02

Expondo atributos

Transcrição

Nas aulas anteriores, fizemos a página de detalhes de produto. Nela, além de exibirmos todos os detalhes do produto, construímos um formulário em que o usuário pode selecionar a opção de preço e clicar no botão de adicionar produto ao carrinho.

Observe que a `action` deste formulário aponta para uma url diferente. Ela aponta para `/carrinho/add`. Não temos nenhum código que atenda as requisições enviadas para este endereço. Até o momento só temos classes que atendam as requisições que usam o caminho `/produtos`.

Veja como está atualmente o formulário de compra de produtos.

```
<form action="/carrinho/add" method="post" class="container">
    <ul id="variants" class="clearfix">
        <input type="hidden" name="produtoId" value="${produto.id}" />
        <c:forEach items="${produto.precos}" var="preco">
          <li class="buy-option">
            <input type="radio" name="tipo" class="variant-radio" id="tipo" value="${preco.tipo
            <label  class="variant-label">
              ${preco.tipo}
            </label>
            <small class="compare-at-price">R$ 39,90</small>
            <p class="variant-price">${preco.valor}</p>
          </li>
        </c:forEach>
    </ul>
    <button type="submit" class="submit-image icon-basket-alt" alt="Compre Agora" title="Compre
</form>
```

Precisamos criar um novo *controller*, assim as requisições enviadas para o caminho `/carrinho` serão atendidas da mesma forma que fizemos com os produtos. Crie então uma nova classe chamada `CarrinhoComprasController` e não se esqueça de deixa-lo no pacote `br.com.casadocodigo.loja.controllers`. Faça uso também das anotações `@Controller` e `@RequestMapping` com o valor `/carrinho`, para que o Spring reconheça esta classe como um `controller` e que o mapeamento da rota seja feito corretamente. Até então teremos a classe `CarrinhoComprasController`:

```
@Controller
@RequestMapping("/carrinho")
public class CarrinhoComprasController {

}
```

A primeira funcionalidade que vamos codificar é a adição de produtos ao carrinho de compras. Vamos começar adicionando um método chamado `add` na classe `CarrinhoComprasController` que retorna um objeto do tipo `ModelAndView`. Este método precisa receber dois parâmetros que são: O id do produto e o tipo de preço selecionado pelo usuário. Lembre-se que apenas o `id` do produto está sendo enviado pelo formulário da página de detalhes e não o produto.

Ao escolher um produto faremos com que o método redirecione o usuário para a listagem de produtos. Sendo assim, até aqui teremos:

```
@RequestMapping("/add")
public ModelAndView add(Integer produtoId, TipoPreco tipo){
    ModelAndView modelAndView = new ModelAndView("redirect:/produtos");
    return modelAndView;
}
```

Observação: Perceba o uso da anotação `@RequestMapping("/add")` que faz o mapeamento da rota `/carrinho/add` para este método no `CarrinhoComprasController` .

Abaixo, vemos a classe `CarrinhoComprasController` com o código completo escrito até aqui:

```
@Controller
@RequestMapping("/carrinho")
public class CarrinhoComprasController {

    @RequestMapping("/add")
    public ModelAndView add(Integer produtoId, TipoPreco tipo){
        ModelAndView modelAndView = new ModelAndView("redirect:/produtos");

        return modelAndView;
    }

}
```

O que precisamos fazer agora é definir uma forma de relacionar o produto com seu preço. Podemos fazer isso de uma forma muito simples: criando um outro objeto que represente este relacionamento. O objeto será criado através da classe `CarrinhoItem` que iremos codificar agora. Esta só terá os atributos `produto` e `tipoPreco` . Um construtor que receba estes dois parâmetros e os ***Getters and Setters*** destes mesmos atributos. Assim teremos a classe `CarrinhoItem` :

```
public class CarrinhoItem {

    private TipoPreco tipoPreco;
    private Produto produto;

    public CarrinhoItem(Produto produto, TipoPreco tipoPreco) {
        this.produto = produto;
        this.tipoPreco = tipoPreco;
    }

    public TipoPreco getTipoPreco() {
        return tipoPreco;
    }

    public void setTipoPreco(TipoPreco tipoPreco) {
        this.tipoPreco = tipoPreco;
    }

    public Produto getProduto() {
        return produto;
    }
```

```
    public void setProduto(Produto produto) {
        this.produto = produto;
    }
}
```

