02

Cadastro do Sorteio

Download

Caso queira começar o treinamento a partir desse vídeo, pode baixar o projeto aqui (https://s3.amazonaws.com/caelum-online-public/auron/auron-stage4.zip). **Só baixe este arquivo se não tiver feito os exercícios dos capítulos anteriores.**

Cadastro do Sorteio

Neste capítulo, começaremos a implementar a lógica do sorteio. A ideia é simples: quando cadastrarmos um sorteio, automaticamente serão sorteados seus pares, ou seja, os amigos e amigos ocultos para este sorteio.

Começaremos criando a tela de sorteio através dos componentes do JSF. Aqui não há novidade, criaremos um novo arquivo HTML pelo Eclipse. O arquivo se chamará `sorteio.xhtml` do tipo `xhtml 1.0 transitional`. Uma vez criado, vamos abrir o arquivo e colocar o namespace padrão do JSF, igual à página anterior.

Dentro da tag `<h:body>` adicionaremos um formulário, novamente usando o componente `<h:form>`. Nesse formulário também usaremos o `<h:panelGrid>` para organizar os componentes. Dentro dele fica um label e um campo de texto usando os componentes `<h:outputText>` e `<h:inputText>` respectivamente.

Vamos testar uma vez a página e carregá-la no navegador. Nosso formulário é exibido conforme esperado. Por fim, criaremos um `<h:commandButton>` para submetermos o formulário. O conteúdo completo do arquivo pode ser visto abaixo:

```
<?xml version="1.0" encoding="US-ASCII" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:h="http://java.sun.com/jsf/html">

 <h:head>
    <meta http-equiv="Content-Type" content="text/html; charset=US-ASCII" />
    <title>Sorteio</title>
 </h:head>
 <h:body>
   <h:form>
    <h:panelGrid columns="2">
        <h:outputLabel value="Nome: " />
        <h:inputText  />
        <h:commandButton value="Cadastrar" />
    </h:panelGrid>
   </h:form>
 </h:body>
</html>
```

ManagedBean do Sorteio

Uma boa prática no JSF é criar um bean separado para cada tela. Desse jeito não misturaremos as responsabilidades entre telas e facilitaremos a manutenção.

Não faremos diferente com nosso formulário, então criaremos uma nova classe chamada `SorteioBean` que receberá seus dados. A classe ficará dentro do pacote `br.com.caelum.auron.beans` .

```
package br.com.caelum.auron.beans;

public class SorteioBean {
}
```

A classe `SorteioBean` será anotada com as já conhecidas anotações `@Named` e `@RequestScoped` :

```
package br.com.caelum.auron.beans;

import javax.enterprise.context.RequestScoped;
import javax.inject.Named;

@Named
@RequestScoped
public class SorteioBean {
}
```

Mapeamento do relacionamento

Para receber os dados do formulário criaremos um novo atributo do tipo `Sorteio` no `SorteioBean` . Como ainda não existe a classe `Sorteio` , vamos gerá-la pelo Eclipse. De forma similar à classe `Participante` , o `Sorteio` também estará presente no pacote `modelo` . A classe será bem simples com apenas dois atributos: `id` e `nome` . Além disso devemos usar as conhecidas anotação de JPA: `@Entity` , `@Id` e `@GeneratedValue` :

```
@Entity
public class Sorteio {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Integer id;
    private String nome;

}
```

A classe `Sorteio` se relacionará com a classe `Participante` através de uma classe intermediária. Um `Sorteio` é composto de *pares* e cada par possui dois participantes. Os participantes são o amigo e amigo oculto.

Ou seja, na classe `Sorteio` teremos uma coleção de pares. Como não pode haver nenhum par repetido, usaremos um `Set` como tipo. A implementação será o `LinkedHashSet` pois a ordem nessa coleção importa. Queremos que os pares sempre mantenham a posição de inserção:

Vamos aproveitar e também mapear o relacionamento através da anotação `@OneToMany` :

```
@Entity
public class Sorteio {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Integer id;
    private String nome;

    @OneToMany
    private Set<Par> pares = new LinkedHashSet<>();

}
```

Da mesma forma que fizemos no exemplo anterior, criaremos a classe `Par`. Nela usaremos as mesmas anotações básicas do JPA, `@Entity`, `@Id` e `@GeneratedValue`.

