dois dígitos para o dia, dois para o mês e quatro para o ano, tudo separado por barra (/).

Esse formatador consegue converter uma String para um Date, bem como o contrário, um Date para String, quando queremos exibir a data formatada. Depois de recuperar os valores e convertê-los, vamos instanciar um produto utilizando o construtor com argumentos. Como o código é gerado pela sequence, podemos passar o valor zero (0). Finalmente, utilizamos o método cadastrar do DAO passando o produto para inserir no banco e adicionamos uma mensagem como atributo do *request*. Caso aconteça algum erro, adicionamos uma mensagem de erro na *request*.

Note que adicionamos dois tipos de erro, um para `DBException`, em que provavelmente ocorreu um erro de banco de dados, e outro `Exception` mais genérico, que pode acontecer caso o usuário insira alguma informação inválida, como uma letra no lugar de número ou uma data com formato inválido. O código-fonte a seguir exibe a implementação completa da classe ProdutoServlet.


```
package br.com.fiap.store.controller;

import java.io.IOException;
import java.text.SimpleDateFormat;
import java.util.Calendar;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import br.com.fiap.store.bean.Produto;
import br.com.fiap.store.dao.ProdutoDAO;
import br.com.fiap.store.exception.DBException;
import br.com.fiap.store.factory.DAOFactory;

@WebServlet("/produto")
public class ProdutoServlet extends HttpServlet {

    private static final long serialVersionUID = 1L;

    private ProdutoDAO dao;

    @Override
    public void init() throws ServletException {
        super.init();
        dao = DAOFactory.getProdutoDAO();
    }

    protected void doPost(HttpServletRequest request,
        HttpServletResponse response) throws ServletException,
        IOException {

        try{
            String nome =
request.getParameter("nome");
            int quantidade =
Integer.parseInt(request.getParameter("quantidade"));
            double preco =
Double.parseDouble(request.getParameter("valor"));
            SimpleDateFormat format = new
SimpleDateFormat("dd/MM/yyyy");
```

```

Calendar                                fabricacao                                =
Calendar.getInstance();

    fabricacao.setTime(format.parse(request.getParameter("fa
bricacao")));

        Produto produto = new Produto(0, nome,
preco, fabricacao, quantidade);
        dao.cadastrar(produto);

        request.setAttribute("msg",          "Produto
cadastrado!");
    } catch (DBException db) {
        db.printStackTrace();
        request.setAttribute("erro",        "Erro    ao
cadastrar");
    } catch (Exception e) {
        e.printStackTrace();
        request.setAttribute("erro", "Por    favor,
valide os dados");
    }
    request.getRequestDispatcher("cadastro-
produto.jsp").forward(request, response);
}

}

```

Código-fonte 17 – Classe ProdutoServlet implementando a funcionalidade de cadastro

Fonte: Elaborado pelo autor (2017)

Você se lembra de que adicionamos mensagens para serem exibidas aos usuários na página? Vamos ajustar a página de cadastro de produto. Para isso, adicione a taglib core do JSTL e utilize a tag <c:if> a fim de testar se existe uma mensagem no request. Caso exista, exiba-a dentro de uma div com a *class* do Bootstrap que define uma caixa de alerta. Isso é necessário, pois, se não validarmos, a caixa sempre será exibida, mesmo sem uma mensagem. Vamos fazer o mesmo para a mensagem de erro, mas com a caixa de alerta em outra cor. Observe as partes destacadas do Código-fonte “Ajuste na página de cadastro para exibir as mensagens de sucesso e erro”.

```

<%@ page language="java" contentType="text/html;
charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="c"
uri="http://java.sun.com/jsp/jstl/core" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01
Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1">
<title>Cadastro de Produto</title>
<%@ include file="header.jsp" %>
</head>
<body>
<%@ include file="menu.jsp" %>
<div class="container">
<h1>Cadastro de Produto</h1>
<c:if test="${not empty msg}">
<div class="alert alert-success">${msg}</div>
</c:if>
<c:if test="${not empty erro}">
<div class="alert alert-danger">${erro}</div>
</c:if>
<form action="produto" method="post">
<input type="hidden" value="cadastrar"
name="acao">
<div class="form-group">
<label for="id-nome">Nome</label>
<input type="text" name="nome" id="id-
nome" class="form-control">
</div>
<div class="form-group">
<label for="id-valor">Valor</label>
<input type="text" name="valor" id="id-
valor" class="form-control">
</div>
<div class="form-group">
<label for="id-
quantidade">Quantidade</label>
<input type="text" name="quantidade"
id="id-quantidade" class="form-control">
</div>
<div class="form-group">

```

```

<label for="id-fabricacao">Data de
Fabricação</label>
<input type="text" name="fabricacao"
id="id-fabricacao" class="form-control">
</div>
<input type="submit" value="Salvar" class="btn
btn-primary">
</form>
</div>
<%@ include file="footer.jsp" %>
</body>
</html>

```

Código-fonte 18 – Ajuste na página de cadastro para exibir as mensagens de sucesso e erro
Fonte: Elaborado pelo autor (2017)

Com tudo implementado, agora é a hora da verdade. Chegou o momento do teste. Para isso, clique com o botão direito do mouse na página “cadastro-produto.jsp” e escolha a opção Run As > Run non Server, conforme mostra a Figura “Execução da página de cadastro no servidor – parte 1”.

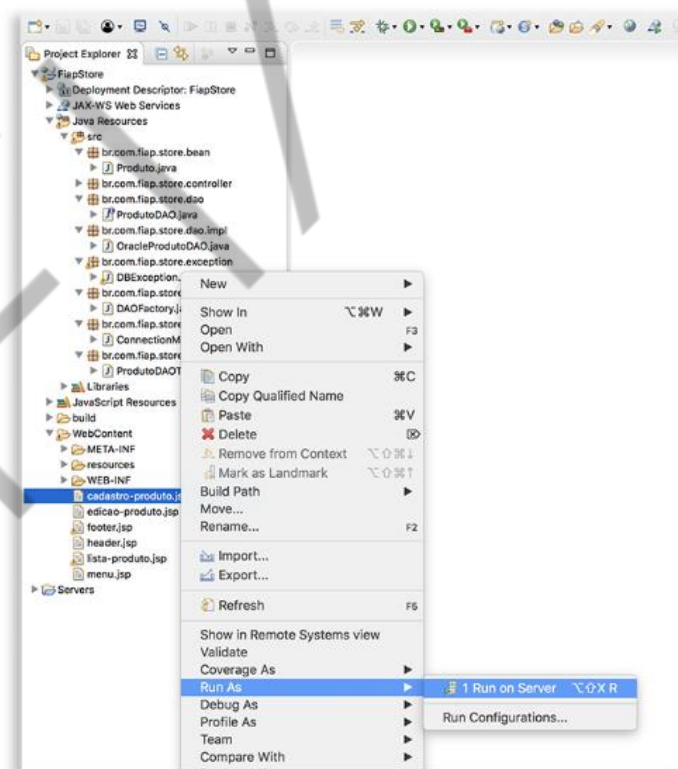


Figura 22 – Execução da página de cadastro no servidor – parte 1
Fonte: Elaborado pelo autor (2017)

Depois, escolha o servidor Tomcat 10 e clique em “Finish” (Figura “Execução da página de cadastro no servidor – parte 2”). Lembre-se de que, na opção Window > Web Browser, você pode modificar o *browser* que será aberto.

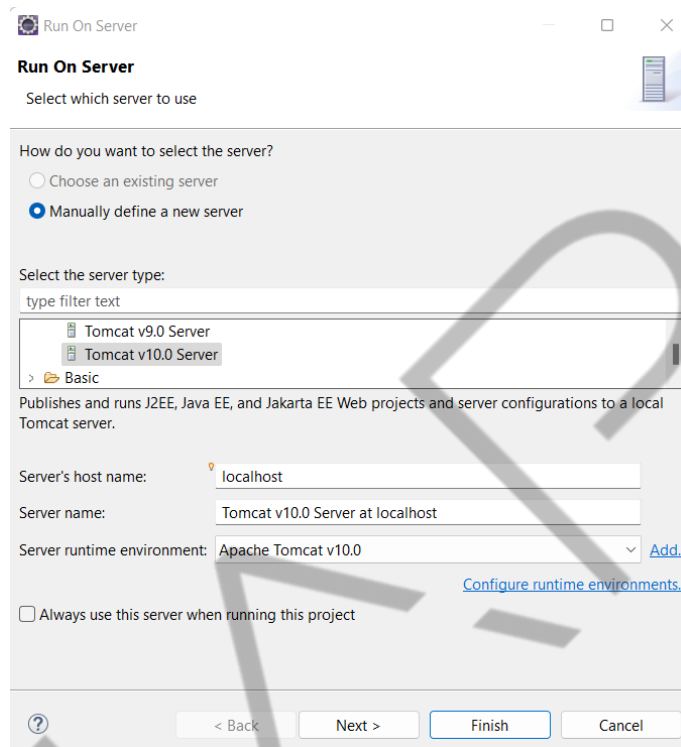


Figura 23 – Execução da página de cadastro no servidor – parte 2
Fonte: Elaborado pelo revisor (2022)

Após a página ser aberta, insira os dados e clique no botão “Salvar”. Se tudo der certo, a mensagem de sucesso deve aparecer (Figura “Teste do cadastro de produto”). Abra o banco de dados e valide se os dados realmente foram gravados.

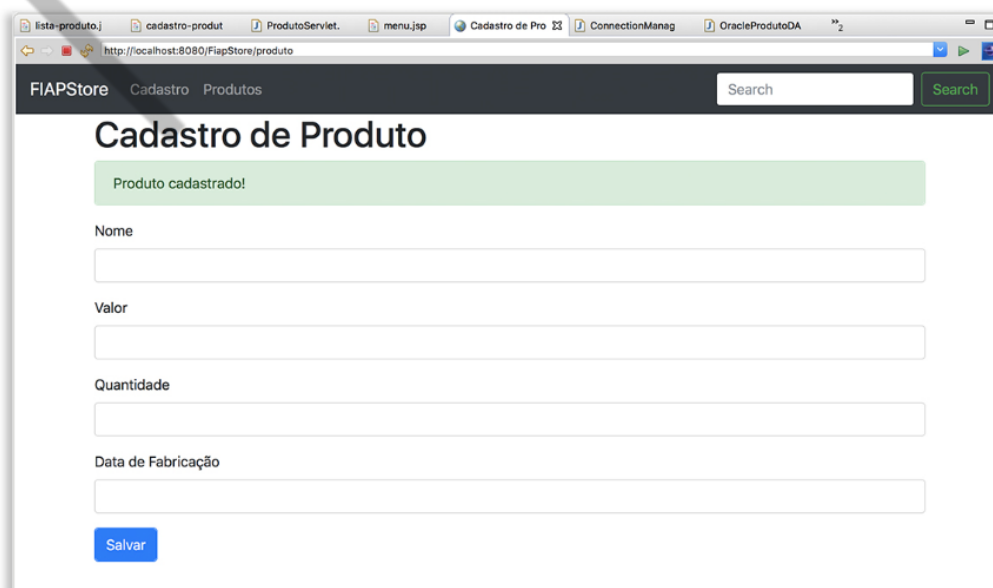


Figura 24 – Teste do cadastro de produto
Fonte: Elaborado pelo autor (2017)

2.2 Listar

Para a listagem, primeiro implemente o método **doGet** na Servlet. Deve ser esse método, pois a requisição será enviada por meio de um link. O método deve buscar todos os produtos cadastrados no banco, por meio do método `listar()` do DAO (Código-fonte “Funcionalidade de listar implementado na Servlet”). O resultado deve ser enviado para a página JSP, por isso é preciso adicionar a lista no *request* e depois redirecionar para a página “lista-produto.jsp”, que vamos criar a seguir.

```
@Override
protected void doGet(HttpServletRequest request,
    HttpServletResponse response) throws ServletException,
    IOException {
    List<Produto> lista = dao.listar();
    request.setAttribute("produtos", lista);
    request.getRequestDispatcher("lista-
produto.jsp").forward(request, response);
}
```

Código-fonte 19 – Funcionalidade de listar implementada na Servlet
Fonte: Elaborado pelo autor (2017)

Agora, crie a página JSP. Clique com o botão direito do mouse na pasta “WebContent” e escolha a opção New > JSP File. Dê o nome “lista-produto.jsp” e finalize o processo.

Faça o include do menu, header e footer. Aproveite e adicione as taglibs core e de formatação do JSTL. Desenvolva a tabela em HTML, com as colunas nome, quantidade, valor e data de fabricação. Para a tabela, utilizamos a classe “table” e “table-striped” a fim de estilizar a tabela e deixar cada linha de uma cor.

Para o conteúdo da tabela, utilize a tag `<c:forEach>` para percorrer a lista de produtos e construir cada linha da tabela, a cada iteração. O atributo “**items**” recebe a lista de produtos que foi enviada da Servlet, o nome deve ser igual ao atributo da Servlet, nesse caso “produtos”. Já o atributo **var** define a variável que recebe cada item da lista e será utilizado para acessar cada atributo do Produto.

Para exibir a data de forma amigável ao usuário, precisamos formatá-la. Para isso, utilizamos a tag de formatação `<fmt:formatDate>`, observe que, no fim do atributo **dataFabricacao**, adicionamos o **.time**, pois essa tag formata objetos do tipo `Date` e não `Calendar`, então precisamos acessar o atributo `time` do `Calendar` para recuperar o objeto `Date` do `Calendar`.

A implementação da página de listagem é apresentada no Código-fonte “Página de listagem de produto”.

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
    <%@ taglib prefix="c"
uri="http://java.sun.com/jsp/jstl/core" %>
    <%@ taglib prefix="fmt"
uri="http://java.sun.com/jsp/jstl/fmt" %>
    <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01
Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
    <html>
    <head>
    <meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1">
    <title>Cadastro de Produto</title>
    <%@ include file="header.jsp" %>
    </head>
    <body>

    <%@ include file="menu.jsp" %>
    <div class="container">
        <h1>Produtos</h1>
        <table class="table table-striped">
            <tr>
                <th>Nome</th>
                <th>Quantidade</th>
                <th>Valor</th>
                <th>Data de Fabricação</th>
            </tr>
            <c:forEach items="${produtos}" var="p">
                <tr>
                    <td>${p.nome}</td>
                    <td>${p.quantidade}</td>
                    <td>${p.valor}</td>
```

```
 <fmt:formatDate value="{p.dataFabricacao.time }" pattern="dd/MM/yyyy"/> |
```

Código-fonte 20 – Página de listagem de produto
Fonte: Elaborado pelo autor (2017)

Para acessar a funcionalidade de listagem e de cadastro, vamos ajustar o arquivo menu.jsp a fim de criar dois links na barra de navegação. Observe que, para o cadastro, utilizamos o nome do arquivo “cadastro-produto.jsp” e, para a listagem, o valor mapeado na classe ProdutoServlet, “produto” (Código-fonte “Menu da aplicação”).

```

<nav class="navbar navbar-expand-lg navbar-dark bg-dark">
  <a class="navbar-brand" href="#">FIAPStore</a>
  <button class="navbar-toggler" type="button" data-
toggle="collapse" data-target="#navbarSupportedContent" aria-
controls="navbarSupportedContent" aria-expanded="false" aria-
label="Toggle navigation">
    <span class="navbar-toggler-icon"></span>
  </button>
  <div class="collapse navbar-collapse"
id="navbarSupportedContent">
    <ul class="navbar-nav mr-auto">
      <li class="nav-item">
        <a class="nav-link" href="cadastro-
produto.jsp">Cadastro</a>
      </li>
      <li class="nav-item">
        <a class="nav-link" href="produto">Produtos</a>
      </li>
    </ul>
    <form class="form-inline my-2 my-lg-0">

```