Observação: Por se tratar de uma classe de negócio, esta ficará no pacote `br.com.casadocodigo.loja.models`. Ela não é um `controller` nem uma `view` e contém uma lógica importante para o negócio.

Nosso próximo passo será criar um método privado na classe `CarrinhoComprasController` para que ele crie e retorne um objeto da classe `CarrinhoItem`. Busque o produto pelo `id` no banco de dados e relacione o produto com o preço selecionado pelo usuário. Chamaremos este método de `criaItem`. Como estamos falando de buscar produtos no banco de dados, criaremos também um atributo na classe `CarrinhoComprasController` do tipo `ProdutoDAO` chamado `produtoDao`, com a anotação `@Autowired`. Iremos usá-lo para a busca do produto.

```
@Autowired
private ProdutoDAO produtoDao;

private CarrinhoItem criaItem(Integer produtoId, TipoPreco tipo){
    Produto produto = produtoDao.find(produtoId);
    CarrinhoItem carrinhoItem = new CarrinhoItem(produto, tipo);
    return carrinhoItem;
}
```

Agora temos que usar o método dentro do método `add` da classe `CarrinhoComprasController`, passando para ele o `produtoId` e o `tipo` de preço. Então receberemos o objeto retornado pelo método `criaItem`.

```
CarrinhoItem carrinhoItem = criaItem(produtoId, tipo);
```

O médoto `add` até o momento ficará assim:

```
@RequestMapping("/add")
public ModelAndView add(Integer produtoId, TipoPreco tipo){
    ModelAndView modelAndView = new ModelAndView("redirect:/produtos");
    CarrinhoItem carrinhoItem = criaItem(produtoId, tipo);
    return modelAndView;
}
```

O que fica faltando para terminarmos a funcionalidade de adicionar produtos ao carrinho de compras é: O próprio carrinho, incluindo pelo menos um método que faça a adição do `carrinhoItem` a uma lista. Vamos fazer isso, então.

Crie uma nova classe chamada `CarrinhoCompras` no pacote br.com.casadocodigo.loja.models e logo em seguida crie um método que não tem retorno, ou seja, `void` chamado `add`. Este método deve receber um objeto da classe `CarrinhoItem` que é a classe que representa um item em nosso carrinho.

```
public class CarrinhoCompras {
    public void add(CarrinhoItem item){
```

```
    }
}
```

Algo comum nas lojas online é que ao selecionarmos determinado produto, podemos também especificar a quantidade. Nossa página de detalhes não permite esse tipo de comportamento, mas em no carrinho de compras fará muito sentido incluir este tipo de informação. Por isso precisaremos de uma lista que associe o item a sua quantidade. Neste caso, usaremos uma do tipo `Map`, que permitirá a associação entre uma chave, que será o item do carrinho, e um valor, que será a quantidade total de cópias do mesmo produto.

