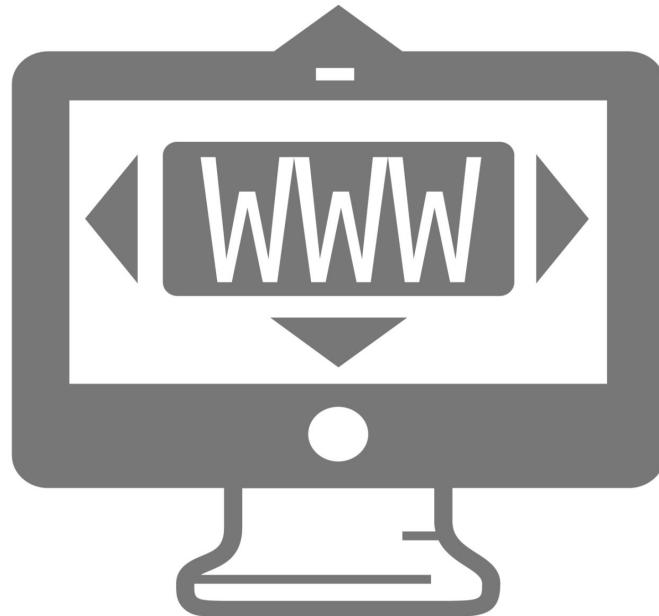


Web rica com JSF 2, Primefaces 4 e CDI

Curso FJ-26



Sumário

1 Laboratório Web com JSF e CDI	1
1.1 Construa aplicações web ricas de forma fácil	1
1.2 Integração com tecnologias do Java EE	2
1.3 Tirando dúvidas	2
1.4 Para onde ir depois?	2
2 Definição do projeto	4
2.1 Descrição do problema	4
2.2 Java EE Web Profile	5
2.3 Servidor de aplicação Glassfish	5
2.4 Modelo	6
2.5 Implementando o DAO	7
2.6 Exercícios: Instalando o Glassfish e criando o projeto	9
2.7 Para saber mais: Em casa	25
3 JavaServer Faces	27
3.1 Motivação: HTTP stateless	27
3.2 Motivação: Swing, Visual Basic e Delphi	28
3.3 JavaServer Faces (JSF)	30
3.4 Primeiro exemplo	31
3.5 Exercícios: primeira página	34
3.6 Criando o formulário de cadastro	35
3.7 Exercícios: a página de cadastro de produtos	37
3.8 Novidade do Java EE 7: JSF2.2 e HTML 5	38
3.9 Managed Beans	39
3.10 Ligando Managed Beans a componentes visuais	41
3.11 Exercícios: Gravação de produtos	44
3.12 Listagem com dataTable	46
3.13 Diferentes estilos de MVC: O MVC Push e o Pull	49
3.14 Exercícios: Listagem de produtos	51

Sumário	Caelum
3.15 Exercícios opcionais	57
3.16 Removendo um produto e parâmetros de EL	52
3.17 Exercícios: Remoção de produtos	53
3.18 Alteração de produtos e o setPropertyActionListener	54
3.19 Exercícios: Alterando produtos	56
3.20 Exercícios opcionais	57
4 Ajax com JSF 2	58
4.1 Ajax	58
4.2 Ajax e JSF	59
4.3 Como utilizar AJAX nas aplicações JSF	59
4.4 Exercícios: Remoção de produtos com Ajax	61
4.5 Para saber mais: A tag	62
4.6 Exercícios: Submetendo formulários com AJAX	63
5 Aplicando layouts à aplicação com CSS	64
5.1 Melhorando a interface gráfica	64
5.2 Integrando CSS com o JSF	65
5.3 Adicionando imagens na aplicação	66
5.4 Exercícios: Melhorando a interface gráfica	66
6 Entendendo JSF a fundo	69
6.1 Reduzindo a complexidade de outras APIs	69
6.2 A árvore de componentes	70
6.3 Alterando a forma de armazenamento da árvore de componentes	70
6.4 Compreendendo o ciclo de vida de uma requisição do JSF	71
6.5 Primeira fase: Restauração da view	71
6.6 Segunda fase: Aplicar os valores da requisição	72
6.7 Terceira fase: Converter e processar validações	72
6.8 Quarta fase: Atualização dos modelos	72
6.9 Quinta fase: Invocação da lógica	73
6.10 Sexta fase: Renderizar resposta	73
6.11 Exercícios: Fases do JSF	74
6.12 Debugando o ciclo de vida do JSF com um PhaseListener	75
6.13 Exercícios: Observando as fases do JSF	77
7 Login e navegação	79
7.1 Criando a funcionalidade de login	79
7.2 Exercício: Criando o formulário de Login	81
7.3 Navegação condicional	83

