04 Cadastro de produtos

PRÓXIMA ATIVIDADE

Nossa aplicação está começando a tomar forma. Ela já lista os produtos em nossa **home**. Agora começaremos a criar o cadastro de produtos, para assim podermos cadastrar livros da Casa do Código.

Para o cadastro de produtos, precisaremos de um formulário. Sendo assim, crie um novo arquivo **JSP** chamado **form.jsp** dentro da pasta `WEB-INF/views/produtos/`. A pasta `produtos` ainda não existe, teremos que criá-la também. faremos alterações para deixar no padrão HTML 5. O arquivo `form.jsp` inicial deve estar parecido com esse:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Livros de Java, Android, iPhone, Ruby, PHP e muito mais - Casa d
</head>
<body>

</body>
</html>
```

O próximo passo é criar o **html** referente ao formulário (`form`) de cadastro dos livros. Eles terão inicialmente os seguintes atributos: **Título**, **Descrição** e **Número de Páginas**, todos do tipo texto. Vamos então criar o formulário com estes campos. O formulário deve ficar parecido com este:

```
<form action="/produtos" method="post">
    <div>
        <label>Título</label>
        <input type="text" name="titulo" />
    </div>
    <div>
        <label>Descrição</label>
        <textarea rows="10" cols="20" name="descricao"></textarea>
    </div>
    <div>
        <label>Páginas</label>
        <input type="text" name="paginas" />
    </div>
    <button type="submit">Cadastrar</button>
</form>
```

Note que estamos fazendo o formulário enviar seus dados para o **path** `/produtos` e que o estamos enviando via `post` no método do formulário.

Nosso `form.jsp` deve ser acessado na url `localhost:8080/casadocodigo/produtos/form`. Se acessarmos agora, veremos uma página de `erro 404`, pois as *views* não podem ser acessadas diretamente. Lembra? Como resolvemos isso? Criando um `Controller`!

Crie então o `ProdutosController` dentro do pacote `br.com.casadocodigo.loja.controllers`. Neste `Controller` crie o método `form` que retorna a *view* com o formulário. Mapeie o **path** que este método vai atender com a anotação `@RequestMapping`. O `ProdutosController` deve ficar parecido com este:

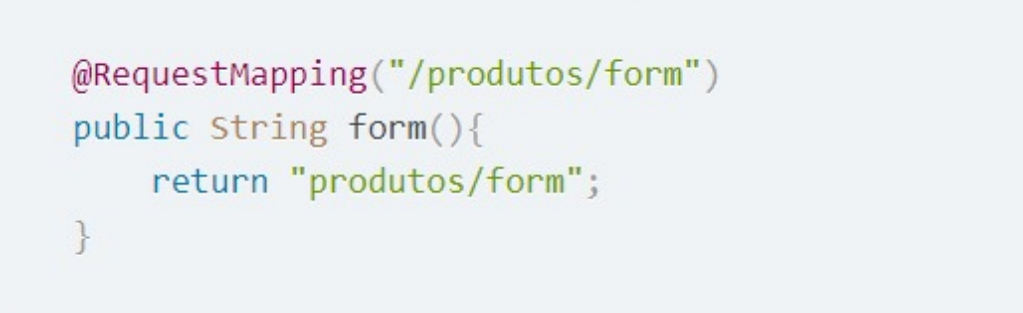
```
@Controller
public class ProdutosController {

    @RequestMapping("/produtos/form")
    public String form(){
        return "produtos/form";
    }

}
```

Se acessarmos agora o nosso formulário em `localhost:8080/casadocodigo/produtos/form`, devemos vê-lo sem nenhum problema. Não esqueça de reiniciar o servidor! Teste o formulário, tente cadastrar um livro!

Quando enviamos o formulário, recebemos um `erro 404`:



Apache Tomcat/7.0.63 - E

localhost:8080/produtos

HTTP Status 404 - /produtos

type Status report

message /produtos

description The requested resource is not available.

Apache Tomcat/7.0.63

Mas tem algo estranho neste erro 404 , note a **url**. O caminho está sem o `/casadocodigo/`. Isto acontece porque o nosso formulário não aponta `/casadocodigo/` em sua action. Vamos fazê-lo apontar, então.