Logo, adicionaremos à classe `carrinho` um atributo privado chamado `itens`, do tipo `Map<CarrinhoItem, Integer>`, e o instanciaremos com um objeto da classe `LinkedHashMap`.

```
private Map<CarrinhoItem, Integer> itens = new LinkedHashMap<CarrinhoItem, Integer>();
```

Com a lista pronta, precisaremos que ao adicionarmos um produto, seja verificado se o produto já existe em nossa lista. Caso exista, somaremos `+1` na quantidade, caso não exista, adicionamos o produto selecionado. Veja o código abaixo:

```
public void add(CarrinhoItem item){
    itens.put(item, getQuantidade(item) + 1);
}
```

Apesar de não termos o método `getQuantidade` ainda, pode-se notar que a lógica é simples. Ele deve retornar o número de vezes em que o produto foi encontrado na lista e somar `1` a essa quantidade. Agora vejamos como fica essa lógica em código.

```
private int getQuantidade(CarrinhoItem item) {
    if(!itens.containsKey(item)){
        itens.put(item, 0);
    }
    return itens.get(item);
}
```

Perceba que estamos usando o método `containsKey` que verifica se a lista tem a chave, que no caso é o item do carrinho. Estamos usando `!` que é o sinal de negação da expressão lógica. Ou seja, se a lista não tiver o item, este será adicionado com o valor `0` na mesma. E então o método `get` usado no `return` retorna o valor que está associado aquela chave.

Explicando melhor... Caso o item não exista na lista, colocamos o item e retornamos o valor 0, que será incrementado pelo metódo `add`. Mas caso o item já exista, retornamos apenas o valor que representa a quantidade de vezes que o produto foi adicionado na lista.

Apesar de usar o método `containsKey` não é o suficiente. Ele usa o método `equals` disponível na classe `Object`. Para que o método `containsKey` consiga comparar corretamente os itens da lista, devemos sobrescrever dois métodos na classe `CarrinhoItem` e depois, na classe `Produto`. Use os geradores do Eclipse e selecione a opção ***Generate HashCode and Equals***. Na classe `CarrinhoItem`, teremos algo parecido com:

```
@Override
public int hashCode() {
```

```
    final int prime = 31;
    int result = 1;
    result = prime * result + ((produto == null) ? 0 : produto.hashCode());
    result = prime * result + ((tipoPreco == null) ? 0 : tipoPreco.hashCode());
    return result;
}

@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    CarrinhoItem other = (CarrinhoItem) obj;
    if (produto == null) {
        if (other.produto != null)
            return false;
    } else if (!produto.equals(other.produto))
        return false;
    if (tipoPreco != other.tipoPreco)
        return false;
    return true;
}
```

Observação: Lembre-se que estes métodos são gerados a partir dos atributos da classe. No caso da classe `CarrinhoItem`, deixamos todos os atributos selecionados, mas no caso da classe `Produto`, usaremos apenas o atributo `id` que é gerado pelo banco de dados e nos garante que o produto é único. Na classe produto, teremos:

```
@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + id;
    return result;
}

@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Produto other = (Produto) obj;
    if (id != other.id)
        return false;
    return true;
}
```

Com tudo isso feito, estamos quase prontos para testar se o carrinho de compras da nossa aplicação está realmente funcionando. O último passo é usar a anotação `@Component` na classe `CarrinhoCompras` para que o **Spring** instancie automaticamente objetos desta classe quando necessário. Depois, a classe ficará desta forma:

```
@Component
public class CarrinhoCompras {
    private Map<CarrinhoItem, Integer> itens = new LinkedHashMap<CarrinhoItem, Integer>();

    public void add(CarrinhoItem item){
        itens.put(item, getQuantidade(item) + 1);
    }

    private int getQuantidade(CarrinhoItem item) {
        if(!itens.containsKey(item)){
            itens.put(item, 0);
        }
        return itens.get(item);
    }

}
```

Lembre-se de que ainda não estamos adicionando os produtos ao carrinho de compras. Precisamos fazer isso no método `add` da classe `CarrinhoComprasController`. Primeiro faremos com que o **Spring** instancie para nós o carrinho de compras. Fazemos isso criando um atributo no *controller* em questão, que chamaremos de `carrinho` e o anotaremos com `@Autowired`.

```
@Controller
@RequestMapping("/carrinho")
public class CarrinhoComprasController {

    @Autowired
    private CarrinhoCompras carrinho;
}
```

Após isso, no método `add` da classe `CarrinhoComprasController`, usaremos o método `add` do objeto `carrinho`, passando o objeto `carrinhoItem` para que este seja adicionado ao carrinho de compras.