7.4 Navegação entre telas	83
7.5 Guardando dados em escopo de sessão	84
7.6 Exercício: Completando o Login	85
7.7 Exercício Opcional: Aplicando layout na tela de login	86
7.8 Redirecionamento durante a navegação	87
7.9 Exercício: Evitando submissões duplicadas	88
8 Injeção de Dependências com CDI	89
8.1 Problemas do alto acoplamento	89
8.2 Estratégias para diminuir o acoplamento	90
8.3 Injeção de dependências (DI)	91
8.4 Implementando DI com Contexts e Dependency Injection API - CDI	92
8.5 Liberação de dependências com @Disposes	95
8.6 Para saber mais: Configuração do Weld no Servlet Container	96
8.7 Exercícios: Utilizando CDI para injetar dependências	97
8.8 Melhorando a autenticação com CDI	99
8.9 Exercícios: Melhor gerenciamento dos escopos das dependências	101
8.10 Para saber mais: Injeção de Dependências e Testes	102
8.11 Interceptando as fases com PhaseListeners	103
8.12 Autorização com PhaseListeners	104
8.13 Exercícios: Autorização com PhaseListener	106
8.14 Exercícios: Logout e tela de cadastro de usuários	108
8.15 Exercícios: Migrar tudo para CDI	109
9 Templates com Facelets	110
9.1 Repetição de código e seus problemas	110
9.2 Resolvendo a duplicidade das Views	110
9.3 Templating com o Facelets	111
9.4 Exercícios: Templates com Facelets	114
9.5 Para saber mais: Campos padrões no template	116
9.6 Exercícios opcionais: Valores default no template	116
9.7 Componentes customizados com Facelets	116
9.8 Exercícios opcionais: Composite-componentes	118
10 Formulário master/detail stateful	120
10.1 Cadastro de notas fiscais	120
10.2 Exercícios: Master (NotaFiscal)	122
10.3 Detalhes da nota	124
10.4 Exercícios: Detail (Item)	126

Sumário	Caelum
10.5 O estado da tela e o @ViewScoped	128
10.6 Novidade do Java EE 7: ViewScoped com CDI	129
10.7 Exercícios: @ViewScoped	130
10.8 Exercícios opcionais: Ajax	131
10.9 Novidade do Java EE 7: Stateless View no JSF	131
11 Recursos Avançados de CDI	132
11.1 Interceptadores do CDI	132
11.2 Exercícios: Trabalhando com conexões e transações	135
11.3 Discussão: Transações demarcadas ou transação por request	136
11.4 Eventos e observers do CDI	137
11.5 Exercícios: Disparando eventos com CDI	138
11.6 Atributos produtores	139
11.7 Qualifiers do CDI	140
11.8 Injeção com @Any	141
11.9 Exercícios: Qualifiers	141
11.10 Menos anotações nas classes com Stereotypes	142
11.11 Exercícios: Stereotypes	144
11.12 Conversation Scope	144
11.13 Exercícios opcional: Escopo de conversação	148
11.14 Novidade do Java EE 7: Faces Flow	148
11.15 Para saber mais: Interceptador de Auditoria	149
11.16 Injection Points	150
11.17 Exercícios opcionais: InjectionPoint	152
12 Interfaces Web 2.0 com componentes ricos	153
12.1 Bibliotecas de componentes	153
12.2 Melhorando a tela de adição de notas fiscais	154
12.3 PrimeFaces	155
12.4 Adicionando datepicker	156
12.5 Exercícios: componente de calendário	157
12.6 Máscara com p:inputMask	158
12.7 Exercícios: máscara no campo CNPJ	158
12.8 Realizando paginação de dados	159
12.9 Exercício: Realizando a paginação dos dados	160
12.10 Paginação customizada com banco de dados	160
12.11 Exercícios: Estendendo componentes para realizar paginação no banco de dados	163
12.12 Menus e navegação	164
12.13 Exercícios: menus	165