```
<form action="/casadocodigo/produtos" method="post">
    [...]
</form>
```

Este é o primeiro passo para resolver nosso problema. Agora precisamos mapear o `path` `/produtos` para um método em no `ProdutosController`. Algo parecido com:

```
@RequestMapping("/produtos")
public String gravar(String titulo, String descricao, int paginas){
    System.out.println(titulo);
    System.out.println(descricao);
    System.out.println(paginas);

    return "ok";
}
```

Nosso método atende ao *path* `/produtos` e se chama `gravar`. Este irá gravar os produtos em um banco de dados posteriormente. Ele recebe os dados do produto e imprime no terminal, por fim, retorna "ok" para indicar que tudo ocorreu bem.

Faça alguns testes! Teremos um erro 404 indicando que a *view* `ok` não foi encontrada, mas não deixe de verificar se os dados do formulário foram impressos no console. Funciona! O SpringMVC sozinho verifica a assinatura do nosso método e faz um *bind* dos parâmetros do método com os *names* do formulário.

Nossa aplicação já funciona, mas antes de continuarmos, vamos melhorar um ponto e corrigir outro.

Primeiro ponto, imagine que o formulário de produtos terá 30 campos. A assinatura do nosso método ficará enorme! Vamos mudar isso, o método `gravar` requer um **produto**. Vamos criar um Produto então.

Crie uma classe chamada `Produto` com os mesmos atributos do formulário e os defina como `private`. Use os atalhos do Eclipse e gere também os *getters and setters*. Gere também o `toString` na classe `Produto` para que deixemos de imprimir aquela mensagem padrão estranha e possamos imprimir o objeto diretamente de forma amigável.A classe `Produto` deve estar no pacote `br.com.casadocodigo.loja.models` .Esta classe representa uma entidade no nosso sistema.

```
package br.com.casadocodigo.loja.models;

public class Produto {
    private String titulo;
    private String descricao;
    private int paginas;


    public String getTitulo() {
        return titulo;
    }
    public void setTitulo(String titulo) {
        this.titulo = titulo;
    }
    public String getDescricao() {
        return descricao;
    }
    public void setDescricao(String descricao) {
        this.descricao = descricao;
    }
    public int getPaginas() {
        return paginas;
    }
    public void setPaginas(int paginas) {
        this.paginas = paginas;
    }

    @Override
    public String toString() {
        return "Produto [titulo=" + titulo + ", descricao=" + descricao
    }

}
```

Desta forma isolamos todo o comportamento e dados dos produtos em uma classe. Podemos então em nosso **Controller** receber um **Produto** em vez de seus dados separadamente.O SpringMVC fará o `bind` dos *names* em nosso formulário com os atributos do **Produto** de agora em diante. Sendo assim, vamos modificar o `ProdutosController` para recebermos um objeto `produto` agora.

O método `gravar` deve ficar assim:

```
@RequestMapping("/produtos")
public String gravar(Produto produto){
    System.out.println(produto);
    return "/produtos/ok";
}
```

Agora recebemos um objeto do tipo produto, imprimimos o produto no console e retornamos a *view* "ok", que deve estar dentro da pasta produtos (perceba que mudamos também o caminho de retorno da view de `/ok` em `/views/` para `/produtos/ok` em `/views/produtos/ok`). Crie a *view* `ok.jsp` na pasta `WEB-INF/views/produtos` com uma mensagem de sucesso. Algo como:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html>
        <html>
<head>
<meta charset="UTF-8">
<title>Livros de Java, Android, iPhone, Ruby, PHP e muito mais - Casa d
</head>
<body>
 <h1>Produto cadastrado com sucesso!</h1>
</body>
</html>
```

Agora, se acessarmos nosso formulário, preenchermos os campos e tentarmos cadastrar um produto (um livro), teremos a mensagem de sucesso e os dados do nosso livro serão mostrados no console. Faça o teste!

localhost:8080/casadocodigo/produtos

Produto cadastrado com sucesso!

Salvando produtos no banco de dados

Nossa aplicação ainda não salva os produtos no banco de dados. Para essa tarefa usaremos a **JPA** (*Java Persistence API*) e o **Hibernate**, que ainda não estão configurados em nosso projeto. Vamos configura-los agora.

No `pom.xml` vamos declarar algumas novas dependências. Entre elas estão a **JPA**, o **Hibernate**, O **SpringORM** e o **Driver MySQL**. No final do `pom.xml` antes do fechamento da tag `<dependencies>` cole as seguintes dependencias:

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-entitymanager</artifactId>
    <version>4.3.0.Final</version>
</dependency>
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>4.3.0.Final</version>
</dependency>
<dependency>
    <groupId>org.hibernate.javax.persistence</groupId>
    <artifactId>hibernate-jpa-2.1-api</artifactId>
    <version>1.0.0.Final</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-orm</artifactId>
    <version>4.1.0.RELEASE</version>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.15</version>
</dependency>
```

Salve o `pom.xml` e aguarde um momento, pois o Maven irá baixar e deixar disponivel em nosso projeto as bibliotecas que acabamos de adicionar como dependências.

Com as dependências configuradas, podemos começar a criar a lógica responsável por salvar os produtos. O primeiro passo é definir que o produto é uma entidade. Fazemos isso marcando a classe `Produto` com a anotação `@Entity`.