```

        <input class="form-control mr-sm-2" type="text"
placeholder="Search" aria-label="Search">
        <button class="btn btn-outline-success my-2 my-sm-
0" type="submit">Search</button>
    </form>
</div>
</nav>

```

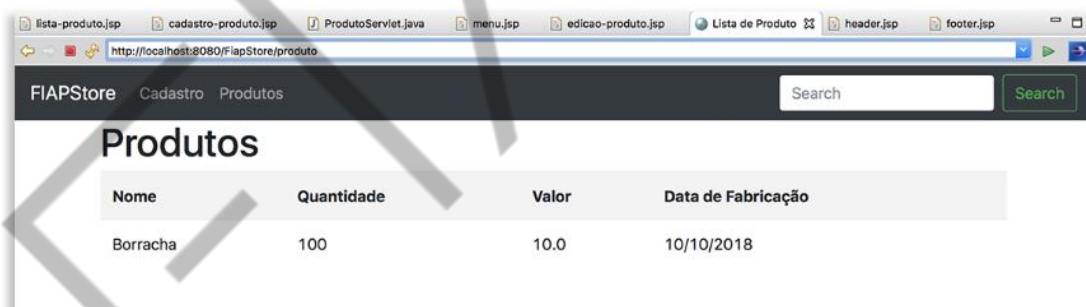
Código-fonte 21 – Menu da aplicação

Fonte: Elaborado pelo autor (2017)

Pronto! Agora podemos testar.

Execute novamente a página de cadastro e utilize o link para acessar a listagem. Caso tente executar a página de listagem diretamente, ocorrerá um erro, pois a página precisa da lista de produtos para ser construída, e a Servlet é responsável por enviar essa informação, por isso, para abrir a página de listagem, primeiro a Servlet deve ser executada.

O resultado da listagem pode ser visto na Figura “Teste de listagem de produtos”.



Nome	Quantidade	Valor	Data de Fabricação
Borracha	100	10.0	10/10/2018

Figura 25 – Teste de listagem de produtos

Fonte: Elaborado pelo autor (2018)

Na Figura “Visão geral da aplicação”, apresentamos a estrutura geral da aplicação, com todas as classes e páginas JSP.

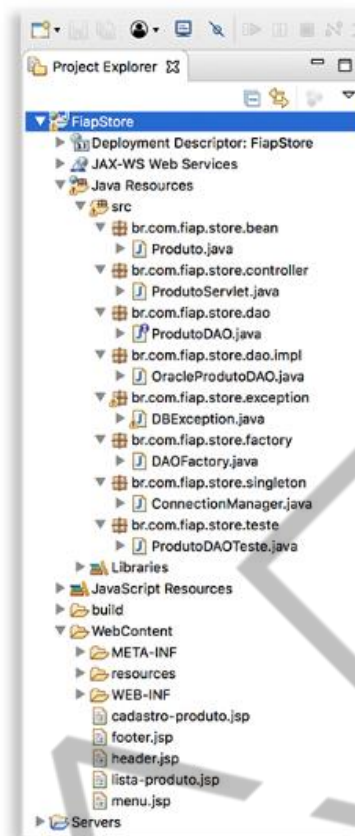


Figura 26 – Visão geral da aplicação
Fonte: Elaborado pelo autor (2017)

2.3 Editar

A funcionalidade de editar tem dois passos para o usuário. Primeiro ele deve selecionar o produto que será alterado, depois deve modificar e enviar as informações para a atualização. Para o usuário escolher o produto que será atualizado, vamos adicionar um botão ao lado de cada produto, na tela de listagem, conforme mostra a Figura “Botão para editar um produto”.

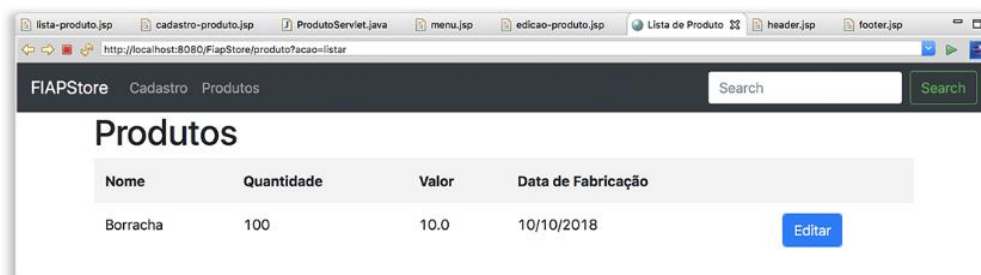


Figura 27 – Botão para editar um produto
Fonte: Elaborado pelo autor (2017)

Ajuste a página da listagem, adicione um link que deve enviar uma requisição para a Servlet com o código do produto que será alterado. Depois, a Servlet deve recuperar o produto por meio do código e enviar as informações para um formulário de edição, que deve vir preenchido com os dados.

Utilize a mesma Servlet para as funcionalidades do Produto, pois ficará mais bem estruturado e não gerará classes desnecessárias. Porém, vamos criar um link para acessar a Servlet, isso significa que utilizaremos o método doGet, mas esse método já é utilizado para a listagem.

O que devemos fazer então? Precisamos ajustar o código para enviar um novo valor para a Servlet, de modo a descrever qual ação deve ser realizada. O Código-fonte “Link para editar um produto” exibe a página de listagem e, em destaque, o *link* para editar um produto.

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib          prefix="c"
uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib          prefix="fmt"
uri="http://java.sun.com/jsp/jstl/fmt" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01
Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1">
<title>Cadastro de Produto</title>
<%@ include file="header.jsp" %>
</head>
<body>

<%@ include file="menu.jsp" %>
<div class="container">
<h1>Produtos</h1>
<table class="table table-striped">
<tr>
<th>Nome</th>
<th>Quantidade</th>
<th>Valor</th>
<th>Data de Fabricação</th>
<th></th>
```

```

        </tr>
        <c:forEach items="${produtos}" var="p">
            <tr>
                <td>${p.nome}</td>
                <td>${p.quantidade}</td>
                <td>${p.valor}</td>
                <td>
                    <fmt:formatDate
value="${p.dataFabricacao.time}" pattern="dd/MM/yyyy"/>
                </td>
                <td>
                    <c:url value="produto"
var="link">
                        <c:param name="acao"
value="abrir-form-edicao"/>
                        <c:param name="codigo"
value="${p.codigo}"/>
                    </c:url>
                    <a
href="${link}">Editar</a>
                </td>
            </tr>
        </c:forEach>
    </table>
</div>
<%@ include file="footer.jsp" %>

</body>
</html>

```

Código-fonte 22 – Link para editar um produto
Fonte: Elaborado pelo autor (2017)

Note que utilizamos a *tag* <c:url> para criar um *link* para o ProdutoServlet que envia dois parâmetros: o código do produto e a ação “abrir-form-edicao”.

Agora, precisamos ajustar a Servlet (Código-fonte “Método doGet ajustado para a ação de editar”) a fim de receber essa requisição, validar e executar a ação corretamente.

```

@Override
protected void doGet(HttpServletRequest request,
    HttpServletResponse response) throws ServletException,
    IOException {
    String acao = request.getParameter("acao");

```

```
switch (acao) {
    case "listar":
        List<Produto> lista = dao.listar();
        request.setAttribute("produtos", lista);
        request.getRequestDispatcher("lista-
produto.jsp").forward(request, response);
        break;
    case "abrir-form-edicao":
        int id =
Integer.parseInt(request.getParameter("codigo"));
        Produto produto = dao.buscar(id);
        request.setAttribute("produto", produto);
        request.getRequestDispatcher("edicao-
produto.jsp").forward(request, response);
    }
}
```

Código-fonte 23 – Método doGet ajustado para a ação de editar
Fonte: Elaborado pelo autor (2017)

Note que a primeira instrução do método é recuperar o parâmetro “acao”, para identificar se o usuário quer listar ou editar um produto. Depois, o método tem a instrução **switch-case**, para separar as duas lógicas. Se preferir, também podemos utilizar o if-else.

O código da listagem foi movido para dentro do **case** “listar”, enquanto a implementação do “abrir-form-edicao” recupera o código enviado pelo link, busca o produto pelo código no banco de dados, adiciona o resultado da pesquisa na requisição, como um atributo e, depois, encaminha o usuário para a página “edição-produto.jsp”.

A página de edição é muito parecida com a página de cadastro, uma das diferenças é o código do produto. Enquanto no cadastro o código é gerado pela *sequence*, na atualização o código deve ser o mesmo que já foi gerado, por isso, no formulário de edição, precisamos adicionar um campo oculto para armazenar o código do produto (Código-fonte “Página de edição de produto”).

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="fmt"
uri="http://java.sun.com/jsp/jstl/fmt" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01
Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1">
<title>Atualização de Produto</title>
<%@ include file="header.jsp" %>
</head>
<body>
<%@ include file="menu.jsp" %>
<div class="container">
    <h1>Edição de Produto</h1>
    <form action="produto" method="post">
        <input type="hidden" value="editar"
name="acao">
        <input type="hidden" value="${produto.codigo }"
name="codigo">
        <div class="form-group">
            <label for="id-nome">Nome</label>
            <input type="text" name="nome" id="id-
nome" class="form-control" value="${produto.nome }" >
        </div>
        <div class="form-group">
            <label for="id-valor">Valor</label>
            <input type="text" name="valor" id="id-
valor" class="form-control" value="${produto.valor }">
        </div>
        <div class="form-group">
            <label for="id-
quantidade">Quantidade</label>
            <input type="text" name="quantidade"
id="id-quantidade" class="form-control"
value="${produto.quantidade }">
        </div>
        <div class="form-group">
            <label for="id-fabricacao">Data de
Fabricação</label>
```

```
        <input type="text" name="fabricacao"
id="id-fabricacao" class="form-control"
        value='<fmt:formatDate
value="\${produto.dataFabricacao.time
pattern="dd/MM/yyyy"/>'>
    </div>
    <input type="submit" value="Salvar" class="btn
btn-primary">
    <a href="produto?acao=listar" class="btn btn-
danger">Cancelar</a>
    </form>
</div>
<%@ include file="footer.jsp" %>
</body>
</html>
```

Código-fonte 24 – Página de edição de produto
Fonte: Elaborado pelo autor (2017)

Outro detalhe que pode ser notado é o parâmetro ação (editar) que o formulário está enviando, da mesma forma que utilizamos um parâmetro para determinar se é preciso abrir a tela de listagem e edição no método doGet. Vamos enviar o parâmetro ação para o método **doPost**, a fim de identificar se deve ser realizado um cadastro ou uma atualização.

Implementamos também um link, com estilo de botão, para voltar à listagem (Cancelar). Esse link envia uma requisição para o ProdutoServlet, enviando a ação listar.

No método doPost, recupere o parâmetro “acao” e construa o **switch**. Para melhorar o código, vamos extrair o código da funcionalidade de cadastrar para um método privado na Servlet. A forma mais simples de fazer isso é selecionar todo o código que queremos extrair para um método, clicar com o botão direito do mouse e escolher Refactory > Extract Method, conforme mostra a Figura “Extraindo parte de código para um método – parte 1”.

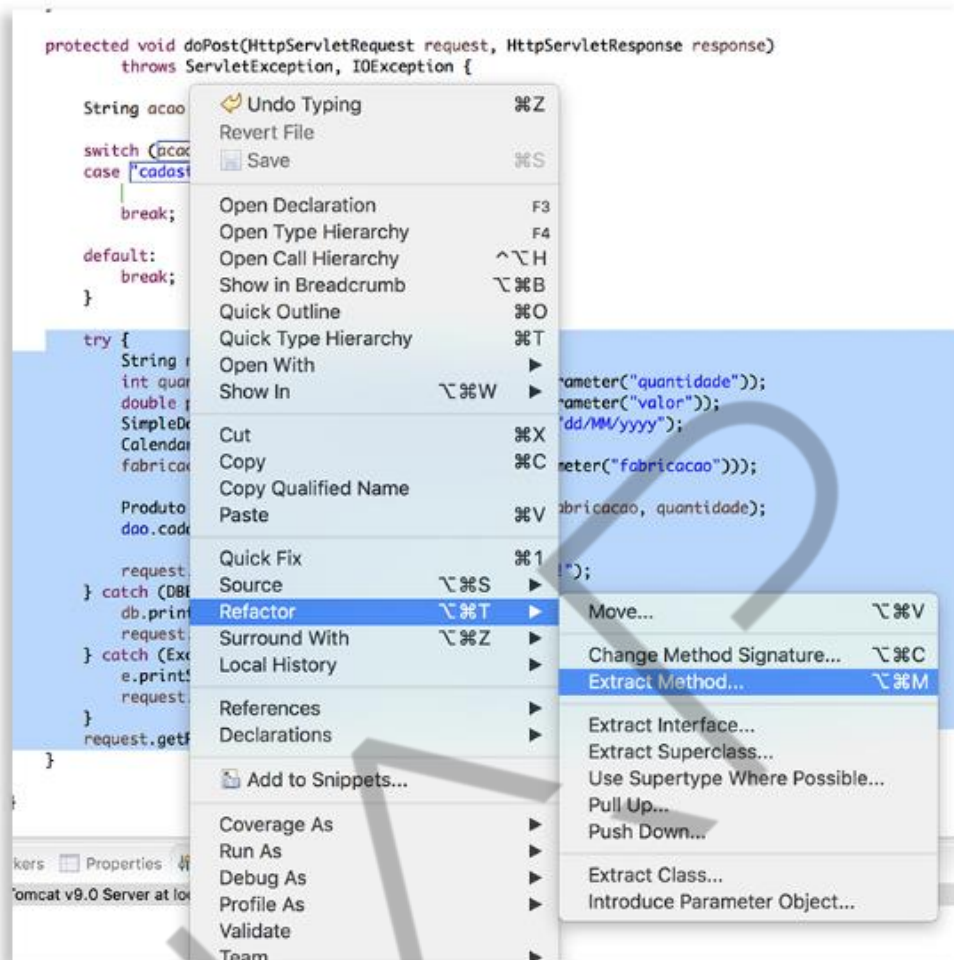


Figura 28 – Extraíndo parte de código para um método – parte 1

Fonte: Elaborado pelo autor (2017)

Depois, defina o nome do método e finalize o processo (Figura “Extraíndo parte de código para um método – parte 2”).

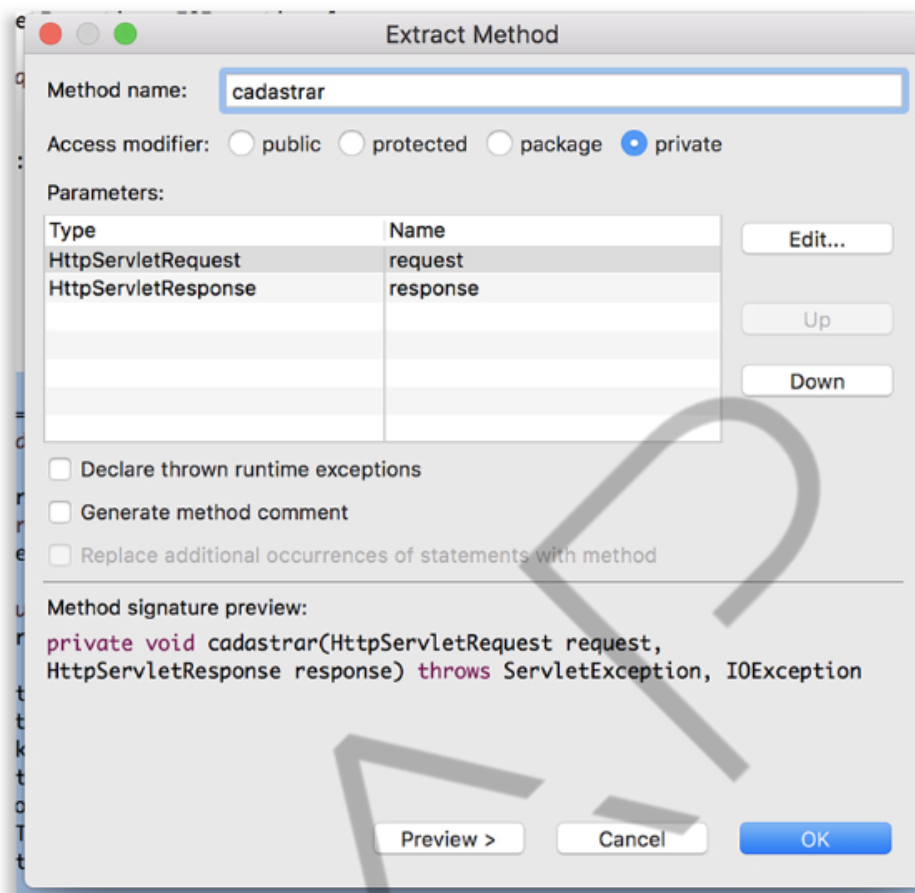


Figura 29 – Extraindo parte de código para um método – parte 2
Fonte: Elaborado pelo autor (2017)

Faça o mesmo para a funcionalidade de listar (Figura “Refactoring da Servlet”).

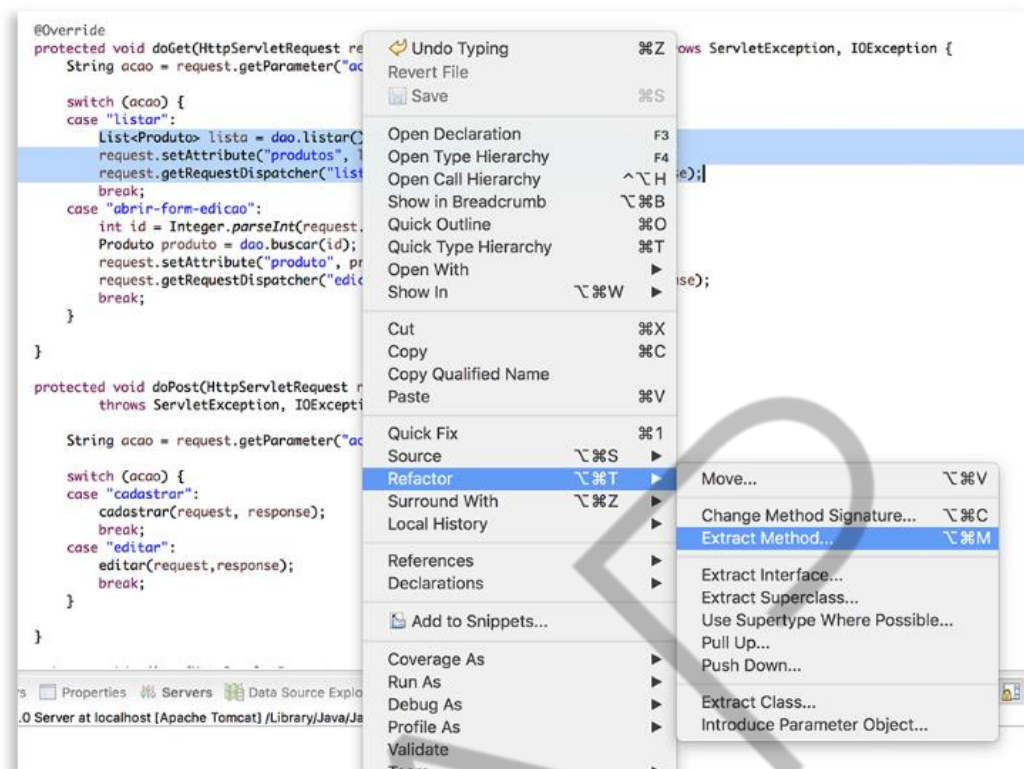


Figura 30 – Refactoring da Servlet
Fonte: Elaborado pelo autor (2017)

Implemente um método privado chamado *editar*, que recebe o *request* e *response* e lança as exceções *ServletException* e *IOException* (Código-fonte “Método de editar um produto”).

```
private void editar(HttpServletRequest request,
    HttpServletResponse response) throws ServletException,
    IOException {
    try {
        int codigo = Integer.parseInt(request.getParameter("codigo"));
        String nome = request.getParameter("nome");
        int quantidade = Integer.parseInt(request.getParameter("quantidade"));
        double preco = Double.parseDouble(request.getParameter("valor"));
        SimpleDateFormat format = new SimpleDateFormat("dd/MM/yyyy");
        Calendar fabricacao = Calendar.getInstance();
```

```
        fabricacao.setTime(format.parse(request.getParameter("fabricacao")));

        Produto produto = new Produto(codigo, nome, preco, fabricacao, quantidade);
        dao.atualizar(produto);

        request.setAttribute("msg", "Produto atualizado!");
    } catch (DBException db) {
        db.printStackTrace();
        request.setAttribute("erro", "Erro ao atualizar");
    } catch (Exception e) {
        e.printStackTrace();
        request.setAttribute("erro", "Por favor, valide os dados");
    }
    listar(request, response);
}
```

Código-fonte 25 – Método de editar um produto

Fonte: Elaborado pelo autor (2017)

O método **editar** recupera cada um dos parâmetros do formulário, realiza as conversões necessárias e instancia, com os dados, uma classe Produto. Depois, utiliza o DAO para atualizar o produto no banco de dados e adiciona uma mensagem de sucesso na requisição; por fim, chama o método listar, para encaminhar o usuário até a página de listagem depois da edição. Caso aconteça algum problema, uma mensagem de erro é adicionada na requisição e o usuário é encaminhado para a listagem. Talvez você possa melhorar essa parte, fazendo com que, caso aconteça um erro, o usuário continue na página de edição. Fica o desafio!

Para finalizar, apresentamos o código completo (Código-fonte “Métodos doPost, cadastrar e editar”), com o método **doPost** e os métodos privados que ele utiliza.