12.14 Adicionando abas	166
12.15 Exercícios: componente de abas	166
12.16 Exercícios opcionais	168
12.17 Para saber mais: Customizando o visual dos componentes	167
12.18 Exercícios opcionais	168
12.19 Para saber mais: Geração de gráficos com PrimeFaces	168
12.20 Desafio: gráficos	170
12.21 Para saber mais: Melhorando a seleção de produtos na tela de itens com o Autocomplete	170
12.22 Exercício opcional	171
12.23 Novidade do Java EE: File Upload com JSF	171
13 Validação e conversão de dados	173
13.1 Validação	173
13.2 Validação com Bean Validation	174
13.3 Bean Validation sem servidor de aplicação	177
13.4 Exercícios: Integrando Bean Validation com o JSF	177
13.5 Validação com componentes JSF	179
13.6 Exercícios: Componentes de validação	180
13.7 Validações customizadas com JSF	180
13.8 Exercício: Criando métodos de validação	181
13.9 Evitando duplicidades de validações criando classes validadoras	182
13.10 Exercícios opcionais: @FacesValidator	183
13.11 Conversores de dados com o JSF	183
13.12 Immediate	184
13.13 Submitted value	185
13.14 Exercícios: immediate	186
13.15 Conversor de produto	187
13.16 Exercícios opcionais	188
14 Apêndice - Um pouco mais sobre o JSF	190
14.1 Lidando com requisições GET no JSF	190
14.2 Novidade do Java EE 7: ViewAction do JSF	191
14.3 Exercícios: f:metadata e f:viewAction	191
14.4 Disparando requisições do tipo GET através de links JSF	192
14.5 Exercícios: Requisições do tipo GET	193
14.6 Binding de componentes do JSF	193
14.7 Exercícios: Binding	195
14.8 Trabalhando com staging da aplicação	196
14.9 Exercícios: Alterando o stage da nossa aplicação	197

14.10 Validação de múltiplos campos com JSF	198
14.11 Exercícios opcionais: Validação de múltiplos campos	200
14.12 Para saber mais: Conversores customizados	201
14.13 Exercícios opcional: Conversores personalizados	203
15 Apêndice - Internacionalização: sua aplicação em várias línguas	205
15.1 Formas de internacionalizar a aplicação	205
15.2 Utilizando o JSF para internacionalizar	206
15.3 Exercícios: Começando a internacionalização e localização da aplicação	207
15.4 Alterando o idioma manualmente	208
15.5 Exercícios: Definindo o idioma através de links	210
15.6 Internacionizando mensagens de erro do Bean Validation	210
15.7 Exercícios: Internacionalizando mensagens de erro	211
15.8 Internacionalizando as mensagens dentro dos Managed Beans	211
15.9 Exercícios: Internacionalizando as mensagens do Managed Beans	213
15.10 Exercício Opcional: Internacionalizando a aplicação por completo	213

Versão: 19.9.10

LABORATÓRIO WEB COM JSF E CDI

"Eu respeito o poder eterno, não cabe a mim determinar seus limites, eu não afirmo nada, contento-me em crer que sejam possíveis mais coisas do que nós pensamos." -- Voltaire, Micromegas, considerado o primeiro conto de ficção científica da história

1.1 CONSTRUA APLICAÇÕES WEB RICAS DE FORMA FÁCIL

Atualmente, o principal tipo de aplicações desenvolvidas e mantidas são web. Isso faz com que exista uma grande oferta de ferramentas no mercado, justamente o caso da plataforma Java, onde temos diversas opções de frameworks para construir essas aplicações.

Alguns frameworks como o Struts, Spring MVC e o VRaptor trabalham de uma maneira conhecida como *Action Based* (Baseado em Ações). Em contrapartida, uma outra série de frameworks, conhecidos como *Component Based* (Baseado em Componentes), vem destacando-se pela facilidade de criar **aplicações web** com recursos visuais complexos de forma simples. Dentre esses frameworks, se destacam o JavaServer Faces (JSF) que será objeto de estudo neste curso.