```
import javax.persistence.Entity;

@Entity
public class Produto {
    [...]
}
```

Atenção: O importe deve ser do pacote: `javax.persistence.Entity`.

Com este passo a classe `Produto` já representa uma entidade em nosso sistema. Fazer isso é apenas o primeiro passo, ainda não temos a lógica responsável por efetivamente salvar o produto no banco de dados.

Vamos então criar uma classe de acesso a dados responsável por manipular os dados dos produtos. Criaremos então a classe ProdutoDAO (DAO: Data Access Object ou Objeto de Acesso a Dados). Inicialmente, criamos nesta classe o método `gravar` que receberá um objeto `produto` e o salvará no banco de dados. Está classe deve ficar no pacote: `br.com.casadocodigo.loja.daos`

```
package br.com.casadocodigo.loja.daos;
import br.com.casadocodigo.loja.models.Produto;

public class ProdutoDAO {

    public void gravar(Produto produto){

    }
}
```

Para que o `ProdutoDAO` realize a **persistencia** ou seja, para que ele salve o **produto** no banco de dados. É necessário que ele tenha um gerenciador de entidades, um `EntityManager`. Este `EntityManager` é fornecido pelo `Spring`. Assim podemos usar o `EntityManager` para persistir os produto no banco de dados. No código teremos algo como:

```
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

import br.com.casadocodigo.loja.models.Produto;

public class ProdutoDAO {

    @PersistenceContext
    private EntityManager manager;

    public void gravar(Produto produto){
        manager.persist(produto);
    }

}
```

Temos quase tudo pronto neste ponto. Precisamos fazer com que agora, quando o **Controller** receba o `produto`, ele use o `ProdutoDAO` para salvar o produto no banco de dados. Estas modificações serão feitas no `ProdutosController`.

```
@Controller
public class ProdutosController {

    @Autowired
    private ProdutoDAO produtoDao;

    @RequestMapping("/produtos")
    public String gravar(Produto produto){
        System.out.println(produto);
        produtoDao.gravar(produto);
        return "/produtos/ok";
    }
    [...]
}
```

A anotação `@Autowired` serve para que nós não nos preocupemos em criar manualmente o `ProdutoDAO` no **Controller**. O **Spring** fará isso automaticamente. Mas para isso, o **Spring** precisa "conhecer" o `ProdutoDAO`. Em outras palavras dizemos que devemos definir que o `ProdutoDAO` será gerenciado pelo **Spring**. Para isso devemos marcar o `ProdutoDAO` com a anotação `@Repository`.

```
@Repository
public class ProdutoDAO {

    @PersistenceContext
    private EntityManager manager;

    public void gravar(Produto produto){
        manager.persist(produto);
    }

}
```

Se tentarmos inicializar o projeto neste momento, teremos dois problemas. O primeiro deles será que, apesar de termos aparentemente configurado todo o necessário para persistir os produtos no banco de dados, **Spring** não conseguirá gerenciar nossas classes, nem mesmo encontrá-las.

Esta configuração está presente em nossa classe `AppWebConfiguration`, na qual configuramos para o **Spring** encontrar nossos *controllers*. Nós vamos configurar para que encontre nossos **daos** também. A anotação `@ComponentScan` deve ficar assim:

```
ComponentScan(basePackageClasses={HomeController.class, ProdutoDAO.clas
```

Note que em momento algum estamos fornecendo para o **Spring** qual é o banco, o usuário ou a senha do banco de dados. Faremos essas configurações em uma nova classe. Crie uma classe no pacote `br.com.casadocodigo.conf` chamada `JPAConfiguration`.

Nesta nova classe, criaremos o método que será gerenciado pelo **Spring** e criará o `EntityManager` usado em nosso **DAO**. Ela também terá as configurações de banco de dados e algumas outras propriedades importantes. Vejamos o código:

```
package br.com.casadocodigo.loja.conf;

import java.util.Properties;

import org.springframework.context.annotation.Bean;
import org.springframework.jdbc.datasource.DriverManagerDataSource;
import org.springframework.orm.jpa.JpaVendorAdapter;
import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBe
import org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter;

public class JPAConfiguration {

    @Bean
    public LocalContainerEntityManagerFactoryBean entityManagerFactory(
        LocalContainerEntityManagerFactoryBean factoryBean = new LocalC
        JpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter(

        factoryBean.setJpaVendorAdapter(vendorAdapter);

