02

Cadastro de participante

Download

Caso queira começar o treinamento a partir desse vídeo, pode baixar o projeto aqui (https://s3.amazonaws.com/caelum-online-public/auron/auron-stage2.zip). **Só baixe este arquivo se não tiver feito os exercícios do capítulo anterior.**

Formulário JSF

Nesse vídeo vamos implementar a primeira funcionalidade: o cadastro dos participantes. No vídeo anterior já criamos uma página JSF, o `index.xhtml`. Agora vamos adicionar os campos de um participante.

O primeiro campo é referente ao nome dele. Vamos usar os componentes `<h:outputText ... />` para a definição da label e o `<h:inputText ... />` para o campo de texto.

```
<h:outputText value="Nome: " />
<h:inputText />
```

O mesmo se repete para o campo *email* e *senha*, sempre definindo `<h:outputText ... />` e `<h:inputText ... />`. Porém, no campo *senha*, usamos o componente `<h:inputSecret ... />` para esconder o conteúdo na hora de digitar a senha. Por fim, criaremos um botão para submeter o formulário usando o componente `<h:commandButton />`.

Vamos testar uma vez o resultado e acessar a página pelo navegador: http://localhost:8080/auron/faces/index.xhtml (http://localhost:8080/auron/faces/index.xhtml).

Podemos ver que os campos ficam um ao lado do outro. Para melhorar a organização dos componentes podemos usar o `<h:panelGrid />` que permite definir colunas:

```
<h:panelGrid columns="2">
    <h:outputText value="Nome: " />
    <h:inputText  />

    <h:outputText value="Email: " />
    <h:inputText  />

    <h:outputText value="Senha: " />
    <h:inputSecret  />

    <h:commandButton value="Cadastrar" />
</h:panelGrid>
```

Vamos carregar novamente a página `index.xhtml` e podemos ver que a organização dos componentes melhorou: http://localhost:8080/auron/faces/index.xhtml (http://localhost:8080/auron/faces/index.xhtml)

O primeiro Bean

Para recebermos os dados e eventos dos componentes JSF vamos criar um *ManagedBean* com o nome `ParticipanteBean`. Ele fica dentro do pacote `beans`:

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

public class ParticipanteBean {
```

Para expormos e usarmos o bean através dos componentes JSF vamos usar a anotação `@Named` do CDI que fica no pacote `javax.inject`. Assim podemos acessar esse bean pela *Expression Language* do JSF. Além disso, colocaremos o `ParticipanteBean` no escopo da requisição configurado pela anotação `@RequestScoped` do pacote `javax.enterprise.context`:

O bean terá um atributo que representa o participante. Já na declaração do atributos vamos dar `new`.

```
@Named
@RequestScoped
public class ParticpanteBean {

    private Participante participante = new Participante();
}
```

Criando o modelo e o mapeamento

Como a classe ainda não existe podemos gerá-la pelo Eclipse. O pacote da classe Participante será o `br.com.caelum.auron.modelo`.

A classe `Participante` é uma entidade, logo vamos anotá-la com `@Entity`. Os atributos são *id, nome, email e senha*.

O *id* representa a chave primária indicado pelas anotações `@Id` e `@GeneratedValue`. Por fim geraremos os getters e setters para cada atributo.

```
@Entity
public class Participante {

    @Id
    @GeneratedValue
    private Integer id;

    private String nome;
    private String email;
    private String senha;

    //getters e setters
}
```

Completando o bean e formulário

Voltando ao `ParticipanteBean` vamos gerar o getter para o atributo `participante`. Além disso, criaremos um método para o componente `<h:commandButton ... />`. O método `cadastrar` será executado ao clicar no `button`.

