06

Mãos na massa: Cadastrando produtos

Chegou a hora de você pôr em prática o que foi visto na aula. Para isso, execute os passos listados abaixo.

Criando o ProdutosController

1) Crie um *controller* específico para os produtos dentro do seu sistema, chamado **`ProdutosController`**, dentro do pacote `br.com.caelum.loja.controllers`. Nessa classe, crie o método `form()`, que atende à URL **`produtos/form`**. Esse método deve retornar **`produtos/form`**, que é justamente o local onde você irá criar o formulário de cadastro de produtos:

```
@Controller
public class ProdutosController {

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

2) Agora, crie a view, ou seja, o formulário de cadastro de produtos. Crie a pasta **produtos** dentro de **src/main/webapp/WEB-INF/views/** e dentro dela, crie a página **form.jsp** com o seguinte conteúdo:

```
<%@ 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, PHP, Ruby e muito mais - Casa do Código</title>
</head>
<body>
    <form action="/casadocodigo/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>
</body>
</html>
```

3) Crie o método `gravar()` na classe ProdutosController.java, que atenderá à URL `casadocodigo/produtos` que é justamente o endereço que o formulário está enviando os dados. Esse método será o responsável por receber os dados do formulário, então receba-os por parâmetro e imprima-os dentro do método. O nome dos atributos devem ser exatamente os mesmos valores contidos nos atributos `name` dos *inputs* do formulário de cadastro, pois o *binding* do Spring vincula cada valor de acordo com o seu **nome**. Por fim, esse método deve retornar `produtos/ok`, que será a página para onde o usuário será enviado:

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

4) Ao invés de receber diversos parâmetros no método, o ideal é receber uma classe que representa todos esses eles. Então, crie a classe `Produto` no pacote `br.com.casadocodigo.loja.models` com os atributos `titulo`, `descricao` e `paginas`, além dos seus *getters* e *setters* e de uma implementação do método `toString`:

```
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 + ", paginas=" + paginas
    }
}
```

5) No método `gravar`, de `ProdutosController`, ao invés de passar todos aqueles parâmetros, envie apenas um `Produto` e imprima-o dentro do método:

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

6) Agora, crie a view **ok.jsp**, dentro **src/main/webapp/WEB-INF/views/produtos**, com o seguinte conteúdo:

```
<%@ 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, PHP, Ruby e muito mais - Casa do Código</title>
</head>
<body>
    <h1>Produto cadastrado com sucesso!</h1>
</body>
</html>
```

Spring com JPA

7) Adicione a dependência do Hibernate ao seu projeto. Abra o arquivo **pom.xml** e dentro da *tag* `<dependencies>`, adicione o seguinte XML:

```
<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>
```

8) Para transformar as classes modelos como entidades do banco de dados, anote-as com **`@Entity`**. Então, faça isso com a classe `Produto`:

```
@Entity
public class Produto {

    // restante do código omitido

}
```

9) Além disso, o Hibernate obriga que **toda entidade precisa de um `id`**, ou seja, um campo que contenha um valor único para cada registro. Atualmente, a classe `Produto` não tem nenhum atributo que possa ser um *id*, portanto, crie mais um atributo, chamado `id`, do tipo `int`, e também gere seus *getters* e *setters*:

```
@Entity
public class Produto {

    private int id;

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    // restante do código omitido

}
```

10) Para o Hibernate entender que o atributo `id` é o *id* da entidade, adicione a anotação **`@Id`** no atributo. Além disso, popule-o antes de persisti-lo, fazendo com que o próprio banco já atribua um valor do `id` automaticamente, anotando-o com **`@GeneratedValue`**.

Por fim, informe ao Hibernate como ele deve atribuir esse `id` automaticamente a partir do atributo `strategy` da anotação `@GeneratedValue`. Nesse caso, faça com que ele seja auto-incremental, enviando o parâmetro **`strategy = GenerationType.IDENTITY`**:

```
@Entity
public class Produto {

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

    // restante do código omitido
```

```
}
```

11) Crie a classe `JPAConfiguration`, no nosso pacote **br.com.casadocodigo.loja.conf**, que será a responsável por configurar o framework, passando informações relevantes como o banco a ser utilizado, seu usuário e senha, e assim por diante:

```
public class JPAConfiguration {

    @Bean
    public LocalContainerEntityManagerFactoryBean entityManagerFactory() {

        LocalContainerEntityManagerFactoryBean factoryBean =
            new LocalContainerEntityManagerFactoryBean();

        JpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();

        factoryBean.setJpaVendorAdapter(vendorAdapter);

        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setUsername("root");
        dataSource.setPassword(""); // modifique para a senha do seu banco
        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.MySQL5Dialect");
        props.setProperty("hibernate.show_sql", "true");
        props.setProperty("hibernate.hbm2ddl.auto", "update");
        factoryBean.setJpaProperties(props);

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

        return factoryBean;

    }
}
```

12) Crie a classe `ProdutoDAO`, no pacote **`br.com.casadocodigo.loja.daos`**. Dentro dessa classe, faça a comunicação com o banco de dados, adicionando um atributo do tipo `EntityManager`, e crie o método `gravar()`, que recebe um produto como parâmetro, e então, a partir do atributo do tipo `EntityManager`, persista um objeto a partir do método `persist()`, enviando o parâmetro do tipo `Produto` como argumento.

Já que o `EntityManager` trata-se de um recurso persistente, utilize a anotação `@PersistenceContext` para que ele seja injetável:

```
@Repository
public class ProdutoDAO {

    @PersistenceContext
    private EntityManager manager;
```

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

13) Dentro do `ProdutosController`, crie um atributo do tipo `ProdutoDAO` e anote com `@Autowired`, para que ele seja injetado. Em seguida, dentro do método `gravar()` do *controller*, use o atributo do tipo `ProdutoDAO` e chame o método `gravar()`, enviando o produto recebido por parâmetro:

```
@Controller
public class ProdutosController {

    @Autowired
    private ProdutoDAO produtoDao;

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

    @RequestMapping("/produtos")
    public String gravar(Produto produto) {
        produtoDao.gravar(produto);

        return "produtos/ok";
    }
}
```

14) Para o Spring encontrar o DAO, adicione-o na anotação **`@ComponentScan`**, na classe **`AppWebConfiguration`**:

```
@EnableWebMvc
@ComponentScan(basePackageClasses={HomeController.class, ProdutoDAO.class})
public class AppWebConfiguration {

    // restante do código omitido

}
```

15) E para o Spring encontrar a classe `JPAConfiguration`, adicione-a no método `getServletConfigClasses`, da classe `ServletSpringMVC`:

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

Configurando o TransactionManager

16) Para o Spring gerenciar as transações para nós, adicione a anotação `@EnableTransactionManagement` na classe `JPAConfiguration`:

```
@EnableTransactionManagement
public class JPAConfiguration {

    // restante do código omitido

}
```

17) Em seguida, adicione um *bean* que será o gerenciador das transações, isto é, a partir desse *bean*, o Spring fornecerá as transações para o `EntityManager`. Para isso adicione o código abaixo na classe `JPAConfiguration`:

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

18) O `ProdutoDAO` é um recurso persistente (persiste dados) dentro do sistema, portanto, anote-o com `@Repository`. E embora o Spring esteja configurado para gerenciar as transações, ainda é necessário indicar que o **`ProdutoDAO` precisa de uma transação**. Faça isso anotando-o com `@Transactional`:

```
@Repository
@Transactional
public class ProdutoDAO {

    @PersistenceContext
    private EntityManager manager;

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

19) Agora, crie a base de dados **`casadocodigo`** no MySQL:

```
mysql -u root
mysql> create database casadocodigo;
```

20) Por fim, reinicie o Tomcat e acesse a URL http://localhost:8080/casadocodigo/produtos/form (http://localhost:8080/casadocodigo/produtos/form) e cadastre um produto. Se ele for cadastrado com sucesso, veja-o criado na base de dados.