        DriverManagerDataSource dataSource = new DriverManagerDataSourc
        dataSource.setUsername("root");
        dataSource.setPassword("");
        dataSource.setUrl("jdbc:mysql://localhost:3306/casadocodigo");
        dataSource.setDriverClassName("com.mysql.jdbc.Driver");

        factoryBean.setDataSource(dataSource);

        Properties props = new Properties();
        props.setProperty("hibernate.dialect", "org.hibernate.dialect.M
        props.setProperty("hibernate.show_sql", "true");
        props.setProperty("hibernate.hbm2ddl.auto", "update");

        factoryBean.setJpaProperties(props);

        factoryBean.setPackagesToScan("br.com.casadocodigo.loja.models"

        return factoryBean;
    }
}
```

Nesta classe estamos criando um único médoto, que será usado pelo **Spring** para gerar o **EntityManager**. Este precisa de um **adapter** e estamos passando um que o **Hibernate** disponibiliza.

Criamos também um **DataSource** que contém as configurações de banco de dados. Criamos um objeto do tipo **Properties** para podermos setar algumas configurações, como por exemplo o dialeto usado para a comunicação com o banco de dados. Setamos também onde o **EntityManager** encontrará nossos **Models**. Feito isso retornamos nossas configurações para o **Spring** poder utiliza-las.

Nosso próximo passo é disponibilizar essa configuração para o **Spring**. Faremos isso na nossa class: `ServletSpringMVC`. No Método: `getServletConfigClasses`.

```
@Override
protected Class<?>[] getServletConfigClasses() {
    return new Class[] {AppWebConfiguration.class, JPAConfiguration.cla
}
```

Mais um passo é necessário para podermos finalizar esta etapa de configuração. Como as nossas entidades serão gerenciadas pelo **Framework**, precisaremos setar mais um atributos, que essencialmente é utilizado sempre que usamos o banco de dados. O **id**. Então em nossa classe `Produto`, definiremos o `id`.

```
@Entity
public class Produto {

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

    [...]
}
```

Basicamente, só precisaríamos do `@Id`. Mas para que não precisemos gerenciá-lo manualmente, usamos a segunda anotação (`@GeneratedValue(strategy=GenerationType.IDENTITY)`) para que o próprio *framework* o gere e o gerencie. Feito isso podemos acessar a página de cadastro de produtos e tentarmos cadastrar um produto. A resposta que temos logo após tentar cadastrar um produto será essa:

```
HTTP Status 500 - Request processing failed; nested exception is javax.
```

Como assim? Aparentemente estava tudo funcionando. Note que o erro é bem claro: `No transactional EntityManager available` do tipo `TransactionRequiredException`. Ou seja, nossa operação com o banco de dados deve ser gerenciada com uma transação.

Façamos então essas últimas configurações para conseguirmos cadastrar nosso produtos no banco de dados. Primeiro precisaremos de um `TransactionManager` que conheça nosso `EntityManager` para que assim ele possa gerenciar as transações de nossas entidades.Na classe `JPAConfiguration` adicionaremos mais um método que criará o `TransactionManager`.

```
@EnableTransactionManagement
public class JPAConfiguration {

    [...]

    @Bean
    public JpaTransactionManager transactionManager(EntityManagerFactor
        return new JpaTransactionManager(emf);
    }
}
```

Note que adicionamos a anotação `@EnableTransactionManagement`. Assim o **Spring** ativa o gerenciamento de transações e já reconhece o `TransactionManager`. Agora precisamos definir que o nosso `ProdutoDAO` é uma classe *Transancional* e fazemos isso atravez da anotação `@Transactional` do pacote org.springframework.transaction.annotation.Transactional.

```
@Repository
@Transactional
public class ProdutoDAO {
    [...]
}
```

É isso! Não precisamos configurar mais nada por hora. Desta forma já conseguiremos cadastrar nossos produtos sem nenhum problema. Não se esqueça de verificar as configurações do seu banco de dados e de criar o banco `casadocodigo` pois o **Hibernate** não cria o banco, mas as tabelas e campos sim.

Experimente cadastrar alguns livros da Casa do Código e verificar se estão realmente no banco de dados. Caso tenha problemas ou dúvidas, publique no forúm, você sempre encontrará alguém para ajudar!

TIRAR DÚVIDA
PRÓXIMA ATIVIDADE