```
@Named
@RequestScoped
public class ParticipanteBean {

    private Participante participante = new Participante();

    public void cadastrar() {
        System.out.println(participante.getNome());
    }
    //getter
}
```

Agora só falta criar a ligação, ou `binding`, dos componentes JSF com as classes criadas. Para isso usaremos a Expression Language no atributo `value` de cada input.

O primeiro `inputText` usa `#{participanteBean.participante.nome}`.

O segundo usa `#{participanteBean.participante.email}`.

E a senha faz o binding `#{participanteBean.participante.senha}`

No `commandButton`, a Expression Language deve ser usada no atributo `action`, pois queremos chamar o método `cadastrar` da classe `ParticipanteBean`. Nesse caso fica no atributo a expressão `#{participanteBean.cadastrar}`.

Para testar vamos chamar novamente a página pelo navegador. Após ter preenchido o formulário e clicado no botão podemos verificar o console. No entanto, o resultado não é o esperado!

Esquecemos um detalhe importante, inputs e commands só funcionam corretamente dentro de formulário. Por isso adicionaremos acima do `<h:panelGrid>` o componente `<h:form>`. Não devemos esquecer de fechar a tag.

Ao testarmos novamente, verificamos o nome do participante no console de Eclipse.

Segue o arquivo xhtml completo:

```
<?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>Insert title here</title>
</h:head>
<h:body>
    <h:form>
        <h:panelGrid columns="2">
            <h:outputText value="Nome: " />
            <h:inputText value="#{participanteBean.participante.nome}" />

            <h:outputText value="Email: " />
            <h:inputText value="#{participanteBean.participante.email}" />

            <h:outputText value="Senha: " />
            <h:inputSecret value="#{participanteBean.participante.senha}" />
```

```
            <h:commandButton action="#{participanteBean.cadastrar}"
                value="Cadastrar" />
        </h:panelGrid>
    </h:form>
</h:body>
</html>
```

Persistência com EJB e JPA

Para finalizarmos este capítulo persistiremos o participante no banco de dados. Já vimos que foi criado um `persistence.xml` pelo JBoss Forge, o JPA já veio configurado. Nada nos impede de começarmos a utilizar JPA dentro de um EJB Session Bean.

Vamos criar um Dao que recebe um `EntityManager` para trabalhar com o banco de dados. A classe `ParticipanteDao` fica dentro do pacote `br.com.caelum.auron.dao` .

Essa classe `ParticipanteDao` será um Session Bean do tipo `@Stateless` . O `EntityManager` vamos injetar usando a anotação `@PersistenceContext` . Por enquanto teremos apenas um método que insere o participante que vem do nosso formulário JSF. Vamos usar o método `persist` do `EntityManager` .

```
@Stateless
public class ParticipanteDao {

    @PersistenceContext
    private EntityManager em;

    public void inserir(Participante participante) {
        em.persist(participante);
    }

}
```

Agora só falta usar o Dao na classe `ParticipanteBean` . O CDI ajuda nessa configuração, basta usar a anotação `@Inject` em cima do atributo do tipo `ParticipanteDao` que iremos adicionar ao Bean. Já no método `cadastrar` delegamos para dao.

Segue uma vez o código completo:

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

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

import br.com.caelum.auron.dao.ParticipanteDao;
import br.com.caelum.auron.modelo.Participante;

@Named
@RequestScoped
public class ParticipanteBean {
```

```
    @Inject
    private ParticipanteDao participanteDao;

    private Participante participante = new Participante();


    public void cadastrar() {
        participanteDao.inserir(participante);
    }

    //getParticipante()
}
```

Para testar vamos publicar a nossa aplicação. Na view *Servers*, expanda o Wildfly e no projeto auron, basta selecionar o item *Full Publish* com o botão direito. Depois de termos carregado a aplicação podemos testar o resultado, acessando http://localhost:8080/auron/index.jsf (http://localhost:8080/auron/index.jsf).

H2 Database

Ao preencher e submeter o formulário vamos verificar a saída do console no Eclipse. Repare que apareceu o SQL gerado para inserir o participante no banco de dados. Mas a pergunta é: qual banco foi utilizado? A resposta está no `persistence.xml`. Ao gerar o projeto, o JBoss Forge aproveita o banco que já vem embutido: o H2 Database. O nosso `persistence.xml` aponta para um *datasource* que já veio com WildFly. Esse *datasource* usa o H2 Database, mas no próximo capítulo configuraremos o banco MySQL.