```
protected void doPost(HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException {
```

```
String acao = request.getParameter("acao");

switch (acao) {
case "cadastrar":
    cadastrar(request, response);
    break;
case "editar":
    editar(request, response);
    break;
}
}

private void editar(HttpServletRequest request,
    HttpServletResponse response) throws ServletException,
    IOException {
    try {
        int codigo =
Integer.parseInt(request.getParameter("codigo"));
        String nome =
request.getParameter("nome");
        int quantidade =
Integer.parseInt(request.getParameter("quantidade"));
        double preco =
Double.parseDouble(request.getParameter("valor"));
        SimpleDateFormat format = new
SimpleDateFormat("dd/MM/yyyy");
        Calendar fabricacao =
Calendar.getInstance();

        fabricacao.setTime(format.parse(request.getParameter("fa
bricacao")));

        Produto produto = new Produto(codigo, nome,
preco, fabricacao, quantidade);
        dao.atualizar(produto);

        request.setAttribute("msg", "Produto
atualizado!");
    } catch (DBException db) {
        db.printStackTrace();
        request.setAttribute("erro", "Erro ao
atualizar");
    }
}
```

```
        } catch (Exception e) {
            e.printStackTrace();
            request.setAttribute("erro", "Por favor,
valide os dados");
        }
        listar(request, response);
    }

    private void cadastrar(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        try {
            String nome =
request.getParameter("nome");
            int quantidade =
Integer.parseInt(request.getParameter("quantidade"));
            double preco =
Double.parseDouble(request.getParameter("valor"));
            SimpleDateFormat format = new
SimpleDateFormat("dd/MM/yyyy");
            Calendar fabricacao =
Calendar.getInstance();

            fabricacao.setTime(format.parse(request.getParameter("fa
bricacao")));

            Produto produto = new Produto(0, nome,
preco, fabricacao, quantidade);
            dao.cadastrar(produto);

            request.setAttribute("msg", "Produto
cadastrado!");
        } catch (DBException db) {
            db.printStackTrace();
            request.setAttribute("erro", "Erro ao
cadastrar");
        } catch (Exception e) {
            e.printStackTrace();
            request.setAttribute("erro", "Por favor,
valide os dados");
        }
        request.getRequestDispatcher("cadastro-
produto.jsp").forward(request, response);
    }
}
```

Código-fonte 26 – Métodos doPost, cadastrar e editar
Fonte: Elaborado pelo autor (2017)

O último ajuste para a funcionalidade de editar é adicionar a exibição das mensagens na página de listagem. Assim, adicione o seguinte trecho de código, conforme o Código-fonte “Exibição das mensagens de sucesso e erro na página JSP”.

```
<c:if test="${not empty msg }">
    <div class="alert alert-
success">${msg}</div>
</c:if>
<c:if test="${not empty erro }">
    <div class="alert alert-
danger">${erro}</div>
</c:if>
```

Código-fonte 27 – Exibição das mensagens de sucesso e erro na página JSP
Fonte: Elaborado pelo autor (2017)

Você pode exibir as mensagens em qualquer lugar na página. Vamos optar por adicionar o código logo acima da tabela, ou seja, acima da tag <table>.

Agora parece que está tudo pronto! Mas, espere, nós não modificamos os métodos da Servlet para receber um parâmetro que define a ação? Isso quer dizer que quebramos as funcionalidades de listar e cadastrar, já que elas não enviam esse parâmetro. Ajuste o link de listar no “menu.jsp” (Código-fonte “Link para a listagem atualizada”) e o formulário na página “cadastro-produto.jsp” (Código-fonte “Formulário de cadastro atualizado”).

```
<li class="nav-item">
    <a class="nav-link"
href="produto?acao=listar">Produtos</a>
</li>
```

Código-fonte 28 – Link para a listagem atualizada
Fonte: Elaborado pelo autor (2017)

No link, adicione o parâmetro por meio do caractere interrogação (?) seguido do nome do parâmetro e seu valor (acao=listar). Dessa forma, a Servlet receberá a ação no método **doGet**.

```
<form action="produto" method="post">
    <input type="hidden" value="cadastrar"
name="acao">
    <div class="form-group">
        <label for="id-nome">Nome</label>
        <input type="text" name="nome" id="id-
nome" class="form-control">
    </div>
    <div class="form-group">
        <label for="id-valor">Valor</label>
        <input type="text" name="valor" id="id-
valor" class="form-control">
    </div>
    <div class="form-group">
        <label for="id-
quantidade">Quantidade</label>
        <input type="text" name="quantidade"
id="id-quantidade" class="form-control">
    </div>
    <div class="form-group">
        <label for="id-fabricacao">Data de
Fabricação</label>
        <input type="text" name="fabricacao"
id="id-fabricacao" class="form-control">
    </div>
    <input type="submit" value="Salvar" class="btn
btn-primary">
</form>
```

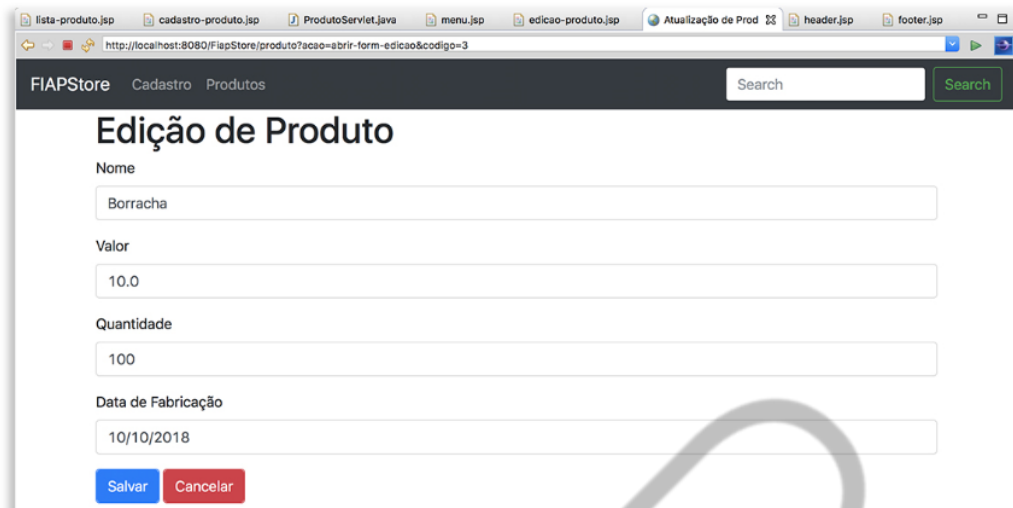
Código-fonte 29 – Formulário de cadastro atualizado

Fonte: Elaborado pelo autor (2017)

Para o cadastro, basta adicionar o campo oculto “acao” com o valor “cadastrar”, assim o método **doPost** saberá o que deve ser feito.

Agora teste todas as funcionalidades novamente.

No teste de edição, o usuário deve selecionar o produto por meio do link da listagem para ser encaminhado à edição com o formulário preenchido (Figura “Teste da funcionalidade de alterar um produto – parte 1”).



FIAPStore Cadastro Produtos

Edição de Produto

Nome
Borracha

Valor
10.0

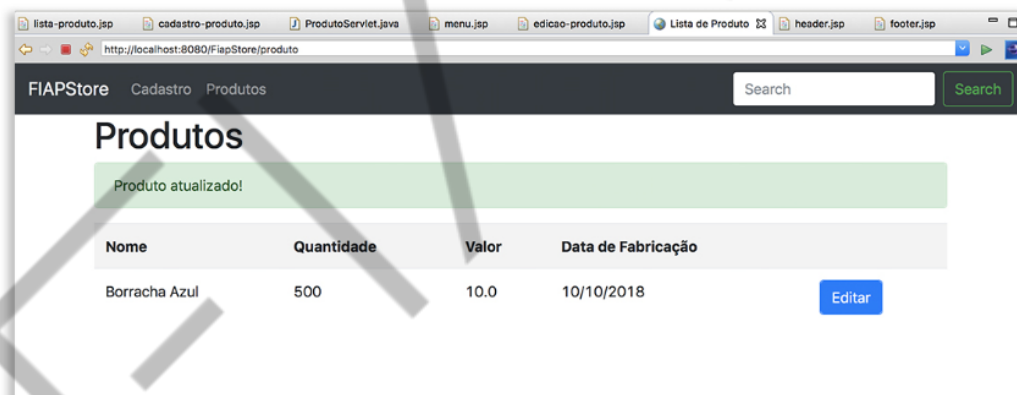
Quantidade
100

Data de Fabricação
10/10/2018

Salvar Cancelar

Figura 31 – Teste da funcionalidade de alterar um produto – Parte 1
Fonte: Elaborado pelo autor (2017)

Depois de modificar alguns valores, clique em “Salvar” para retornar à listagem com a mensagem de sucesso e o produto modificado (Figura “Teste da funcionalidade de alterar um produto – parte 2”).



FIAPStore Cadastro Produtos

Produtos

Produto atualizado!

Nome	Quantidade	Valor	Data de Fabricação
Borracha Azul	500	10.0	10/10/2018

Editar

Figura 32 – Teste da funcionalidade de alterar um produto – parte 2
Fonte: Elaborado pelo autor (2017)

2.4 Excluir

A última funcionalidade para fechar o CRUD (*Create, Read, Update e Delete*) é a remoção. Para isso, vamos adicionar um botão de excluir, ao lado do botão de alterar, na listagem.

Esse botão vai acionar uma caixa de diálogo para confirmar se o usuário realmente deseja excluir o produto. Nessa funcionalidade, é importante o usuário ter a chance de cancelar a operação, caso tenha clicado errado no botão.

Vamos utilizar o modal do Bootstrap para implementar o botão e o modal (caixa de confirmação). Seguindo o exemplo da documentação do Bootstrap 4, chegamos a esta implementação (Código-fonte “Página de listagem com a opção de remoção”):

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="c"
uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fmt"
uri="http://java.sun.com/jsp/jstl/fmt" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01
Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1">
<title>Lista de Produto</title>
<%@ include file="header.jsp" %>
</head>
<body>

<%@ include file="menu.jsp" %>
<div class="container">
<h1>Produtos</h1>
<c:if test="${not empty msg}">
<div class="alert alert-
success">${msg}</div>
</c:if>
<c:if test="${not empty erro}">
<div class="alert alert-
danger">${erro}</div>
</c:if>
<table class="table table-striped">
<tr>
<th>Nome</th>
<th>Quantidade</th>
<th>Valor</th>
<th>Data de Fabricação</th>
<th></th>
</tr>
<c:forEach items="${produtos}" var="p">
<tr>
```

```
 <td>${p.nome}</td>     <td>${p.quantidade}</td>     <td>${p.valor}</td>     <td>         <fmt:formatDate value="${p.dataFabricacao.time }" pattern="dd/MM/yyyy"/>     </td>     <td>         <c:url          value="produto" var="link">         <c:param      name="acao" value="abrir-form-edicao"/>         <c:param name="codigo" value="${p.codigo }"/>     </c:url>     <a              href="${link}" class="btn btn-primary btn-xs">Editar</a>         <button      type="button" class="btn  btn-danger  btn-xs" data-toggle="modal" data- target="#excluirModal" onclick="codigoExcluir.value = ${p.codigo}">             Excluir         </button>     </td> </tr> </c:forEach> </table> </div>  <%@ include file="footer.jsp" %>  <!-- Modal --> <div class="modal fade" id="excluirModal" tabindex="-1" role="dialog" aria-labelledby="exampleModalLabel" aria- hidden="true">     <div class="modal-dialog" role="document">         <div class="modal-content">             <div class="modal-header">                 <h5          class="modal-title" id="exampleModalLabel">Confirmação</h5>                 <button type="button" class="close" data- dismiss="modal" aria-label="Close">                     <span aria-hidden="true">&times;</span>                 </button>             </div>         </div>     </div> |
```

```
</div>
<div class="modal-body">
    Deseja realmente excluir o produto?
</div>
<div class="modal-footer">
    <form action="produto" method="post">
        <input type="hidden" name="acao"
value="excluir">
        <input type="hidden" name="codigo"
id="codigoExcluir">
        <button type="button" class="btn btn-
secondary" data-dismiss="modal">Cancelar</button>
        <button type="submit" class="btn btn-
danger">Excluir</button>
    </form>
</div>
</div>
</div>
</div>

</body>
</html>
```

Código-fonte 30 – Página de listagem com a opção de remoção
Fonte: Elaborado pelo autor (2017)

A parte destacada do código implementa o botão de excluir, que abre o modal e passa o id do produto para o *input* do formulário de exclusão, ao ser clicado. O atributo **target** da *tag button* referencia o id do *modal* que será aberto. O botão ganhou o evento **onclick**, para passar o id do produto para o *input* com o id **codigoExcluir**, via javascript.

Já o modal, este tem um formulário com dois campos ocultos, um para o código do produto (que é passado pelo botão que abre o modal) e outro para a ação de excluir.

Para o modal funcionar, precisamos adicionar uma dependência javascript necessária para o Bootstrap. Assim, adicione no arquivo “header.jsp”, logo abaixo do jQuery, a dependência do popper.js (Código-fonte “Ajustes nos javascripts do projeto”).

```
<script type="text/javascript" src="resources/js/jquery-3.2.1.min.js"></script>
<script
src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.11.0/
umd/popper.min.js" integrity="sha384-
b/U6ypiBEHpOf/4+1nzFpr53nxSS+GLCkfwBdFNTxtclqqenISfwAzpKaMNFN
mj4" crossorigin="anonymous"></script>
<script type="text/javascript"
src="resources/js/bootstrap.min
```

Código-fonte 31 – Ajustes nos javascripts do projeto
Fonte: Elaborado pelo autor (2017)

Para facilitar, dessa vez, nós utilizamos um arquivo javascript que está hospedado em algum servidor (CDN). Com isso, não precisamos baixar o arquivo e copiar no projeto.

Para finalizar a implementação da funcionalidade, adicione a ação de excluir no método doPost do ProdutoServlet (Código-fonte “Funcionalidade de excluir na Servlet”).

```
protected void doPost(HttpServletRequest request,
HttpServletRequest response)
    throws ServletException, IOException {

    String acao = request.getParameter("acao");

    switch (acao) {
        case "cadastrar":
            cadastrar(request, response);
            break;
        case "editar":
            editar(request, response);
            break;
        case "excluir":
            excluir(request, response);
            break;
    }
}

private void excluir(HttpServletRequest request,
HttpServletRequest response)
    throws ServletException, IOException {
```

```

        int                codigo
Integer.parseInt(request.getParameter("codigo"));
        try {
            dao.remove(codigo);
            request.setAttribute("msg", "Produto
removido!");
        } catch (DBException e) {
            e.printStackTrace();
            request.setAttribute("erro", "Erro ao
atualizar");
        }
        listar(request, response);
    }

```

Código-fonte 32 – Funcionalidade de excluir na Servlet
 Fonte: Elaborado pelo autor (2017)

Foi adicionado mais um *case* no *switch* do método *doPost* para o excluir. Essa opção chama o método *excluir*, que recupera o código do produto e utiliza o DAO para excluir o produto, passando o código. Depois adiciona uma mensagem de sucesso e encaminha o usuário para a listagem.

Caso aconteça algum problema, uma mensagem de erro é adicionada na requisição. Pronto! “Bora” para os testes! Liste os produtos e clique no botão excluir (Figura “Teste da funcionalidade de excluir um produto – parte 1”).

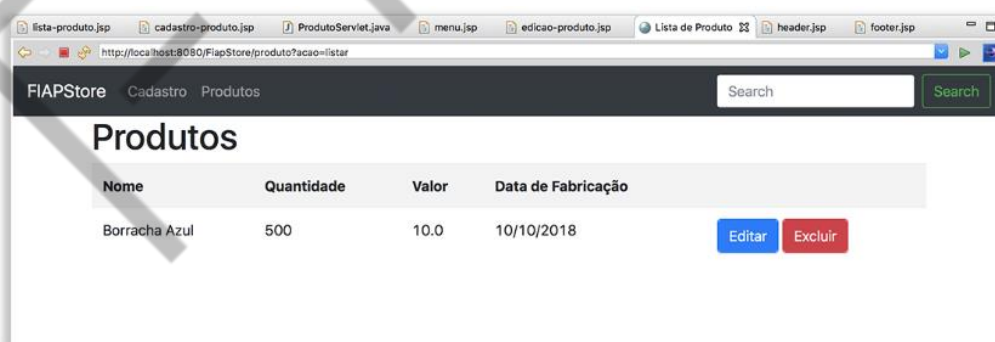


Figura 33 – Teste da funcionalidade de excluir um produto – parte 1
 Fonte: Elaborado pelo autor (2017)

A caixa de confirmação será aberta (Figura “Teste da funcionalidade de excluir um produto – parte 2”). Teste as opções de fechar (x) e cancelar, elas devem fechar o modal sem executar a ação. Depois, teste a opção excluir.



Figura 34 – Teste da funcionalidade de excluir um produto – parte 2
Fonte: Elaborado pelo autor (2017)

Após a confirmação, uma mensagem de sucesso deve aparecer e o produto não deverá ser listado na tabela (Figura “Teste da funcionalidade de excluir um produto – parte 3”).