Até aqui o método `add` da classe `CarrinhoComprasController` está assim:

```
@RequestMapping("/add")
public ModelAndView add(Integer produtoId, TipoPreco tipo){
    ModelAndView modelAndView = new ModelAndView("redirect:/produtos");
    CarrinhoItem carrinhoItem = criaItem(produtoId, tipo);
    return modelAndView;
}
```

Em seguida, ficará assim:

```
@RequestMapping("/add")
public ModelAndView add(Integer produtoId, TipoPreco tipo){
    ModelAndView modelAndView = new ModelAndView("redirect:/produtos");
    CarrinhoItem carrinhoItem = criaItem(produtoId, tipo);
    carrinho.add(carrinhoItem);
    return modelAndView;
}
```

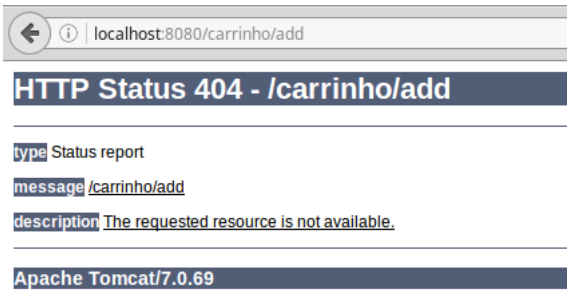
Podemos iniciar agora o servidor `Tomcat` para podermos testar essa nova funcionalidade. Mas ao iniciar, recebemos uma mensagem de erro:

```
No qualifying bean of type [br.com.casadocodigo.loja.models.CarrinhoCompras] found for dependen
```

Já vimos este erro antes. O **Spring** não está conseguindo encontrar a classe `CarrinhoCompras`. Isto porque o pacote `models` não está sendo scaneado pelo **Spring**. Da mesma forma que fizemos com o `ProdutoDAO` outra vez, faremos agora com o `CarrinhoCompras`. Na anotação `@ComponentScan` da classe `AppWebConfiguration` adicionaremos a classe `CarrinhoCompras`.

```
@EnableWebMvc
@ComponentScan(basePackageClasses={HomeController.class, ProdutoDAO.class, FileSaver.class, Carı
public class AppWebConfiguration {
    [...]
}
```

Se tentarmos testar novamente agora, não iremos receber nenhum erro do servidor. Então, tente acessar a listagem de produtos, escolher um tipo de preço e clicar no botão de adicionar produto ao carrinho. Teremos um novo erro.



Note a URL da página, ela não tem o contexto da aplicação. O contexto seria o `/casadocodigo` que é o nome do projeto. A URL da página fica com o endereço `localhost:8080/carrinho/add`. Para corrigir isso, usaremos uma tag da `taglib` core da `JSTL` no formulário da página de detalhes do produto.

No Arquivo `detalhes.jsp` na tag `form`, teremos o atribudo `action` com o valor `/carrinho/add`. Este tipo de endereço, faz com que o contexto da aplicação seja perdido. A tag `c:url` da `JSTL` cria URLs baseadas no contexto. Ela resolverá nosso problema. Antes das alterações, a `action` do formulário em `detalhes.jsp` está assim:

```
[...]
    <form action="/carrinho/add" method="post" class="container">
[...]
```

Agora, ela ficará assim:

```
[...]
    <form action='<c:url value="/carrinho/add" />' method="post" class="container">
[...]
```

Observação: Fique atento sobre as **aspas** nesta modificação, foi necessária a mudança por causa de conflitos que seriam gerados se usássemos aspas iguais no atribudo `action` e na tag `c:url`. Agora, estará tudo funcionando normalmente.