Como já foi mencionado, um par representa a ligação entre `Sorteio` e `Participante`. Para ser mais concreto, a classe `Par` terá dois participantes, um é o amigo e outro é o amigo oculto, além de um outro atributo que representa qual foi `sorteio`. Todos os atributos possuem a cardinalidade *muitos-para-um*, usa-se então a anotação `@ManyToOne`.

```
@Entity
public class Par {

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

    @ManyToOne
    private Participante amigo;

    @ManyToOne
    private Participante amigoOculto;

    @ManyToOne
    private Sorteio sorteio;
}
```

Depois disso, geraremos o construtor baseado nos atributos `amigo`, `amigoOculto` e `sorteio`, ou seja sem o atributo `id`. Para o JPA continuar funcionando é preciso criar o construtor padrão, no entanto ele não será público.

Faltou gerar os getters e setters na classe `Sorteio` para cada atributo que precisaremos para a tela JSF funcionar. Além dos métodos de acesso, criaremos um método que simplifica a adição de um par no sorteio. O método se chama `adicionaPar` e recebe um par que colocaremos na coleção.

Depois que um sorteio for realizado, nenhuma outra classe poderá alterar seus pares. Para evitar o acesso indevido, devolveremos um `Set` *inalterável*. No getter `getPares` usaremos `Collections.unmodifiableSet(this.pares)`.

Por fim, criaremos um relacionamento bidirecional entre `Sorteio` e `Par`, mas apenas um lado pode ser *lado forte*, o lado que define o banco de dados. No nosso caso a classe `Par` define o relacionamento, ou seja, é o lado forte. O lado do `Sorteio` define *apenas* o lado fraco do relacionamento, indicado pelo atributo `mappedBy` da anotação `@OneToMany`. `mappedBy` significa que o relacionamento já foi mapeado e recebe o nome da atributo que representa o lado forte, o `sorteio`.

Veja o código completo:

```
@Entity
public class Sorteio {

    //outros atributos omitidos

    @OneToMany(mappedBy="sorteio")
    private Set<Par> pares = new LinkedHashSet<>();

    public Set<Par> getPares() {
        return Collections.unmodifiableSet(this.pares);
    }

    public void adicionaPar(Par par) {
        this.pares.add(par);
    }
}
```

Terminamos o mapeamento! Vamos voltar para a classe `SorteioBean`. Para os componentes JSF poderem acessar o `Sorteio`, precisamos criar seu getter. Além do getter, adicionaremos também o método que será chamado pelo botão.

Vamos chamar o método de `sortear` que imprime apenas uma mensagem no console por enquanto:

```
package br.com.caelum.auron.beans;

import javax.enterprise.context.RequestScoped;
import javax.inject.Named;

import br.com.caelum.auron.modelo.Sorteio;

@Named
@RequestScoped
public class SorteioBean {

    private Sorteio sorteio = new Sorteio();

    public Sorteio getSorteio() {
        return sorteio;
    }

    public void sortear() {
        System.out.println("Sorteio:  " + sorteio.getNome());
    }

}
```

Antes de testarmos pelo navegador, não podemos esquecer de realizar o binding dos componentes JSF com o bean. No arquivo xhtml ligaremos o input ao atributo `nome` da classe `Sorteio` e o botão chamará o método `sortear` da classe `SorteioBean`. Muito bem, está tudo pronto para testar.

Vamos realizar um *Full publish* para garantir a atualização do projeto no Wildfly. Em seguida, abriremos a página no navegador. No preenchimento e submissão do formulário, deve aparecer uma mensagem com o nome no console

indicando que o binding funcionou corretamente.

Já estamos chamando o método `sortear` pelo formulário, mas ainda falta implementar a lógica do sorteio. Isso ficará para o próximo capítulo, pois agora é a hora dos exercícios.