Figura 35 – Teste da funcionalidade de excluir um produto – parte 3
Fonte: Elaborado pelo autor (2017)

Show! Tudo está funcionando. Para finalizar, vamos ver o código do ProdutoServlet completo (Código-fonte “Implementação completa do ProdutoServlet”):

```
package br.com.fiap.store.controller;

import java.io.IOException;
import java.text.SimpleDateFormat;
import java.util.Calendar;
import java.util.List;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import br.com.fiap.store.bean.Produto;
```

```
import br.com.fiap.store.dao.ProdutoDAO;
import br.com.fiap.store.exception.DBException;
import br.com.fiap.store.factory.DAOFactory;

@WebServlet("/produto")
public class ProdutoServlet extends HttpServlet {

    private static final long serialVersionUID = 1L;

    private ProdutoDAO dao;

    @Override
    public void init() throws ServletException {
        super.init();
        dao = DAOFactory.getProdutoDAO();
    }

    @Override
    protected void doGet(HttpServletRequest request,
        HttpServletResponse response) throws ServletException,
        IOException {
        String acao = request.getParameter("acao");

        switch (acao) {
            case "listar":
                listar(request, response);
                break;
            case "abrir-form-edicao":
                abrirFormEdicao(request, response);
                break;
        }
    }

    private void abrirFormEdicao(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        int id = Integer.parseInt(request.getParameter("codigo"));
        Produto produto = dao.buscar(id);
        request.setAttribute("produto", produto);
        request.getRequestDispatcher("edicao-
        produto.jsp").forward(request, response);
    }
}
```

```
private void listar(HttpServletRequest request,
HttpServletResponse response) throws ServletException,
IOException {
    List<Produto> lista = dao.listar();
    request.setAttribute("produtos", lista);
    request.getRequestDispatcher("lista-
produto.jsp").forward(request, response);
}

protected void doPost(HttpServletRequest request,
HttpServletResponse response)
    throws ServletException, IOException {

    String acao = request.getParameter("acao");

    switch (acao) {
    case "cadastrar":
        cadastrar(request, response);
        break;
    case "editar":
        editar(request, response);
        break;
    case "excluir":
        excluir(request, response);
        break;
    }
}

private void excluir(HttpServletRequest request,
HttpServletResponse response)
    throws ServletException, IOException {
    int codigo =
Integer.parseInt(request.getParameter("codigo"));
    try {
        dao.remover(codigo);
        request.setAttribute("msg", "Produto
removido!");
    } catch (DBException e) {
        e.printStackTrace();
        request.setAttribute("erro", "Erro ao
atualizar");
    }
    listar(request, response);
}
```



```

    }

    private void editar(HttpServletRequest request,
        HttpServletResponse response) throws ServletException,
        IOException {
        try {
            int            codigo            =
Integer.parseInt(request.getParameter("codigo"));
            String          nome            =
request.getParameter("nome");
            int            quantidade        =
Integer.parseInt(request.getParameter("quantidade"));
            double          preco            =
Double.parseDouble(request.getParameter("valor"));
            SimpleDateFormat format          = new
SimpleDateFormat("dd/MM/yyyy");
            Calendar        fabricacao        =
Calendar.getInstance();

            fabricacao.setTime(format.parse(request.getParameter("fa
bricacao")));

            Produto produto = new Produto(codigo, nome,
preco, fabricacao, quantidade);
            dao.atualizar(produto);

            request.setAttribute("msg",          "Produto
atualizado!");
        } catch (DBException db) {
            db.printStackTrace();
            request.setAttribute("erro",          "Erro ao
atualizar");
        } catch (Exception e) {
            e.printStackTrace();
            request.setAttribute("erro",          "Por favor,
valide os dados");
        }
        listar(request, response);
    }

    private void cadastrar(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        try {

```

```

        String                nome                =
request.getParameter("nome");

        int                quantidade                =
Integer.parseInt(request.getParameter("quantidade"));

        double                preco                =
Double.parseDouble(request.getParameter("valor"));

        SimpleDateFormat                format                =        new
SimpleDateFormat("dd/MM/yyyy");

        Calendar                fabricacao                =
Calendar.getInstance();

        fabricacao.setTime(format.parse(request.getParameter("fa
bricacao")));

        Produto produto = new Produto(0, nome,
preco, fabricacao, quantidade);
        dao.cadastrar(produto);

        request.setAttribute("msg",        "Produto
cadastrado!");
    } catch (DBException db) {
        db.printStackTrace();
        request.setAttribute("erro",        "Erro    ao
cadastrar");
    } catch (Exception e) {
        e.printStackTrace();
        request.setAttribute("erro",        "Por    favor,
valide os dados");
    }
    request.getRequestDispatcher("cadastro-
produto.jsp").forward(request, response);
}

}

```

Código-fonte 33 – Implementação completa do ProdutoServlet
 Fonte: Elaborado pelo autor (2017)

2.5 Relacionamentos

Terminamos as operações do CRUD para uma tabela, porém, em um banco de dados relacional, os relacionamentos entre as tabelas são muito comuns. Logo,

vamos relacionar a tabela TB_PRODUTO com a tabela TB_CATEGORIA, adicionando assim uma categoria ao produto.

2.6 Banco de dados

O primeiro passo será ajustar a base de dados. Crie a tabela de categorias e altere a tabela de produtos para adicionar a chave estrangeira (FK) para a tabela de categoria. O código-fonte a seguir apresenta o SQL para criar a tabela e a *sequence* de categoria:

```
CREATE TABLE TB_CATEGORIA (CD_CATEGORIA INT PRIMARY KEY,  
NM_CATEGORIA VARCHAR(50) NOT NULL);  
  
CREATE SEQUENCE SQ_TB_CATEGORIA MINVALUE 1 START WITH 1  
INCREMENT BY 1;
```

Código-fonte 34 – SQL para criar a tabela e a sequence de categoria
Fonte: Elaborado pelo autor (2017)

Já o Código-fonte “SQL para criar e alterar a tabela de produto” altera a tabela de produto de modo a adicionar a coluna para armazenar o código da categoria e adicionar a chave estrangeira.

```
ALTER TABLE TB_PRODUTO ADD CD_CATEGORIA INT;  
  
ALTER TABLE TB_PRODUTO ADD CONSTRAINT CD_CATEGORIA  
FOREIGN KEY (CD_CATEGORIA) REFERENCES  
TB_CATEGORIA(CD_CATEGORIA);
```

Código-fonte 35 – SQL para criar e alterar a tabela de produto
Fonte: Elaborado pelo autor (2017)

Como implementamos a operação de cadastro de produto, não vamos implementar o cadastro de categoria. Uma vez que as duas funcionalidades são muito parecidas, só muda o tipo do objeto que será cadastrado. Assim, vamos cadastrar as categorias disponíveis na aplicação diretamente do banco de dados, ou seja, vamos realizar uma carga de dados no banco, conforme o Código-fonte “SQL para inserir as categorias no banco de dados”.

```
INSERT INTO TB_CATEGORIA (CD_CATEGORIA, NM_CATEGORIA)  
VALUES (SQ_TB_CATEGORIA.NEXTVAL, 'Escritório');  
  
INSERT INTO TB_CATEGORIA (CD_CATEGORIA, NM_CATEGORIA)  
VALUES (SQ_TB_CATEGORIA.NEXTVAL, 'Móveis e Decoração');
```

```
INSERT INTO TB_CATEGORIA (CD_CATEGORIA, NM_CATEGORIA)
VALUES (SQ_TB_CATEGORIA.NEXTVAL, 'Livros');
INSERT INTO TB_CATEGORIA (CD_CATEGORIA, NM_CATEGORIA)
VALUES (SQ_TB_CATEGORIA.NEXTVAL, 'Informática');

commit;
```

Código-fonte 36 – SQL para inserir as categorias no banco de dados
Fonte: Elaborado pelo autor (2017)

Fique à vontade para cadastrar as categorias, só não se esqueça do *commit* no final, para efetuar o cadastro.

2.7 Java Bean e DAO

Crie o Java Bean para a categoria com dois atributos: código e nome, adicione os *gets* e *sets* e os construtores com todos os argumentos e o vazio (Código-fonte “Java Bean Categoria”).

```
package br.com.fiap.store.bean;

public class Categoria {

    private int codigo;

    private String nome;

    public Categoria() {
        super();
    }

    public Categoria(int codigo, String nome) {
        super();
        this.codigo = codigo;
        this.nome = nome;
    }

    public int getCodigo() {
        return codigo;
    }

    public void setCodigo(int codigo) {
        this.codigo = codigo;
    }
}
```

```
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }
}
```

Código-fonte 37 – Java Bean Categoria
Fonte: Elaborado pelo autor (2017)

Depois da classe Categoria, chegou o momento de criar o DAO. Como não vamos implementar o CRUD para a categoria, precisaremos somente da funcionalidade de listar, pois esta será utilizada para exibir as categorias disponíveis no cadastro/alteração de um produto. Crie a interface para o DAO e defina o método de listar (Código-fonte “Interface CategoriaDAO”).

```
package br.com.fiap.store.dao;

import java.util.List;
import br.com.fiap.store.bean.Categoria;

public interface CategoriaDAO {

    List<Categoria> listar();

}
```

Código-fonte 38 – Interface CategoriaDAO
Fonte: Elaborado pelo autor (2017)

Após a interface, crie a classe “OracleCategoriaDAO” no seu respectivo pacote e implemente a interface “CategoriaDAO”. O método de listar não tem segredo, é muito parecido com o método de listar produto (Código-fonte “Classe OracleCategoriaDAO”).

```
package br.com.fiap.store.dao.impl;

import java.sql.Connection;
```

```
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.List;

import br.com.fiap.store.bean.Categoria;
import br.com.fiap.store.dao.CategoriaDAO;
import br.com.fiap.store.singleton.ConnectionManager;

public class OracleCategoriaDAO implements CategoriaDAO{

    private Connection conexao;

    @Override
    public List<Categoria> listar() {
        List<Categoria> lista = new
        ArrayList<Categoria>();
        PreparedStatement stmt = null;
        ResultSet rs = null;
        try {
            conexao =
            ConnectionManager.getInstance().getConnection();
            stmt = conexao.prepareStatement("SELECT *
            FROM TB_CATEGORIA");
            rs = stmt.executeQuery();

            //Percorre todos os registros encontrados
            while (rs.next()) {
                int codigo =
                rs.getInt("CD_CATEGORIA");
                String nome =
                rs.getString("NM_CATEGORIA");
                Categoria categoria = new
                Categoria(codigo,nome);
                lista.add(categoria);
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }finally {
            try {
                stmt.close();
                rs.close();
                conexao.close();
            }
```

```
        } catch (SQLException e) {  
            e.printStackTrace();  
        }  
    }  
    return lista;  
}  
}
```

Código-fonte 39 – Classe OracleCategoriaDAO
Fonte: Elaborado pelo autor (2017)

Para finalizar, ajuste o “DAOFactory” adicionando um método que retorna uma instância do tipo “CategoriaDAO” (Código-fonte “Ajuste na classe DAOFactory”).

```
package br.com.fiap.store.factory;  
  
import br.com.fiap.store.dao.CategoriaDAO;  
import br.com.fiap.store.dao.ProdutoDAO;  
import br.com.fiap.store.dao.impl.OracleCategoriaDAO;  
import br.com.fiap.store.dao.impl.OracleProdutoDAO;  
  
public class DAOFactory {  
  
    public static ProdutoDAO getProdutoDAO() {  
        return new OracleProdutoDAO();  
    }  
  
    public static CategoriaDAO getCategoriaDAO() {  
        return new OracleCategoriaDAO();  
    }  
  
}
```

Código-fonte 40 – Ajuste na classe DAOFactory
Fonte: Elaborado pelo autor (2017)

O que falta agora? O teste. Vamos implementar uma classe para testar o listar categorias (Código-fonte “Classe de teste de CategoriaDAO”).

```
package br.com.fiap.store.teste;  
  
import java.util.List;  
  
import br.com.fiap.store.bean.Categoria;  
import br.com.fiap.store.dao.CategoriaDAO;
```

```
import br.com.fiap.store.factory.DAOFactory;

public class CategoriaDAOTeste {

    public static void main(String[] args) {
        CategoriaDAO dao =
        DAOFactory.getCategoriaDAO();

        List<Categoria> lista = dao.listar();
        for (Categoria categoria : lista) {
            System.out.println(categoria.getCodigo() +
            " " + categoria.getNome());
        }
    }
}
```

Código-fonte 41 – Classe de teste de CategoriaDAO

Fonte: Elaborado pelo autor (2017)

Lembre-se de que essa classe deve ser executada como “Java Application”, o resultado da execução pode ser visto na Figura “Resultado da execução da classe de teste”.



Figura 36 – Resultado da execução da classe de teste

Fonte: Elaborado pelo autor (2017)

Será que falta alguma coisa? Relacionar o produto com a categoria! Abra a classe Produto e adicione um atributo do tipo Categoria, conforme o Código-fonte “Classe Produto com a Categoria”. Não se esqueça dos métodos *get* e *set*.

```
package br.com.fiap.store.bean;

import java.util.Calendar;

public class Produto {

    private int codigo;
```



```
private String nome;

private double valor;

private Calendar dataFabricacao;

private int quantidade;

private Categoria categoria;

public Produto() {
    super();
}

public Produto(int codigo, String nome, double valor,
Calendar dataFabricacao, int quantidade) {
    super();
    this.codigo = codigo;
    this.nome = nome;
    this.valor = valor;
    this.dataFabricacao = dataFabricacao;
    this.quantidade = quantidade;
}

public int getCodigo() {
    return codigo;
}

public void setCodigo(int codigo) {
    this.codigo = codigo;
}

public String getNome() {
    return nome;
}

public void setNome(String nome) {
    this.nome = nome;
}

public double getValor() {
    return valor;
}
```

```
public void setValor(double valor) {  
    this.valor = valor;  
}  
  
public Calendar getDataFabricacao() {  
    return dataFabricacao;  
}  
  
public void setDataFabricacao(Calendar  
dataFabricacao) {  
    this.dataFabricacao = dataFabricacao;  
}  
  
public int getQuantidade() {  
    return quantidade;  
}  
  
public void setQuantidade(int quantidade) {  
    this.quantidade = quantidade;  
}  
  
public Categoria getCategory() {  
    return categoria;  
}  
  
public void setCategoria(Categoria categoria) {  
    this.categoria = categoria;  
}  
}
```

Código-fonte 42 – Classe Produto com a Categoria
Fonte: Elaborado pelo autor (2017)

Agora precisamos modificar os métodos do ProdutoDAO para adicionar esse novo atributo da classe Produto. Como só vamos alterar a implementação do método, não será necessário modificar a interface, vamos ajustar somente a classe.

O método **cadastrar** deve adicionar a coluna “CD_CATEGORIA” na query e enviar o código da categoria como parâmetro. O Código-fonte “Método cadastrar produto atualizado” exhibe as modificações necessárias, em destaque.

```

@Override
public void cadastrar(Produto produto) throws
DBException {
    PreparedStatement stmt = null;

    try {
        conexao =
        ConnectionManager.getInstance().getConnection();
        String sql = "INSERT INTO TB_PRODUTO
(CD_PRODUTO, NM_PRODUTO, QT_PRODUTO, VL_PRODUTO,
DT_FABRICACAO, CD_CATEGORIA) VALUES (SQ_TB_PRODUTO.NEXTVAL, ?,
?, ?, ?, ?)";

        stmt = conexao.prepareStatement(sql);
        stmt.setString(1, produto.getNome());
        stmt.setInt(2, produto.getQuantidade());
        stmt.setDouble(3, produto.getValor());
        java.sql.Date data = new
        java.sql.Date(produto.getDataFabricacao().getTimeInMillis());
        stmt.setDate(4, data);
        stmt.setInt(5,
        produto.getCategoria().getCodigo());

        stmt.executeUpdate();
    } catch (SQLException e) {
        e.printStackTrace();
        throw new DBException("Erro ao
cadastrar.");
    } finally {
        try {
            stmt.close();
            conexao.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}

```

Código-fonte 43 – Método cadastrar produto atualizado

Fonte: Elaborado pelo autor (2017)

O método atualizar também precisa ser atualizado, igual ao método cadastrar (Código-fonte “Método atualizar produto atualizado”).

```

@Override
public void atualizar(Produto produto) throws
DBException {
    PreparedStatement stmt = null;

    try {
        conexao =
        ConnectionManager.getInstance().getConnection();
        String sql = "UPDATE TB_PRODUTO SET
NM_PRODUTO = ?, QT_PRODUTO = ?, VL_PRODUTO = ?, DT_FABRICACAO
= ?, CD_CATEGORIA = ? WHERE CD_PRODUTO = ?";
        stmt = conexao.prepareStatement(sql);
        stmt.setString(1, produto.getNome());
        stmt.setInt(2, produto.getQuantidade());
        stmt.setDouble(3, produto.getValor());
        java.sql.Date data = new
        java.sql.Date(produto.getDataFabricacao().getTimeInMillis());
        stmt.setDate(4, data);
        stmt.setInt(5,
produto.getCategoria().getCodigo());
        stmt.setInt(6, produto.getCodigo());

        stmt.executeUpdate();
    } catch (SQLException e) {
        e.printStackTrace();
        throw new DBException("Erro ao
atualizar.");
    } finally {
        try {
            stmt.close();
            conexao.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}

```

Código-fonte 44 – Método atualizar produto atualizado
Fonte: Elaborado pelo autor (2017)

O método remover não precisa ser alterado. Porém, os dois métodos – buscar e listar – precisam recuperar também a categoria do produto. Para isso, vamos modificar a query para realizar um inner join e trazer as informações das duas tabelas. Além do objeto Produto, vamos instanciar também um objeto Categoria, preencher

seus valores e atribuir ao Produto. As modificações do método buscar são apresentadas no Código-fonte “Método buscar produto atualizado”.

```

@Override
public Produto buscar(int id) {
    Produto produto = null;
    PreparedStatement stmt = null;
    ResultSet rs = null;
    try {
        conexao
        ConnectionManager.getInstance().getConnection();
        stmt = conexao.prepareStatement("SELECT *
FROM      TB_PRODUTO      INNER      JOIN      TB_CATEGORIA      ON
TB_PRODUTO.CD_CATEGORIA = TB_CATEGORIA.CD_CATEGORIA WHERE
TB_PRODUTO.CD_PRODUTO = ?");
        stmt.setInt(1, id);
        rs = stmt.executeQuery();

        if (rs.next()){
            int codigo = rs.getInt("CD_PRODUTO");
            String nome
rs.getString("NM_PRODUTO");
            int qtd = rs.getInt("QT_PRODUTO");
            double valor
rs.getDouble("VL_PRODUTO");
            java.sql.Date data
rs.getDate("DT_FABRICACAO");
            Calendar dataFabricacao
Calendar.getInstance();

            dataFabricacao.setTimeInMillis(data.getTime());

            int codigoCategoria
rs.getInt("CD_CATEGORIA");
            String nomeCategoria
rs.getString("NM_CATEGORIA");

            produto = new Produto(codigo, nome,
valor, dataFabricacao, qtd);
            Categoria categoria = new
Categoria(codigoCategoria,nomeCategoria);
            produto.setCategoria(categoria);
        }

    } catch (SQLException e) {

```

```

        e.printStackTrace();
    } finally {
        try {
            stmt.close();
            rs.close();
            conexao.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
    return produto;
}

```

Código-fonte 45 – Método buscar produto atualizado
 Fonte: Elaborado pelo autor (2017)

Para finalizar o DAO, vamos atualizar o método listar da mesma forma que foi feito no método buscar.

```

@Override
public List<Produto> listar() {
    List<Produto> lista = new ArrayList<Produto>();
    PreparedStatement stmt = null;
    ResultSet rs = null;
    try {
        conexao
        ConnectionManager.getInstance().getConnection();
        stmt = conexao.prepareStatement("SELECT *
FROM    TB_PRODUTO    INNER    JOIN    TB_CATEGORIA    ON
TB_PRODUTO.CD_CATEGORIA = TB_CATEGORIA.CD_CATEGORIA");
        rs = stmt.executeQuery();

        //Percorre todos os registros encontrados
        while (rs.next()) {
            int codigo = rs.getInt("CD_PRODUTO");
            String nome
rs.getString("NM_PRODUTO");
            int qtd = rs.getInt("QT_PRODUTO");
            double valor
rs.getDouble("VL_PRODUTO");
            java.sql.Date data
rs.getDate("DT_FABRICACAO");
            Calendar dataFabricacao
Calendar.getInstance();

```

```
        dataFabricacao.setTimeInMillis(data.getTime());
        int codigoCategoria =
rs.getInt("CD_CATEGORIA");
        String nomeCategoria =
rs.getString("NM_CATEGORIA");

        Produto produto = new Produto(codigo,
nome, valor, dataFabricacao, qtd);
        Categoria categoria = new
Categoria(codigoCategoria,nomeCategoria);
        produto.setCategoria(categoria);
        lista.add(produto);
    }
} catch (SQLException e) {
    e.printStackTrace();
} finally {
    try {
        stmt.close();
        rs.close();
        conexao.close();
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
return lista;
}
```

Código-fonte 46 – Método listar produto atualizado
Fonte: Elaborado pelo autor (2017)

2.8 View e Controller

No *frontend*, primeiro vamos atualizar a funcionalidade de cadastrar produto para que a página exiba as categorias cadastradas em um *select* e o usuário possa escolher a categoria no momento de cadastrar um produto.