Da página de detalhes somos levados à listagem dos produtos quando selecionados o tipo de preço do produto e clicamos na opção de adicionar o produto ao carrinho. Mas nada indica que o produto foi realmente adicionado. Vamos fazer com que o número de produtos no carrinho seja exibido junto ao link carrinho encontrado na página de detalhes. No seguinte trecho de código:

```
<li><a href="/cart" rel="nofollow">Carrinho</a></li>
```

Faremos algo do tipo:

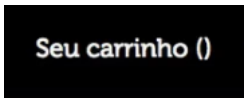
```
<li><a href="/cart" rel="nofollow">Carrinho (${carrinhoCompras.quantidade}) </a></li>
```

A classe `CarrinhoCompras` já tem um método `getQuantidade` , mas ele só retorna a quantidade de um produto especifico. Criaremos um novo método `getQuantidade` que irá iterar entre todos os itens do carrinho e contar quantos produtos estão na lista de itens. Observe o seguinte código:

```
public int getQuantidade(){
    return itens.values().stream().reduce(0, (proximo, acumulador) -> (proximo + acumulador));
}
```

Este código percorre toda a lista de itens e soma o valor de cada item a um aculumador. Caso não conheça a API de Stream e Lambdas, que são recursos do Java 8, recomendamos que faça o curso de java 8 (https://www.alura.com.br/curso-online-java8-lambdas) disponível aqui no Alura.

Veja o link de menu como se encontra após estas modificações:



Apesar de parecer funcionar, não funciona. Ao testarmos adicionar alguns produtos no carrinho, veremos que não há um número indicando que a contagem dos itens está sendo feita corretamente. Isto acontece porque o objeto que representa o carrinho de compras não está disponível em lugar nenhum a não ser dentro do escopo do **Spring**.

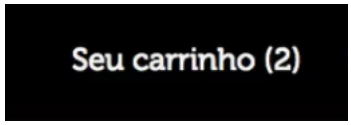
Anteriormente, marcamos a classe `CarrinhoCompras` com a anotação `@Component` . Essa anotação indica que a classe será tratada como um ***Bean*** do **Spring**. Para que possamos acessar esse ***Bean*** em nossas `view` , precisaremos adicionar uma configuração na classe `WebAppConfiguration` . No método `InternalResourceViewResolver` poderiamos usar o método `setExposeContextBeansAsAttributes` do objeto `resolver` com o valor `true` , mas esta configuração tornará todos os ***Beans*** da aplicação disponíveis, o que parece não ser uma boa ideia.

Ao invés disso, usaremos o método `setExposedContextBeanNames` deste mesmo objeto. Este método nos permite dizer qual ***Bean** estará disponível para a `view` . Os nomes dos ***Beans*** seguem um padrão bem simples. O padrão é o nome da classe com sua primeira em minúsculo, ou seja, a classe `CarrinhoCompras` fica `carrinhoCompras` . Com esta mudança o método `InternalResourceViewResolver` da classe `WebAppConfiguration` fica assim:

```
@Bean
public InternalResourceViewResolver internalResourceViewResolver(){
    InternalResourceViewResolver resolver = new InternalResourceViewResolver();
    resolver.setPrefix("/WEB-INF/views/");
    resolver.setSuffix(".jsp");
```

```
        resolver.setExposedContextBeanNames("carrinhoCompras");
        return resolver;
    }
```

Teste novamente, verá que agora a contagem acontece normalmente e é exibida na parte superior da página de detalhes da seguinte forma:



Até aqui, fizemos uma séria de adições em nossa aplicação. Criamos o carrinho de compras, relacionamos os produtos e seus tipos de preço atravé de um item de carrinho e exibimos a contagem de itens no carrinho em nossa `view`.

Caso tenha dúvidas, fique a vontade para perguntar no forúm. Agora vamos praticar um pouco mais com alguns exercícios.