Como vamos precisar da lista de categorias, a primeira alteração na Servlet é adicionar a CategoriaDAO como um atributo e, no método init, obter uma instância do DAO e associar no atributo, que foi definido anteriormente (Código-fonte “Ajustes na classe ProdutoServlet”).

```
@WebServlet("/produto")
public class ProdutoServlet extends HttpServlet {

    private static final long serialVersionUID = 1L;

    private ProdutoDAO dao;
    private CategoriaDAO categoriaDao;

    @Override
    public void init() throws ServletException {
        super.init();
        dao = DAOFactory.getProdutoDAO();
        categoriaDao = DAOFactory.getCategoriaDAO();
    }

    //... Mais códigos..
}
```

Código-fonte 47 – Ajustes na classe ProdutoServlet
Fonte: Elaborado pelo autor (2017)

O segundo passo é modificar a Servlet para adicionar uma nova ação no método doGet, essa ação deve encaminhar para a página de cadastro, enviando a lista de categorias cadastradas. Dessa forma, a página JSP pode percorrer a lista de categorias e definir as opções do select. O Código-fonte “Método doGet da classe ProdutoServlet atualizado” mostra os métodos **doGet** e **abrirFormCadastro** do ProdutoServlet.

```
@Override
protected void doGet(HttpServletRequest request,
    HttpServletResponse response) throws ServletException,
    IOException {
    String acao = request.getParameter("acao");

    switch (acao) {
        case "listar":
            listar(request, response);
            break;
        case "abrir-form-edicao":
            abrirFormEdicao(request, response);
            break;
        case "abrir-form-cadastro":
            abrirFormCadastro(request, response);
    }
}
```



```

        break;
    }

}

private void abrirFormCadastro(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    List<Categoria> lista = categoriaDao.listar();
    request.setAttribute("categorias", lista);
    request.getRequestDispatcher("cadastro-
produto.jsp").forward(request, response);
}

```

Código-fonte 48 – Método doGet da classe ProdutoServlet atualizado
 Fonte: Elaborado pelo autor (2017)

O método **abrirFormCadastro** recupera a lista de Categorias cadastradas, por meio do CategoriaDAO, adiciona essa lista no atributo da *request* e encaminha o usuário para a página “cadastro-produto.jsp”.

A página JSP deve recuperar essa lista de categorias e “montar” o *select*, percorrendo a lista de categoria e criando uma <option> a cada iteração (Código-fonte “Página de cadastro de produto com as opções de categoria”).

```

<%@ page language="java" contentType="text/html;
charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="c"
uri="http://java.sun.com/jsp/jstl/core" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01
Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1">
<title>Cadastro de Produto</title>
<%@ include file="header.jsp" %>
</head>
<body>
<%@ include file="menu.jsp" %>
<div class="container">
    <h1>Cadastro de Produto</h1>

```

```

<c:if test="${not empty msg }">
    <div class="alert alert-success">${msg}</div>
</c:if>
<c:if test="${not empty erro }">
    <div class="alert alert-danger">${erro}</div>
</c:if>
<form action="produto" method="post">
    <input type="hidden" value="cadastrar"
name="acao">
    <div class="form-group">
        <label for="id-nome">Nome</label>
        <input type="text" name="nome" id="id-
nome" class="form-control">
    </div>
    <div class="form-group">
        <label for="id-valor">Valor</label>
        <input type="text" name="valor" id="id-
valor" class="form-control">
    </div>
    <div class="form-group">
        <label for="id-
quantidade">Quantidade</label>
        <input type="text" name="quantidade"
id="id-quantidade" class="form-control">
    </div>
    <div class="form-group">
        <label for="id-fabricacao">Data de
Fabricação</label>
        <input type="text" name="fabricacao"
id="id-fabricacao" class="form-control">
    </div>
    <div class="form-group">
        <label for="id-
categoria">Categoria</label>
        <select name="categoria" id="id-categoria"
class="form-control">
            <option value="0">Selecione</option>
            <c:forEach items="${categorias }"
var="c">
                <option value="${c.codigo }"
>${c.nome }</option>
            </c:forEach>
        </select>
    </div>

```

```

        <input type="submit" value="Salvar" class="btn
btn-primary">
    </form>
</div>
<%@ include file="footer.jsp" %>
</body>
</html>

```

Código-fonte 49 – Página de cadastro de produto com as opções de categoria
Fonte: Elaborado pelo autor (2017)

Com esse ajuste, o formulário envia um novo parâmetro: categoria, com o código da categoria escolhido pelo usuário. Com isso, atualize o método de cadastrar na Servlet, para recuperar esse código e adicionar uma categoria ao produto.

```

private void cadastrar(HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException {
    try {
        String nome =
request.getParameter("nome");
        int quantidade =
Integer.parseInt(request.getParameter("quantidade"));
        double preco =
Double.parseDouble(request.getParameter("valor"));
        SimpleDateFormat format = new
SimpleDateFormat("dd/MM/yyyy");
        Calendar fabricacao =
Calendar.getInstance();

        fabricacao.setTime(format.parse(request.getParameter("fa
bricacao")));
        int codigoCategoria =
Integer.parseInt(request.getParameter("categoria"));

        Categoria categoria = new Categoria();
        categoria.setCodigo(codigoCategoria);

        Produto produto = new Produto(0, nome,
preco, fabricacao, quantidade);
        produto.setCategoria(categoria);

        dao.cadastrar(produto);
    }
}

```

```

        request.setAttribute("msg", "Produto
cadastrado!");
    } catch (DBException db) {
        db.printStackTrace();
        request.setAttribute("erro", "Erro ao
cadastrar");
    } catch (Exception e) {
        e.printStackTrace();
        request.setAttribute("erro", "Por favor,
valide os dados");
    }
    abrirFormCadastro(request, response);
}

```

Código-fonte 50 – Método cadastrar da Servlet atualizado

Fonte: Elaborado pelo autor (2017)

Um detalhe da implementação é a última linha do método. Em vez de encaminhar diretamente o usuário para uma página, chamamos o método **abrirFormCadastro** para enviar novamente a lista de categorias e, depois, encaminhar para a página de cadastro de produto. Caso isso não seja feito, depois que o usuário cadastrar e for redirecionado para a página, não existirão as opções do *select*, já que a lista de categorias não foi enviada pela Servlet.

Você já deve ter percebido que não podemos mais ir diretamente até a página de cadastro, precisamos acionar primeiro a Servlet para depois sermos encaminhados para a página. Portanto, ajuste o link de cadastrar no menu (Código-fonte “Link do menu atualizado”).

```

<a class="nav-link" href="produto?acao=abrir-form-
cadastro">Cadastro</a>

```

Código-fonte 51 – Link do menu atualizado

Fonte: Elaborado pelo autor (2017)

O valor do parâmetro “acao” deve ser igual ao valor no “case” do método **doPost**. Agora já podemos testar. A Figura “Opções de categorias” mostra as opções do *select*, que foram carregadas corretamente.

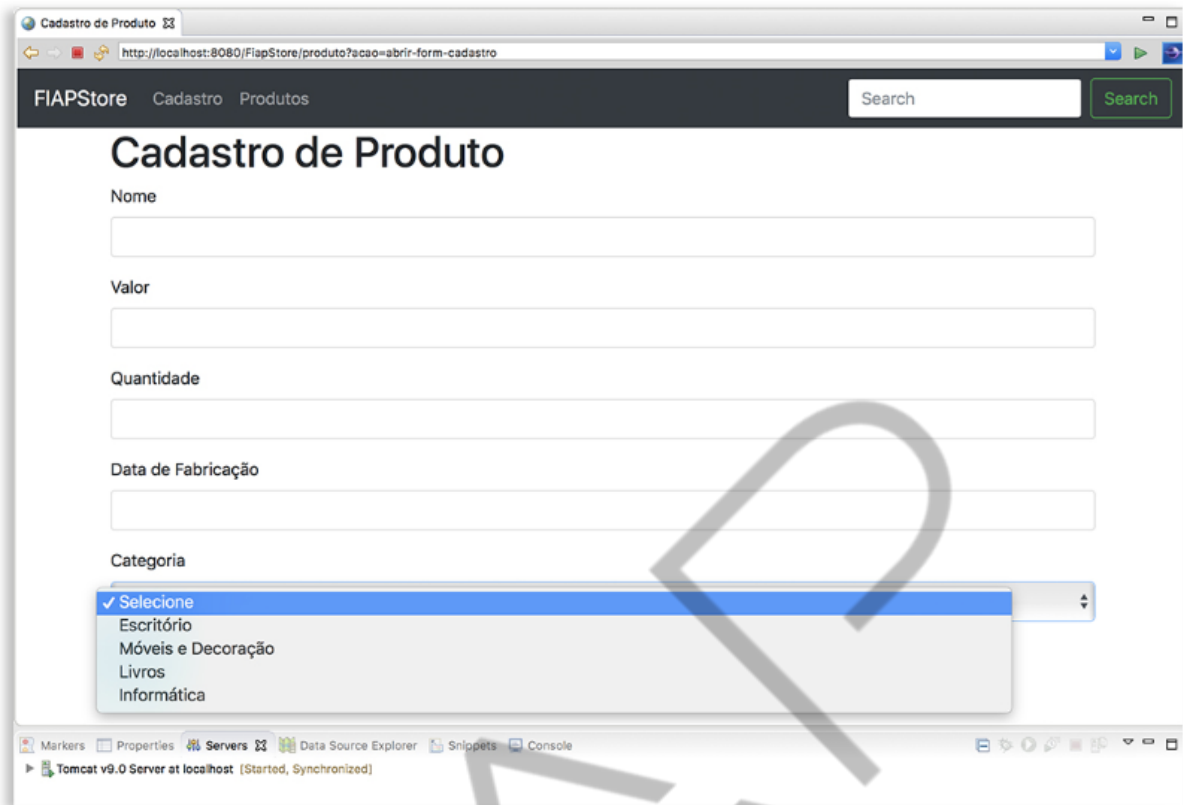


Figura 37 – Opções de categorias
Fonte: Elaborado pelo autor (2017)

Para exibir a categoria do produto na listagem, precisamos apenas modificar a página JSP. Vá até a página “lista-produto.jsp” e adicione uma coluna na tabela, para a categoria. O Código-fonte “Exibindo a categoria do produto na listagem” destaca as modificações na implementação.

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="c"
uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fmt"
uri="http://java.sun.com/jsp/jstl/fmt" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01
Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1">
<title>Lista de Produto</title>
<%@ include file="header.jsp" %>
</head>
<body>
```

```

<%@ include file="menu.jsp" %>
<div class="container">
  <h1>Produtos</h1>
  <c:if test="${not empty msg}">
    <div class="alert alert-
success">${msg}</div>
  </c:if>
  <c:if test="${not empty erro}">
    <div class="alert alert-
danger">${erro}</div>
  </c:if>
  <table class="table table-striped">
    <tr>
      <th>Nome</th>
      <th>Quantidade</th>
      <th>Valor</th>
      <th>Data de Fabricação</th>
      <th>Categoria</th>
      <th></th>
    </tr>
    <c:forEach items="${produtos}" var="p">
      <tr>
        <td>${p.nome}</td>
        <td>${p.quantidade}</td>
        <td>${p.valor}</td>
        <td>
          <fmt:formatDate
value="${p.dataFabricacao.time}" pattern="dd/MM/yyyy"/>
        </td>
        <td>${p.categoria.nome}</td>
        <td>
          <c:url value="produto"
var="link">
          <c:param name="acao"
value="abrir-form-edicao"/>
          <c:param name="codigo"
value="${p.codigo}"/>
        </c:url>
        <a href="${link}"
class="btn btn-primary btn-xs">Editar</a>
        <button type="button"
class="btn btn-danger btn-xs" data-toggle="modal" data-
target="#excluirModal" onclick="codigoExcluir.value =
${p.codigo}">
          Excluir
        </button>
      </td>
    </c:forEach>
  </table>

```

```

        </tr>
      </c:forEach>
    </table>
  </div>

  <%@ include file="footer.jsp" %>

  <!-- Modal -->
  <div class="modal fade" id="excluirModal" tabindex="-1"
  role="dialog" aria-labelledby="exampleModalLabel" aria-
  hidden="true">
    <div class="modal-dialog" role="document">
      <div class="modal-content">
        <div class="modal-header">
          <h5 class="modal-title"
  id="exampleModalLabel">Confirmação</h5>
          <button type="button" class="close" data-
  dismiss="modal" aria-label="Close">
            <span aria-hidden="true">&times;</span>
          </button>
        </div>
        <div class="modal-body">
          Deseja realmente excluir o produto?
        </div>
        <div class="modal-footer">
          <form action="produto" method="post">
            <input type="hidden" name="acao"
  value="excluir">
            <input type="hidden" name="codigo"
  id="codigoExcluir">
            <button type="button" class="btn btn-
  secondary" data-dismiss="modal">Cancelar</button>
            <button type="submit" class="btn btn-
  danger">Excluir</button>
          </form>
        </div>
      </div>
    </div>
  </div>

</body>
</html>

```

Código-fonte 52 – Exibindo a categoria do produto na listagem

Fonte: Elaborado pelo autor (2017)

O resultado da execução da listagem é exibido na Figura “Listagem de produtos com a categoria”.

Nome	Quantidade	Valor	Data de Fabricação	Categoria
Celular	1	1000.0	15/10/2014	Informática
Caderno	100	20.0	16/09/2017	Escritório

Figura 38 – Listagem de produtos com a categoria
Fonte: Elaborado pelo autor (2017)

O último ajuste necessário é na funcionalidade de atualização de um produto. O sistema deve permitir que o usuário modifique a categoria de um produto. Da mesma forma que no cadastro, a Servlet precisa enviar a lista de categorias para a tela de edição (Código-fonte “Método de carregar a lista de categoria na Servlet”).

```

private void abrirFormCadastro(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    carregarOpcoesCategoria(request);
    request.getRequestDispatcher("cadastro-
produto.jsp").forward(request, response);
}

private void carregarOpcoesCategoria(HttpServletRequest request) {
    List<Categoria> lista = categoriaDao.listar();
    request.setAttribute("categorias", lista);
}

private void abrirFormEdicao(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    int id = Integer.parseInt(request.getParameter("codigo"));
    Produto produto = dao.buscar(id);
    request.setAttribute("produto", produto);
    carregarOpcoesCategoria(request);
    request.getRequestDispatcher("edicao-
produto.jsp").forward(request, response);
}

```


}

Código-fonte 53 – Método de carregar a lista de categoria na Servlet
 Fonte: Elaborado pelo autor (2017)

Observe que, para o método **carregarOpcoesCategoria**, extraímos o código que se repete, tanto no cadastro quanto na edição. Assim, os dois métodos de abrir a página (cadastro e atualização) utilizam o método de carregar a lista de categoria.

O próximo passo é modificar a tela de edição (Código-fonte “Tela de atualização de produto atualizada”).

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="c"
uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fmt"
uri="http://java.sun.com/jsp/jstl/fmt" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01
Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1">
<title>Atualização de Produto</title>
<%@ include file="header.jsp" %>
</head>
<body>
<%@ include file="menu.jsp" %>
<div class="container">
<h1>Edição de Produto</h1>
<form action="produto" method="post">
<input type="hidden" value="editar"
name="acao">
<input type="hidden" value="${produto.codigo }"
name="codigo">
<div class="form-group">
<label for="id-nome">Nome</label>
<input type="text" name="nome" id="id-
nome" class="form-control" value="${produto.nome }" >
</div>
<div class="form-group">
```

```

        <label for="id-valor">Valor</label>
        <input type="text" name="valor" id="id-
valor" class="form-control" value="{produto.valor }">
    </div>
    <div class="form-group">
        <label                                for="id-
quantidade">Quantidade</label>
        <input type="text" name="quantidade"
id="id-quantidade" class="form-control"
value="{produto.quantidade }">
    </div>
    <div class="form-group">
        <label for="id-fabricacao">Data    de
Fabricação</label>
        <input type="text" name="fabricacao"
id="id-fabricacao" class="form-control"
value='{<fmt:formatDate
value="{produto.dataFabricacao.time                }"
pattern="dd/MM/yyyy"/>'>
    </div>
    <div class="form-group">
        <label                                for="id-
categoria">Categoria</label>
        <select name="categoria" id="id-categoria"
class="form-control">
            <option value="0">Selecione</option>
            <c:forEach items="{categorias    }"
var="c">
                <c:if test="{c.codigo    ==
produto.categoria.codigo }">
                    <option value="{c.codigo
}" selected>{c.nome }</option>
                </c:if>
                <c:if test="{c.codigo    !=
produto.categoria.codigo }">
                    <option value="{c.codigo
}">{c.nome }</option>
                </c:if>
            </c:forEach>
        </select>
    </div>
    <input type="submit" value="Salvar" class="btn
btn-primary">

```

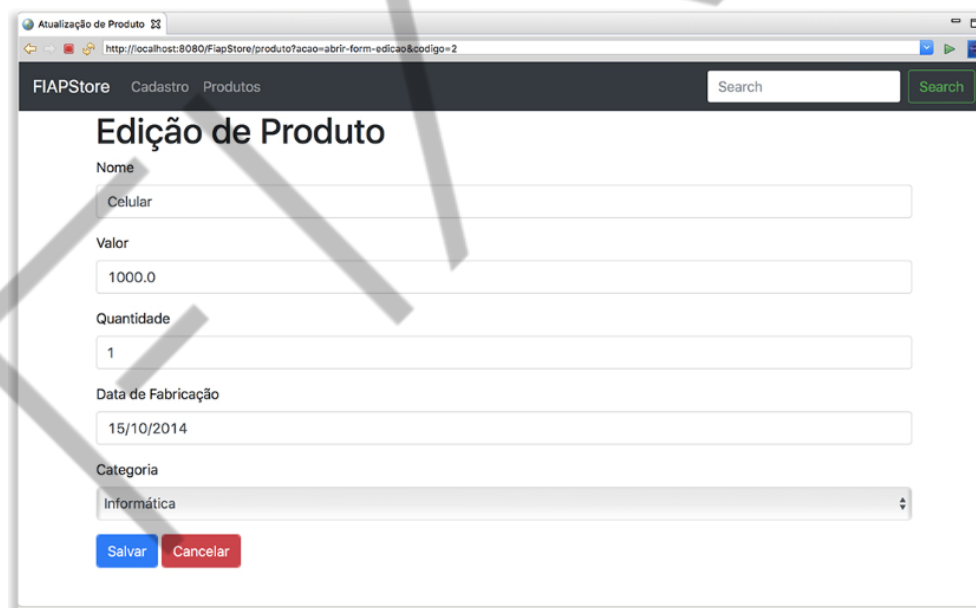
```
<a href="produto?acao=listar" class="btn btn-danger">Cancelar</a>
</form>
</div>
<%@ include file="footer.jsp" %>
</body>
</html>
```

Código-fonte 54 – Tela de atualização de produto atualizada
Fonte: Elaborado pelo autor (2017)

Nesta tela, adicionamos o campo categoria de forma muito parecida com a da tela de cadastro. A diferença é que fizemos uma comparação para trazer a categoria do produto selecionada.

Vamos testar!

Navegue até o formulário de edição de um produto (Figura “Listagem de produtos com a categoria”).



A imagem mostra uma interface web para a edição de um produto. No topo, há uma barra de navegação com o nome 'FIAPStore' e links para 'Cadastro' e 'Produtos'. Um campo de busca com o botão 'Search' está no canto superior direito. O título principal da página é 'Edição de Produto'. Abaixo dele, há um formulário com os seguintes campos: 'Nome' com o valor 'Celular', 'Valor' com '1000.0', 'Quantidade' com '1', 'Data de Fabricação' com '15/10/2014' e 'Categoria' com 'Informática' selecionada em uma lista suspensa. Na base do formulário, há dois botões: 'Salvar' (azul) e 'Cancelar' (vermelho).

Figura 39 – Listagem de produtos com a categoria
Fonte: Elaborado pelo autor (2017)

Perfeito! O formulário é carregado com as informações do produto e a opção de categoria já vem selecionada corretamente. Porém, se tentar atualizar, não dará certo ainda, pois não modificamos o método de atualizar na Servlet.

Na Servlet, vamos modificar o método de atualização para recuperar o código da categoria selecionada (Código-fonte “Tela de atualização de produto atualizada”).

```

        private void editar(HttpServletRequest request,
        HttpServletResponse response) throws ServletException,
        IOException {

            try {

                int                codigo                =
Integer.parseInt(request.getParameter("codigo"));
                String            nome                =
request.getParameter("nome");
                int                quantidade            =
Integer.parseInt(request.getParameter("quantidade"));
                double            preco                =
Double.parseDouble(request.getParameter("valor"));
                SimpleDateFormat    format            = new
SimpleDateFormat("dd/MM/yyyy");
                Calendar            fabricacao          =
Calendar.getInstance();

                fabricacao.setTime(format.parse(request.getParameter("fa
bricacao")));

                int                codigoCategoria      =
Integer.parseInt(request.getParameter("categoria"));

                Categoria categoria = new Categoria();
                categoria.setCodigo(codigoCategoria);

                Produto produto = new Produto(codigo, nome,
preco, fabricacao, quantidade);
                produto.setCategoria(categoria);
                dao.atualizar(produto);

                request.setAttribute("msg",            "Produto
atualizado!");
            } catch (DBException db) {
                db.printStackTrace();
                request.setAttribute("erro",          "Erro    ao
atualizar");
            } catch (Exception e) {
                e.printStackTrace();
                request.setAttribute("erro",          "Por    favor,
valide os dados");
            }
            listar(request, response);
        }

```

Código-fonte 55 – Método atualizar da Servlet atualizada
Fonte: Elaborado pelo autor (2017)

Agora, sim, podemos testar o funcionamento da alteração por completo! (Figura “Teste da alteração da categoria de um produto”).

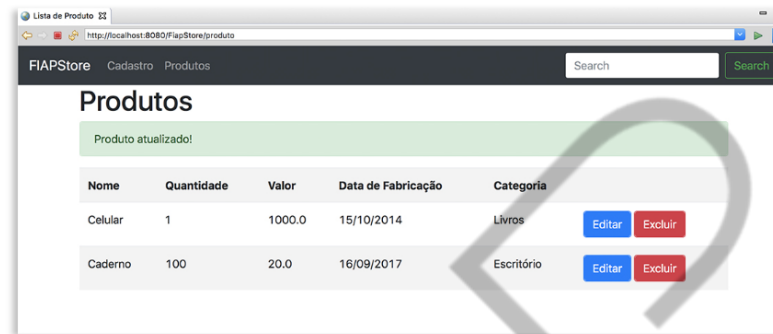


Figura 40 – Teste da alteração da categoria de um produto
Fonte: Elaborado pelo autor (2017)

A listagem completa do código do ProdutoServlet é apresentada a seguir (Código-fonte “Código completo do ProdutoServlet atualizado”).

```
package br.com.fiap.store.controller;

import java.io.IOException;
import java.text.SimpleDateFormat;
import java.util.Calendar;
import java.util.List;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import br.com.fiap.store.bean.Categoria;
import br.com.fiap.store.bean.Produto;
import br.com.fiap.store.dao.CategoriaDAO;
import br.com.fiap.store.dao.ProdutoDAO;
import br.com.fiap.store.exception.DBException;
import br.com.fiap.store.factory.DAOFactory;

@WebServlet("/produto")
public class ProdutoServlet extends HttpServlet {
```

```
private static final long serialVersionUID = 1L;

private ProdutoDAO dao;
private CategoriaDAO categoriaDao;

@Override
public void init() throws ServletException {
    super.init();
    dao = DAOFactory.getProdutoDAO();
    categoriaDao = DAOFactory.getCategoriaDAO();
}

@Override
protected void doGet(HttpServletRequest request,
    HttpServletResponse response) throws ServletException,
    IOException {
    String acao = request.getParameter("acao");

    switch (acao) {
        case "listar":
            listar(request, response);
            break;
        case "abrir-form-edicao":
            abrirFormEdicao(request, response);
            break;
        case "abrir-form-cadastro":
            abrirFormCadastro(request, response);
            break;
    }
}

private void abrirFormCadastro(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    carregarOpcoesCategoria(request);
    request.getRequestDispatcher("cadastro-
produto.jsp").forward(request, response);
}

private void carregarOpcoesCategoria(HttpServletRequest request) {
    List<Categoria> lista = categoriaDao.listar();
```

```
        request.setAttribute("categorias", lista);
    }

    private void abrirFormEdicao(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        int id = Integer.parseInt(request.getParameter("codigo"));
        Produto produto = dao.buscar(id);
        request.setAttribute("produto", produto);
        carregarOpcoesCategoria(request);
        request.getRequestDispatcher("edicao-
        produto.jsp").forward(request, response);
    }

    private void listar(HttpServletRequest request,
        HttpServletResponse response) throws ServletException,
        IOException {
        List<Produto> lista = dao.listar();
        request.setAttribute("produtos", lista);
        request.getRequestDispatcher("lista-
        produto.jsp").forward(request, response);
    }

    protected void doPost(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        String acao = request.getParameter("acao");

        switch (acao) {
            case "cadastrar":
                cadastrar(request, response);
                break;
            case "editar":
                editar(request, response);
                break;
            case "excluir":
                excluir(request, response);
                break;
        }
    }
}
```

```

        private void excluir(HttpServletRequest request,
        HttpServletResponse response)
            throws ServletException, IOException {
            int                codigo                =
Integer.parseInt(request.getParameter("codigo"));
            try {
                dao.remover(codigo);
                request.setAttribute("msg",        "Produto
removido!");
            } catch (DBException e) {
                e.printStackTrace();
                request.setAttribute("erro",        "Erro    ao
atualizar");
            }
            listar(request, response);
        }

        private void editar(HttpServletRequest request,
        HttpServletResponse response) throws ServletException,
        IOException {
            try {
                int                codigo                =
Integer.parseInt(request.getParameter("codigo"));
                String                nome                =
request.getParameter("nome");
                int                quantidade                =
Integer.parseInt(request.getParameter("quantidade"));
                double                preco                =
Double.parseDouble(request.getParameter("valor"));
                SimpleDateFormat        format        =        new
SimpleDateFormat("dd/MM/yyyy");
                Calendar                fabricacao                =
Calendar.getInstance();

                fabricacao.setTime(format.parse(request.getParameter("fa
bricacao")));
                int                codigoCategoria                =
Integer.parseInt(request.getParameter("categoria"));

                Categoria categoria = new Categoria();
                categoria.setCodigo(codigoCategoria);

```



```

        Produto produto = new Produto(codigo, nome,
preco, fabricacao, quantidade);
        produto.setCategoria(categoria);
        dao.atualizar(produto);

        request.setAttribute("msg", "Produto
atualizado!");
    } catch (DBException db) {
        db.printStackTrace();
        request.setAttribute("erro", "Erro ao
atualizar");
    } catch (Exception e) {
        e.printStackTrace();
        request.setAttribute("erro", "Por favor,
valide os dados");
    }
    listar(request, response);
}

private void cadastrar(HttpServletRequest request,
HttpServletResponse response)
    throws ServletException, IOException {
    try {
        String nome =
request.getParameter("nome");
        int quantidade =
Integer.parseInt(request.getParameter("quantidade"));
        double preco =
Double.parseDouble(request.getParameter("valor"));
        SimpleDateFormat format = new
SimpleDateFormat("dd/MM/yyyy");
        Calendar fabricacao =
Calendar.getInstance();

        fabricacao.setTime(format.parse(request.getParameter("fa
bricacao")));
        int codigoCategoria =
Integer.parseInt(request.getParameter("categoria"));

        Categoria categoria = new Categoria();
        categoria.setCodigo(codigoCategoria);

        Produto produto = new Produto(0, nome,
preco, fabricacao, quantidade);

```

```
        produto.setCategoria(categoria);

        dao.cadastrar(produto);

        request.setAttribute("msg", "Produto
cadastrado!");
    } catch (DBException db) {
        db.printStackTrace();
        request.setAttribute("erro", "Erro ao
cadastrar");
    } catch (Exception e) {
        e.printStackTrace();
        request.setAttribute("erro", "Por favor,
valide os dados");
    }
    abrirFormCadastro(request, response);
}
```

Código-fonte 56 – Código completo do ProdutoServlet atualizado
Fonte: Elaborado pelo autor (2017)

A funcionalidade de remoção não precisa de ajuste, pois ela funciona normalmente.

2.9 Login

A maioria das aplicações tem uma forma de autenticar o usuário. Por isso vamos implementar essa funcionalidade em nosso sistema. Após o usuário se autenticar, um e-mail será enviado para notificá-lo do evento. Outro detalhe é que não podemos armazenar a senha diretamente no banco de dados. É preciso criptografá-la antes de salvar definitivamente na base de dados.

2.10 Banco de dados

Para salvar as informações do usuário, crie uma tabela com as colunas de e-mail, que será o username, ou seja, o identificador do usuário na aplicação, e a senha. O e-mail será a chave primária da tabela de usuário. Claro que podemos gravar mais informações, porém, para o nosso exemplo, vamos trabalhar com as informações

básicas. O código SQL exibe o comando para criar a tabela de usuário (Código-fonte “SQL para criar a tabela de usuário”).

```
CREATE TABLE TB_USUARIO (DS_EMAIL VARCHAR(150) PRIMARY  
KEY, DS_SENHA VARCHAR(50) NOT NULL);
```

Código-fonte 57 – SQL para criar a tabela de usuário
Fonte: Elaborado pelo autor (2017)

2.11 Criptografia

Por motivos de segurança, não vamos salvar diretamente a senha do usuário na base de dados. Então, aplicaremos um algoritmo de criptografia para trabalhar com a senha do usuário. Existem vários algoritmos, como MD5, SHA-1, SHA-2 etc.

Em nosso sistema, vamos trabalhar com o MD5, um algoritmo de criptografia de 128 bits, unidirecional desenvolvido pela RSA Data Security, Inc. Um algoritmo unidirecional gera um *hash* que não pode ser transformado no texto que lhe deu origem. Dessa forma, vamos armazenar o *hash* na base de dados e, para comparar a senha, no momento do login, vamos transformar a senha informada e verificar a igualdade dos *hashes*.

Crie uma classe chamada “CriptografiaUtils” no pacote “br.com.fiap.store.util” para implementar um método que criptografa a senha com o algoritmo MD5 (Código-fonte “Classe utilitária para criptografar as senhas”).

```
package br.com.fiap.store.util;  
  
import java.math.BigInteger;  
import java.security.MessageDigest;  
  
public class CriptografiaUtils {  
  
    public static String criptografar(String senha)  
throws Exception {  
        //Obtém a instância de um algoritmo  
        MessageDigest md=  
MessageDigest.getInstance("MD5");  
        //Passa os dados a serem criptografados e  
estipula encoding padrão
```

```
        md.update(senha.getBytes("ISO-8859-1"));
        //Gera a chave criptografada em array de Bytes
para posterior hashing
        BigInteger hash= new BigInteger(1,
md.digest());
        //Retorna a senha criptografada
        return hash.toString(16);
    }
}
```

Código-fonte 58 – Classe utilitária para criptografar as senhas
Fonte: Elaborado pelo autor (2017)

Para testar o algoritmo, crie uma classe de teste com o método *main*, chamada “CriptografiaTeste”, conforme o Código-fonte “Classe de teste de criptografia”.

```
package br.com.fiap.store.teste;

import br.com.fiap.store.util.CriptografiaUtils;

public class CriptografiaTeste {

    public static void main(String[] args) {
        try {

            System.out.println(CriptografiaUtils.criptografar("12345
6"));

            System.out.println(CriptografiaUtils.criptografar("12345
6"));

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Código-fonte 59 – Classe de teste de criptografia
Fonte: Elaborado pelo autor (2017)

Depois de executar essa classe como Java Application, note que os dois *hashes* são iguais, pois as *strings* utilizadas para a criptografia também são idênticas.



Figura 41 – Resultado da execução do teste de criptografia
Fonte: Elaborado pelo autor (2017)

Em nosso exemplo, vamos focar somente nas implementações diferentes do que já realizamos. Por isso não vamos implementar o CRUD para o usuário. Contudo, para testar a funcionalidade de *login*, precisamos de um usuário cadastrado no banco de dados. Assim, execute o SQL do Código-fonte “SQL para cadastrar um usuário na base de dados” a fim de gravar um usuário (com um e-mail válido) e a senha, já criptografada.

```
INSERT INTO TB_USUARIO (DS_EMAIL, DS_SENHA) VALUES  
( 'teste@fiap.com.br', 'e10adc3949ba59abbe56e057f20f883e' );  
commit;
```

Código-fonte 60 – SQL para cadastrar um usuário na base de dados
Fonte: Elaborado pelo autor (2017)

Qual foi a senha gravada no banco? Foi o *hash* para a senha supersegura “123456”.

2.12 Java Bean

Talvez você já esteja se acostumando com os passos para trabalhar na aplicação, não existe uma regra, porém fica mais fácil trabalhar nesta ordem: banco de dados, javabean, dao, *controller* e *view*.

Portanto, vamos implementar a classe Java Bean Usuário. Essa classe deve possuir somente dois atributos: email e senha (Código-fonte “Java Bean Usuário”).

```
package br.com.fiap.store.bean;  
  
import br.com.fiap.store.util.CriptografiaUtils;  
  
public class Usuario {  
  
    private String email;
```

```
private String senha;

public Usuario(String email, String senha) {
    super();
    this.email = email;
    setSenha(senha);
}

public Usuario() {
    super();
}

public String getEmail() {
    return email;
}

public void setEmail(String email) {
    this.email = email;
}

public String getSenha() {
    return senha;
}

public void setSenha(String senha) {
    try {
        this.senha =
CriptografiaUtils.criptografar(senha);
    } catch (Exception e) {
        e.printStackTrace();
    }
}

}
```

Código-fonte 61 – Java Bean Usuário
Fonte: Elaborado pelo autor (2017)

Observe, na classe Usuário, que o método **setUsuario** utiliza o método de criptografia para armazenar o *hash* md5 da senha. Outro detalhe é que o construtor que recebe argumentos invoca o método **setUsuario** pelo mesmo motivo.

2.13 UsuarioDAO

Agora, crie a interface “UsuarioDAO”. Como não vamos implementar o CRUD para o usuário, o DAO tem um único método: **validarUsuario** (Código-fonte “Interface UsuarioDAO”).

```
package br.com.fiap.store.dao;

import br.com.fiap.store.bean.Usuario;

public interface UsuarioDAO {

    boolean validarUsuario(Usuario usuario);

}
```

Código-fonte 62 – Interface UsuarioDAO
Fonte: Elaborado pelo autor (2017)

Depois, crie a classe “OracleUsuarioDAO” e implemente a interface e o método de validar usuário (Código-fonte “Classe OracleUsuarioDAO”).

```
package br.com.fiap.store.dao.impl;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import br.com.fiap.store.bean.Usuario;
import br.com.fiap.store.dao.UsuarioDAO;
import br.com.fiap.store.singleton.ConnectionManager;

public class OracleUsuarioDAO implements UsuarioDAO{

    private Connection conexao;

    @Override
    public boolean validarUsuario(Usuario usuario) {
        PreparedStatement stmt = null;
        ResultSet rs = null;
        try {
            conexao
            ConnectionManager.getInstance().getConnection();
```

```
        stmt = conexao.prepareStatement("SELECT *  
FROM TB_USUARIO WHERE DS_EMAIL = ? AND DS_SENHA = ?");  
        stmt.setString(1, usuario.getEmail());  
        stmt.setString(2, usuario.getSenha());  
        rs = stmt.executeQuery();  
  
        if (rs.next()){  
            return true;  
        }  
  
    } catch (SQLException e) {  
        e.printStackTrace();  
    }finally {  
        try {  
            stmt.close();  
            rs.close();  
            conexao.close();  
        } catch (SQLException e) {  
            e.printStackTrace();  
        }  
    }  
    return false;  
}  
}
```

Código-fonte 63 – Classe OracleUsuarioDAO
Fonte: Elaborado pelo autor (2017)

O método de validação de usuário busca por um registro com o e-mail e o *hash* da senha informados; caso encontre algum usuário, retorna verdadeiro, caso contrário, retorna falso.

2.14 Menu com a opção de login

Modifique o menu para ter um formulário de login. Esse formulário deve possuir três componentes: e-mail, senha e um botão. A *action* do formulário será “login”, para a Servlet que vamos criar, e o método será *post* (Código-fonte “Menu com o formulário de login”).

```
<%@           taglib           prefix="c"  
uri="http://java.sun.com/jsp/jstl/core" %>
```



```

<nav class="navbar navbar-expand-lg navbar-dark bg-dark">
  <a class="navbar-brand" href="home.jsp">FIAPStore</a>
  <button class="navbar-toggler" type="button" data-
toggle="collapse" data-target="#navbarSupportedContent" aria-
controls="navbarSupportedContent" aria-expanded="false" aria-
label="Toggle navigation">
    <span class="navbar-toggler-icon"></span>
  </button>

  <div class="collapse navbar-collapse"
id="navbarSupportedContent">
    <ul class="navbar-nav mr-auto">
      <li class="nav-item">
        <a class="nav-link" href="produto?acao=abrir-
form-cadastro">Cadastro</a>
      </li>
      <li class="nav-item">
        <a class="nav-link"
href="produto?acao=listar">Produtos</a>
      </li>
    </ul>

    <span class="navbar-text text-danger"
style="margin-right:10px" >
      ${erro }
    </span>
    <form class="form-inline my-2 my-lg-0"
action="login" method="post">
      <input class="form-control mr-sm-2"
type="text" name="email" placeholder="E-mail">
      <input class="form-control mr-sm-2"
type="password" name="senha" placeholder="Senha">
      <button class="btn btn-outline-success my-2 my-
sm-0" type="submit">Entrar</button>
    </form>
  </div>
</nav>

```

Código-fonte 64 – Menu com o formulário de login
Fonte: Elaborado pelo autor (2017)

Note que adicionamos também uma área para exibir as mensagens de erro, para o caso de o usuário entrar com um usuário ou uma senha inválidos.

2.15 Serviço de envio de e-mail

Antes de criar a Servlet (*Controller*), vamos implementar um serviço de envio de e-mail, já que, quando o usuário se logar na aplicação, um e-mail será enviado.

Para o nosso sistema conseguir enviar um e-mail, precisamos da biblioteca Java Mail. Dessa forma, copie a biblioteca para o diretório “WebContent/WEB-INF/lib” (Figura “Biblioteca de envio de e-mail do Java”). A documentação e a biblioteca para download podem ser encontradas no link: <<http://www.oracle.com/technetwork/java/javamail/index.html>>.

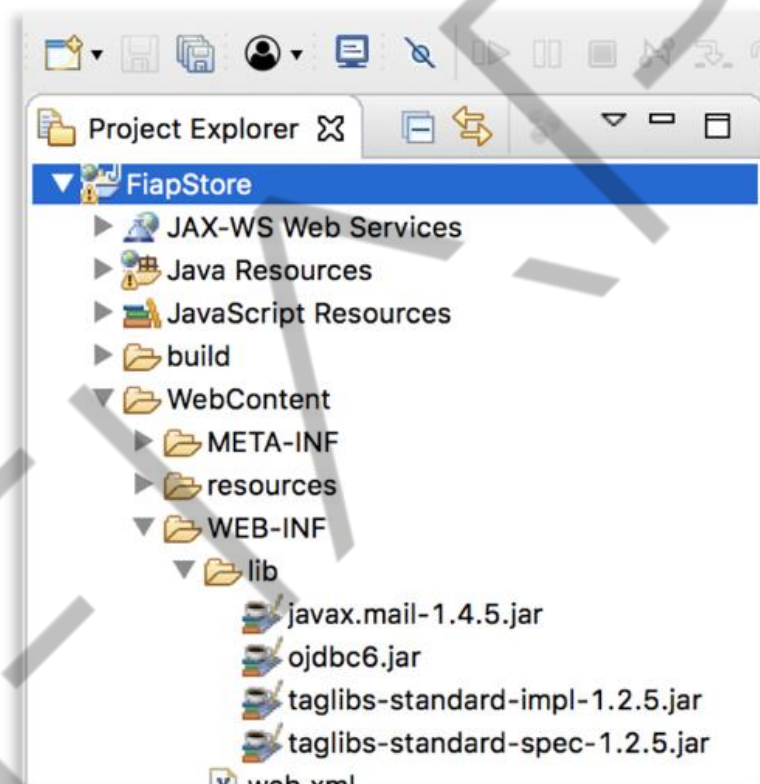


Figura 42 – Biblioteca de envio de e-mail do Java
Fonte: Elaborado pelo autor (2017)

Em nosso exemplo, vamos trabalhar com o e-mail do Gmail. Crie uma classe chamada “EmailBO”, no pacote “br.com.fiap.store.bo” (Código-fonte “Classe para envio de e-mail”). O BO são as iniciais de *Business Object*, ou seja, objeto de negócios, uma camada na qual ficam as regras de negócios da aplicação.

```
package br.com.fiap.store.bo;  
  
import java.util.Properties;
```

```
import javax.mail.Message;
import javax.mail.MessagingException;
import javax.mail.PasswordAuthentication;
import javax.mail.Session;
import javax.mail.Transport;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeMessage;

import br.com.fiap.store.exception.EmailException;

public class EmailBO {

    public void enviarEmail(String destinatario, String
assunto, String mensagem) throws EmailException{
        final String username = "email@gmail.com";
        final String password = "senha";

        Properties props = new Properties();
        props.put("mail.smtp.auth", "true");
        props.put("mail.smtp.starttls.enable",
"true");

        props.put("mail.smtp.host", "smtp.gmail.com");
        props.put("mail.smtp.port", "587");

        Session session = Session.getInstance(props,
        new javax.mail.Authenticator() {
            protected PasswordAuthentication
getPasswordAuthentication() {
                return new
PasswordAuthentication(username, password);
            }
        });

        try {

            Message email = new MimeMessage(session);
            email.setFrom(new
InternetAddress(username));

            email.setRecipients(Message.RecipientType.TO,
InternetAddress.parse(destinatario));
            email.setSubject(assunto);
            email.setText(mensagem);
```

```
        Transport.send(email);

    } catch (MessagingException e) {
        throw new EmailException("Erro ao enviar o
e-mail");
    }
}
```

Código-fonte 65 – Classe para envio de e-mail
Fonte: Elaborado pelo autor (2017)

O método de envio de e-mail recebe o destinatário, o assunto e a mensagem. O método está configurado para envio de e-mail utilizando o Gmail, as modificações necessárias são o e-mail e a senha do Gmail que será utilizado nos envios dos e-mails. Dessa forma, atualize as variáveis username e password. Caso algum erro aconteça, a exception “EmailException” será lançada. Assim, crie essa exception, conforme o Código-fonte “EmailException”.

```
package br.com.fiap.store.exception;

public class EmailException extends Exception {

    public EmailException() {
        super();
        // TODO Auto-generated constructor stub
    }

    public EmailException(String message, Throwable
cause, boolean enableSuppression, boolean writableStackTrace)
{
        super(message, cause, enableSuppression,
writableStackTrace);
        // TODO Auto-generated constructor stub
    }

    public EmailException(String message, Throwable
cause) {
        super(message, cause);
        // TODO Auto-generated constructor stub
    }

    public EmailException(String message) {
```

```
        super(message);  
        // TODO Auto-generated constructor stub  
    }  
  
    public EmailException(Throwable cause) {  
        super(cause);  
        // TODO Auto-generated constructor stub  
    }  
  
}
```

Código-fonte 66 – EmailException
Fonte: Elaborado pelo autor (2017)

2.16 LoginServlet

Com o serviço de e-mail pronto, vamos criar o *controller* para receber as informações do formulário de login e validar o usuário. Para isso, crie uma nova Servlet chamada “LoginServlet”. Implemente o método **doPost**, conforme o Código-fonte “LoginServlet”.

```
package br.com.fiap.store.controller;  
  
import java.io.IOException;  
  
import javax.servlet.ServletException;  
import javax.servlet.annotation.WebServlet;  
import javax.servlet.http.HttpServlet;  
import javax.servlet.http.HttpServletRequest;  
import javax.servlet.http.HttpServletResponse;  
import javax.servlet.http.HttpSession;  
import br.com.fiap.store.bean.Usuario;  
import br.com.fiap.store.bo.EmailBO;  
import br.com.fiap.store.dao.UsuarioDAO;  
import br.com.fiap.store.exception.EmailException;  
import br.com.fiap.store.factory.DAOFactory;  
  
@WebServlet("/login")  
public class LoginServlet extends HttpServlet {  
    private static final long serialVersionUID = 1L;  

```

```
private UsuarioDAO dao;
private EmailBO bo;

public LoginServlet() {
    dao = DAOFactory.getUsuarioDAO();
    bo = new EmailBO();
}

protected void doPost(HttpServletRequest request,
    HttpServletResponse response) throws ServletException,
    IOException {
    String email = request.getParameter("email");
    String senha = request.getParameter("senha");

    Usuario usuario = new Usuario(email, senha);

    if (dao.validarUsuario(usuario)) {
        HttpSession session =
request.getSession();
        session.setAttribute("user", email);
        String mensagem = "Um login foi realizado";
        try {
            bo.enviarEmail(email, "Login
Realizado", mensagem);
        } catch (EmailException e) {
            e.printStackTrace();
        }
    } else {
        request.setAttribute("erro", "Usuário e/ou
senha inválidos");
    }

    request.getRequestDispatcher("home.jsp").forward(request
, response);
}

}
```

Código-fonte 67 – LoginServlet
Fonte: Elaborado pelo autor (2017)

A classe “LoginServlet” precisa dos objetos “UsuarioDAO”, para validar o usuário e a senha, e “EmailBO”, para enviar o e-mail ao usuário. Por isso declaramos dois atributos, um de cada tipo, e inicializamos os objetos no método **init** da Servlet.

No método **doPost**, recuperamos os valores de email e senha do formulário de login, instanciamos um objeto do tipo `Usuario` (a senha é criptografada no momento que o objeto `Usuario` é construído), e utilizamos o `UsuarioDAO` para validar se as informações estão corretas.

Caso o usuário seja autenticado corretamente, criamos uma seção para o usuário e armazenamos o e-mail dele no atributo de nome "user". Nunca armazene a senha na sessão, já que a sessão do usuário pode ser manipulada.

Depois, um e-mail é enviado para notificar o usuário do login e é redirecionado para a página "home.jsp". Caso o usuário informe valores inválidos, uma mensagem de erro é exibida na página inicial "home.jsp". Antes de testar a funcionalidade, vamos criar a página "home.jsp" (Código-fonte "Página inicial da aplicação").

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01
Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1">
<title>Home</title>
<%@ include file="header.jsp" %>
</head>
<body>

<%@ include file="menu.jsp" %>
<div class="container">
<h1>Bem-vindos à FiapStore</h1>

<%@ include file="footer.jsp" %>
</div>
</body>
</html>
```

Código-fonte 68 – Página inicial da aplicação
Fonte: Elaborado pelo autor (2017)

A página inicial é bem simples, só possui um título de boas-vindas. Mas como vamos identificar se o usuário foi autenticado corretamente? Ajuste o menu (Código-

fonte “Ajuste na barra de navegação para exibir o usuário ou o formulário de login”), para que exiba o e-mail do usuário (atributo user da sessão) quando ele estiver logado.

```
<%@ taglib prefix="c"
uri="http://java.sun.com/jsp/jstl/core" %>
<nav class="navbar navbar-expand-lg navbar-dark bg-
dark">
    <a class="navbar-brand" href="home.jsp">FIAPStore</a>
    <button class="navbar-toggler" type="button" data-
toggle="collapse" data-target="#navbarSupportedContent" aria-
controls="navbarSupportedContent" aria-expanded="false" aria-
label="Toggle navigation">
        <span class="navbar-toggler-icon"></span>
    </button>

    <div class="collapse navbar-collapse"
id="navbarSupportedContent">
        <ul class="navbar-nav mr-auto">
            <li class="nav-item">
                <a class="nav-link" href="produto?acao=abrir-
form-cadastro">Cadastro</a>
            </li>
            <li class="nav-item">
                <a class="nav-link"
href="produto?acao=listar">Produtos</a>
            </li>
        </ul>
        <c:if test="${empty user }">
            <span class="navbar-text text-danger"
style="margin-right:10px" >
                ${erro }
            </span>
            <form class="form-inline my-2 my-lg-0"
action="login" method="post">
                <input class="form-control mr-sm-2"
type="text" name="email" placeholder="E-mail">
                <input class="form-control mr-sm-2"
type="password" name="senha" placeholder="Senha">
                <button class="btn btn-outline-success my-2 my-
sm-0" type="submit">Entrar</button>
            </form>
        </c:if>
        <c:if test="${not empty user }">
```



```

<span class="navbar-text">
  ${user }
  <a href="/logout" class="btn btn-outline-
primary my-2 my-sm-0">Sair</a>
</span>
</c:if>
</div>
</nav>

```

Código-fonte 69 – Ajuste na barra de navegação para exibir o usuário ou o formulário de login
 Fonte: Elaborado pelo autor (2017)

Observe que utilizamos a tag `<c:if>` para validar se existe o atributo “user”, caso exista, exibimos o e-mail do usuário e um link para efetuar o *logout*. Caso não exista, exibimos a mensagem de erro e o formulário de login. Agora podemos testar! Primeiro, teste um usuário e uma senha inválidos (Figura “Teste de login inválido”).

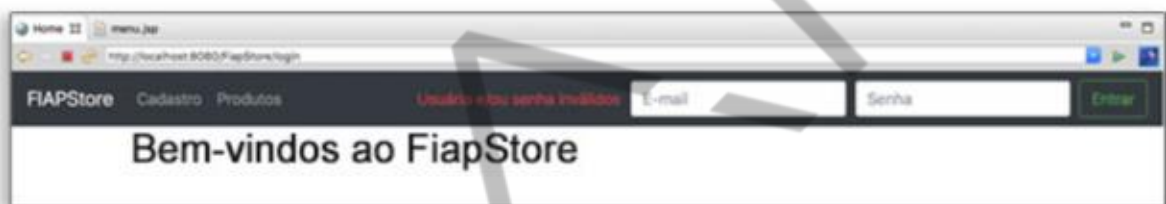


Figura 43 – Teste de login inválido
 Fonte: Elaborado pelo autor (2017)

Agora, vamos testar com um usuário válido (Figura “Teste de login válido”).

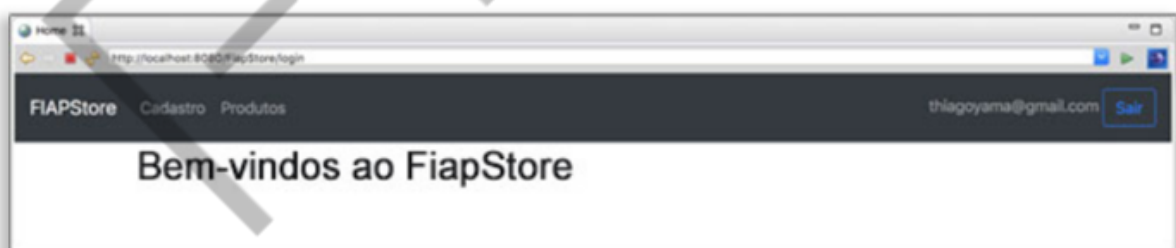


Figura 44 – Teste de login válido
 Fonte: Elaborado pelo autor (2017)

Sucesso! O e-mail do usuário foi gravado na sessão e é exibido na barra de navegação, próximo ao link de *logout*. Valide também se o e-mail foi enviado corretamente.

Para finalizar, o usuário também deve ser capaz de deslogar da aplicação. Implemente o método **doGetb** da classe `LoginServlet` (Código-fonte “Funcionalidade

de logout”), para invalidar a sessão do usuário, após este clicar no link “Sair” da barra de navegação.

```
protected void doGet(HttpServletRequest request,
    HttpServletResponse response) throws ServletException,
    IOException {
    HttpSession session = request.getSession();
    session.invalidate();

    request.getRequestDispatcher("home.jsp").forward(request
    , response);
}
```

Código-fonte 70 – Funcionalidade de logout
Fonte: Elaborado pelo autor (2017)

O método recupera a sessão do usuário e utiliza o método **invalidate()** para excluir a sessão do usuário, depois encaminha para a página “home.jsp”. Vamos testar! Clique no link “Sair”, o formulário de login deve ser apresentado novamente (Figura “Teste da funcionalidade de logout”).

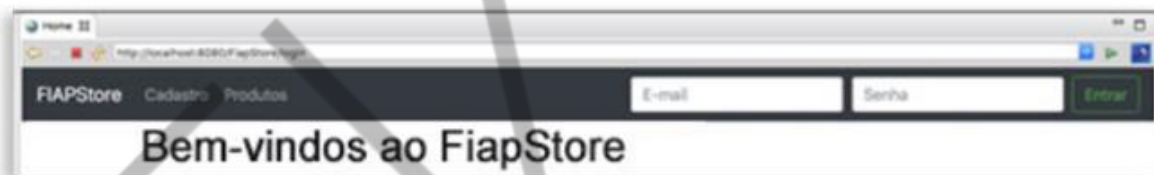


Figura 45 – Teste da funcionalidade de logout
Fonte: Elaborado pelo autor (2017)

2.17 Filtro

Com o login funcionando, podemos agora criar uma forma de permitir somente aos usuários logados acessarem as outras funcionalidades do sistema. Por exemplo, antes de listar ou cadastrar um produto, o usuário deve estar autenticado no sistema.

Para implementar essa restrição, precisaremos dos filtros. Os filtros fazem exatamente o que o nome diz, filtram as requisições antes de encaminhar os usuários ao seu destino. Crie um novo Filtro, clique com o botão direito do mouse na pasta “src” e escolha New > Filter, conforme a Figura “Criando um Filtro – parte 1”.

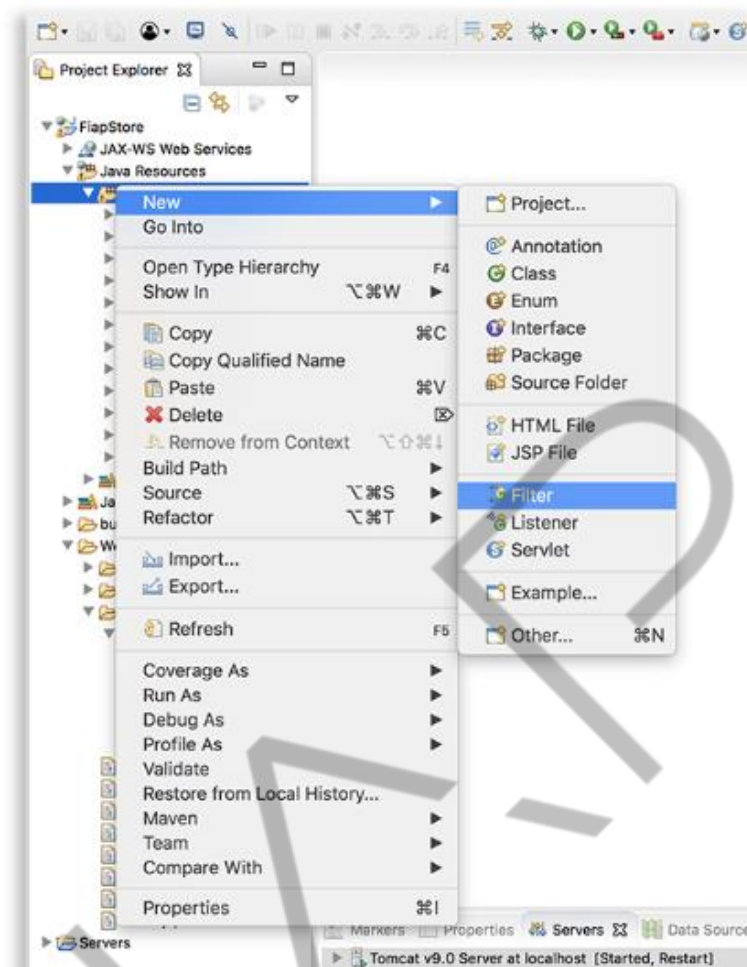


Figura 46 – Criando um Filtro – parte 1
Fonte: Elaborado pelo autor (2017)

Dê o nome “LoginFilter”, configure o pacote para “br.com.fiap.store.filter” (Figura “Criando um Filtro – parte 2”) e finalize o processo.

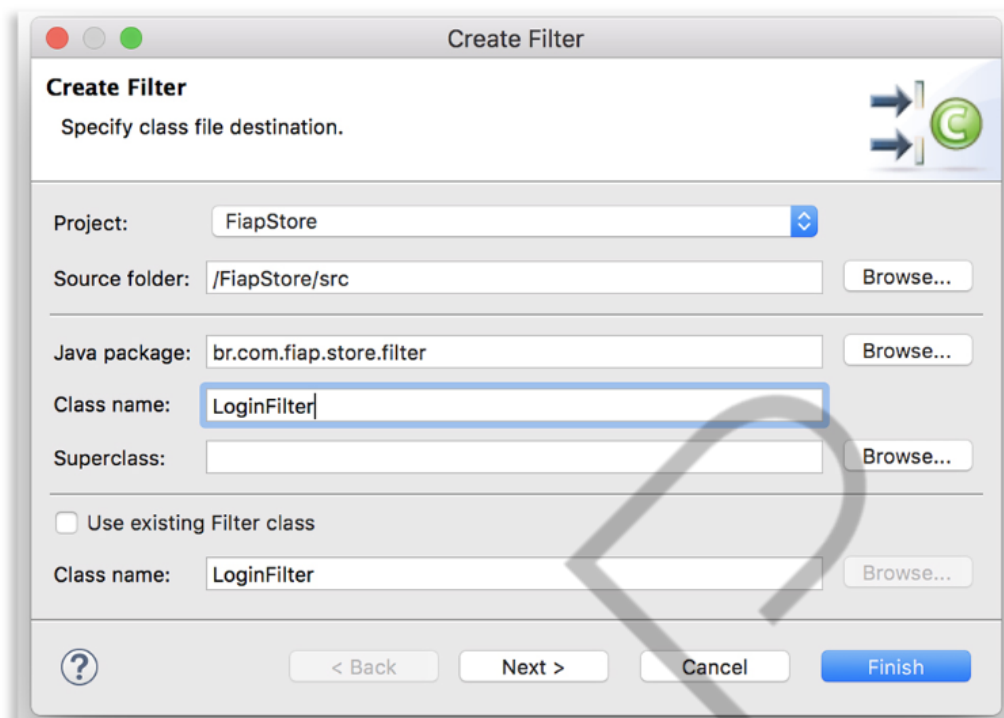


Figura 47 – Criando um Filtro – parte 2
Fonte: Elaborado pelo autor (2017)

Esse processo cria automaticamente alguns métodos do ciclo de vida do Filter, como os métodos **init** e **destroy**. O único método de que vamos precisar é o **doFilter**, responsável por filtrar as requisições enviadas ao servidor.

A anotação **@WebFilter** determina a URL em que o filtro vai atuar. Como queremos filtrar todas as requisições, vamos configurar a URL como “/*”, o asterisco (*) é o caractere coringa que pode assumir qualquer valor. Dessa forma, qualquer URL será interceptada pelo filtro. Analise a implementação do filtro, no Código-fonte “Implementação do Filtro”.

```
package br.com.fiap.store.filter;

import java.io.IOException;

import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.annotation.WebFilter;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;
```

```
@WebFilter("/*")
public class LoginFilter implements Filter{

    @Override
    public void doFilter(ServletRequest request,
        ServletResponse response, FilterChain chain)
        throws IOException, ServletException {

        HttpServletRequest req = (HttpServletRequest)
request;

        HttpSession session = req.getSession();
        String url = req.getRequestURI();

        if (session.getAttribute("user") == null &&
!url.endsWith("login") && !url.contains("resources") &&
!url.contains("home")) {
            request.setAttribute("erro", "Entre com o
usuário e senha!");

            request.getRequestDispatcher("home.jsp").forward(request
, response);
        }else {
            chain.doFilter(request, response);
        }
    }
}
```

Código-fonte 71 – Implementação do Filtro

Fonte: Elaborado pelo autor (2017)

A primeira instrução da implementação do método **doFilter** é um cast, para transformar um **ServletRequest** em um **HttpServletRequest**. Depois, obtemos a sessão do usuário e a URL que o usuário está tentando acessar.

Na validação, verificamos se o usuário está logado, por meio do atributo de sessão “user”, que é adicionado na Servlet “LoginServlet”, quando o usuário é autenticado com sucesso. Validamos também se a URL não possui parte do conteúdo “resources”, “login” ou “home”, já que o usuário não precisa estar logado para acessar os arquivos de css, js, imagens etc., que estão no diretório “resources”; ou a funcionalidade de “login” ou a página inicial do sistema, “home.jsp”.

Caso alguma condição do if seja falsa, o usuário é redirecionado para a página inicial com uma mensagem de erro. Caso contrário, o método **doFilter** do objeto FilterChain encaminha a requisição para o seu destino. Agora podemos testar a aplicação. Tente acessar a página de cadastrar ou listar produtos sem estar autenticado (Figura “Teste do filtro de usuário – parte 1”).

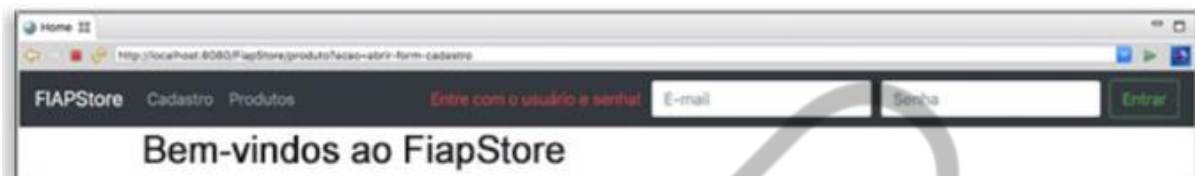


Figura 48 – Teste do filtro de usuário – parte 1
Fonte: Elaborado pelo autor (2017)

Agora autentique-se e tente acessar a página novamente (Figura “Teste do filtro de usuário – parte 2”).

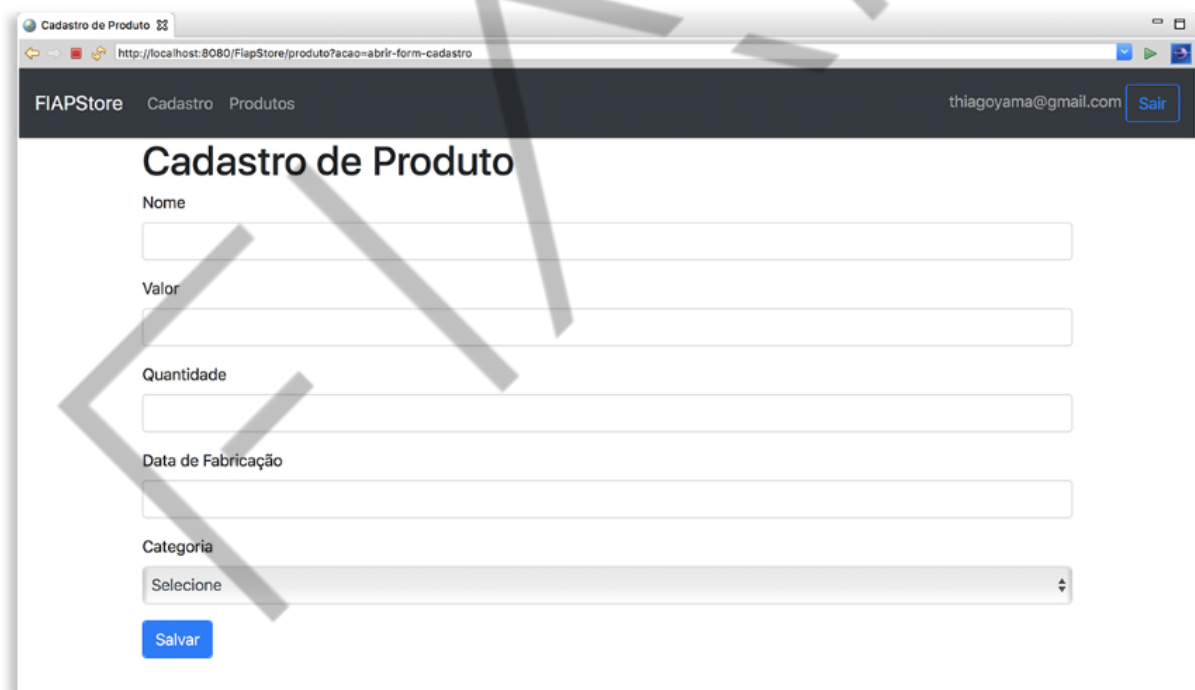


Figura 49 – Teste do filtro de usuário – parte 2
Fonte: Elaborado pelo autor (2017)

Perfeito! Note que mesmo navegando entre as telas, você pode ver o e-mail do usuário na barra de navegação, já que essa informação está na sessão do usuário e não na requisição, ou seja, estará presente até a sessão ser invalidada.

2.18 Configurações da aplicação

Sempre que criamos um projeto Java Web, configuramos para criar o arquivo “web.xml”, que fica no diretório “WebContent/WEB-INF”. Porém, até o momento, não o utilizamos.

Esse arquivo permite configurar a aplicação web, como as Servlets e os Filtros da aplicação, em vez de utilizar as anotações `@WebServlet` e `@WebFilter`, optamos pelas anotações, pois é muito mais fácil e produtivo.

Mas existem outras configurações possíveis e que vamos utilizar agora.

A primeira é a página inicial da aplicação. No final da URL, sempre existe o nome do arquivo JSP ou da URL mapeada para uma Servlet, mas e se não houver nada? Ou seja, e se a URL terminar no nome da aplicação?

Podemos configurar a página inicial por meio das tags `<welcome-file-list>` e `<welcome-file>`. A configuração recebe uma lista de nomes de páginas que têm prioridades, ou seja, a primeira página encontrada na lista é exibida para o usuário. Ajuste a configuração para que a página “home.jsp” seja a página inicial da aplicação (Código-fonte “Página inicial da aplicação”).

```
<welcome-file-list>
  <welcome-file>home.jsp</welcome-file>
</welcome-file-list>
```

Código-fonte 72 – Página inicial da aplicação
Fonte: Elaborado pelo autor (2017)

Vamos testar, selecione o projeto e execute-o no servidor (Figura “Teste da página inicial do sistema”).

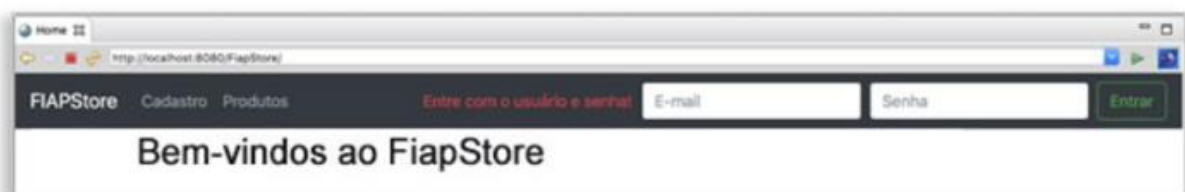


Figura 50 – Teste da página inicial do sistema
Fonte: Elaborado pelo autor (2017)

A próxima configuração é o timeout de sessão. Podemos configurar o tempo de expiração da sessão do usuário, ou seja, o tempo que o usuário pode ficar sem acessar a nossa aplicação. Essa configuração é determinada em minutos.

O Código-fonte “Configuração de timeout da aplicação” configura a aplicação para um timeout de 5 minutos.

```
<session-config>
  <session-timeout>5</session-timeout>
</session-config>
```

Código-fonte 73 – Configuração de timeout da aplicação
Fonte: Elaborado pelo autor (2017)

Com isso, o usuário perde a sessão após 5 minutos de inatividade, ou se ele utilizar o link “Sair” da aplicação. A última configuração que vamos realizar são as páginas de erro. Provavelmente você já acessou um site ou digitou uma URL e um erro 404 aconteceu. Se você tentar acessar uma página inexistente em nossa aplicação, esse erro vai acontecer (Figura “Erro 404, recurso não encontrado”).

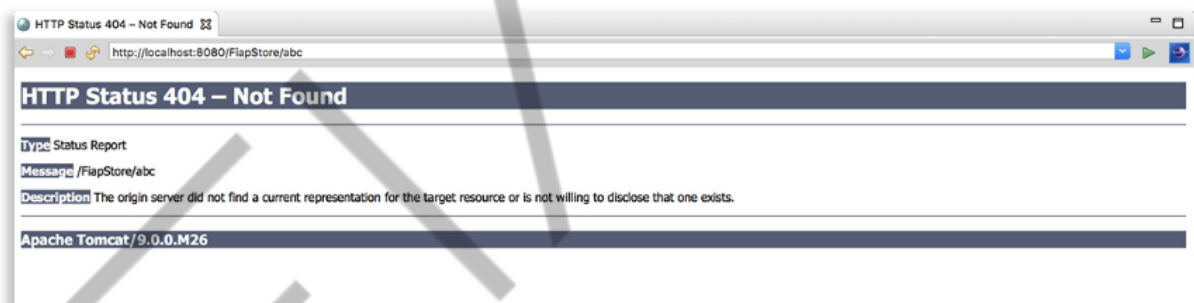


Figura 51 – Erro 404, recurso não encontrado
Fonte: Elaborado pelo autor (2017)

O detalhe é que você precisa estar logado, pois, caso contrário, se tentar acessar qualquer URL, você será redirecionado para a página inicial da aplicação.

Para exibir uma página amigável ao usuário, podemos configurar uma página de erro. Vamos configurar uma página para o erro 404, porém, podemos configurar várias páginas para diferentes erros, como HTTP Status 500, NullPointerException etc. No diretório “WebContent”, crie uma página de erro chamada “404.jsp” e configure o “web.xml”, conforme o Código-fonte “Configuração de página de erro”.

```
<error-page>
  <error-code>404</error-code>
```



```
<location>/404.jsp</location>
</error-page>
```

Código-fonte 74 – Configuração de página de erro
Fonte: Elaborado pelo autor (2017)

O próximo passo é testar. Logue-se e tente acessar uma página inválida (Figura “Teste da página de erro 404”).

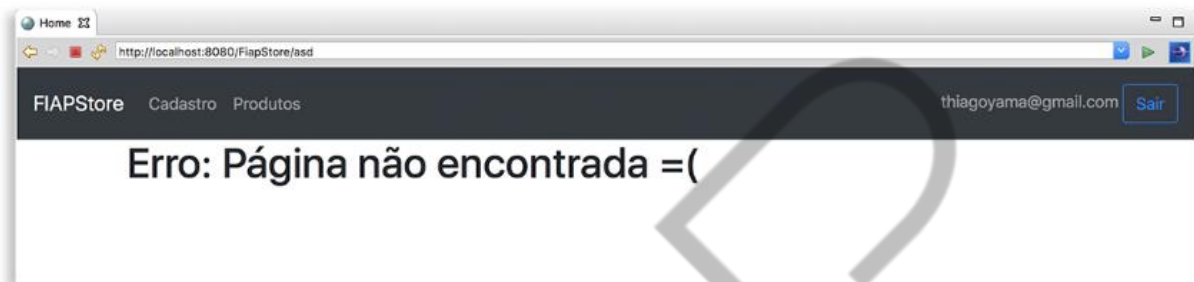


Figura 52 – Teste da página de erro 404
Fonte: Elaborado pelo autor (2017)

Muito melhor! Para mais uma página de erro, basta adicionar outra tag <error-page>. No Código-fonte “Configuração completa do web.xml”, apresentamos a configuração completa do “web.xml”, com a página de erro para a exceção NullPointerException.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/jav
aee http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
id="WebApp_ID" version="3.1">
<display-name>FiapStore</display-name>
<welcome-file-list>
<welcome-file>home.jsp</welcome-file>
</welcome-file-list>
<session-config>
<session-timeout>5</session-timeout>
</session-config>
<error-page>
<error-code>404</error-code>
<location>/404.jsp</location>
</error-page>
</error-page>
```

```
<exception-  
type>NullPointerException</exception-type>  
    <location>/erro.jsp</location>  
</error-page>  
</web-app>
```

Código-fonte 75 – Configuração completa do web.xml
Fonte: Elaborado pelo autor (2017)

Com isso, finalizamos a nossa aplicação, com todas as operações envolvendo o banco de dados (CRUD), relacionamentos, login, criptografia, filtros, envio de e-mail e configurações da aplicação web. É claro que sempre podemos melhorar o código. Fica o incentivo para refatorar o código, retirando as implementações duplicadas e adicionando mais funcionalidades, como cadastro de categoria e usuário. Lembre-se de que é praticando que se aprende!

REFERÊNCIAS

BARNES, D. J. **Programação Orientada a Objetos com Java**: uma Introdução Prática Utilizando Blue J. São Paulo: Pearson, 2004.

BASHAM, B.; SIERRA, H.; BATES, B. **Use a Cabeça! Servlets & JSP**. Rio de Janeiro: Alta Books, 2005.

GEARY, D. M. **Dominando Javasever Pages Avançado**. Rio de Janeiro: Ciência Moderna, 2002.

HORSTMANN, C.; CORNELL, G. **Core Java**. 8. ed. São Paulo: Pearson, 2009. (Volume 1 – Fundamentos)

SAMPAIO, C. **Java Enterprise**. 6. ed. Rio de Janeiro: Brasport, 2011.

WATRALL, E.; SIARTO, J. **Use a Cabeça! Web Design**. Rio de Janeiro: Alta Books, 2012.