Durante o curso Laboratório Web com JSF e CDI (FJ-26) o aluno construirá uma aplicação completa, com funcionalidades que vão desde um simples CRUD, onde será trabalhado um domínio de notas fiscais, passando por processos de autenticação e autorização, formulários de cadastro master-detail, internacionalização da aplicação, criação de templates dentre outras tarefas.

Para o gerenciamento de componentes da infraestrutura usaremos o CDI (*Contexts and Dependency Injection*) que foi fortemente influenciado pelo JBoss Seam e Google Guice, outros dois frameworks famosos. Tudo isso rodando dentro do **servidor de aplicações** (Application Server) Glassfish, que vem destacando-se no mercado principalmente por sua leveza e rapidez.

Outro ponto importante ao se trabalhar com JSF é a integração com CSS (*Cascading Style Sheets*) para criar um layout elegante e visualmente agradável. Essa integração também será vista durante o curso, sem requerer conhecimentos prévios em CSS do aluno.

Um dos objetivos do curso é fazer com que o aluno vivencie o ambiente de desenvolvimento de uma aplicação web, simulando problemas e requisitos enfrentados no dia a dia de uma equipe, dessa forma, o conhecimento adquirido poderá ser utilizado imediatamente após o término do curso já no mercado de trabalho.

Já conhece os cursos online Alura?



A **Alura** oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Programação, Front-end, Mobile, Design & UX, Infra e Business, com um plano que dá acesso a todos os cursos. Ex aluno da Caelum tem 15% de desconto neste link!

[Conheça os cursos online Alura.](#)

1.2 INTEGRAÇÃO COM TECNOLOGIAS DO JAVA EE

O JSF é uma especificação do Java EE (JSR 344) que pode integrar-se com diversas outras especificações, como a JPA (*Java Persistence API* / JSR 338), por exemplo. Nesse curso, estudaremos a fundo a integração do JSF com a especificação CDI (JSR 299), fazendo com que nossos códigos sigam boas práticas da orientação a objetos, como por exemplo o baixo acoplamento.

Além disso, estudaremos formas de desacoplar nossa aplicação do controle das transações e da abertura de conexões com o banco de dados.

1.3 TIRANDO DÚVIDAS

Para tirar dúvidas dos exercícios, ou de Java em geral, recomendamos o fórum do site do GUJ (<http://www.guj.com.br/>), onde suas dúvidas serão respondidas prontamente.

Além disso, sinta-se à vontade para entrar em contato com seu instrutor e tirar todas as dúvidas que tiver durante o curso.

1.4 PARA ONDE IR DEPOIS?

O curso de **Persistência com JPA, Hibernate e EJB lite (FJ-25)** é uma excelente continuação para o FJ-26, abordando a persistência de dados através da EJB lite, JPA e Hibernate rodando dentro do *Application Server JBoss 7* com vários detalhes e tópicos avançados.

O curso de **Arquitetura e Design de Projetos Java (FJ-91)**, vai abordar inúmeras escolhas e decisões que podemos tomar durante o projeto de criação de uma aplicação, mostrando vantagens e desvantagens de cada framework, de cada opção de design, protocolo, etc.

Caso esteja interessado em livros, indicamos o livro Aplicações Java para web com JSF e JPA do Gilliard Cordeiro, publicado pela editora Casa do Código: <http://www.casadocodigo.com.br/>

Saber inglês é muito importante em TI

galandra O Galandra auxilia a prática de inglês através de *flash cards* e *spaced repetition learning*. Conheça e aproveite os preços especiais.

[Pratique seu inglês no Galandra.](#)

DEFINIÇÃO DO PROJETO

"O verdadeiro significado das coisas se encontra na capacidade de dizer as mesmas coisas com outras palavras." -- Charlie Chaplin

Ao término desse capítulo, você será capaz de:

- Descrever o projeto que será desenvolvido durante o curso;
- Determinar as tecnologias que utilizaremos.

2.1 DESCRIÇÃO DO PROBLEMA

Uma distribuidora de bebidas da cidade de Uberlândia com diversos clientes por todo o Brasil está precisando informatizar algumas tarefas. A distribuidora, chamada *UberDist*, está com sérios problemas na organização das notas fiscais relacionadas às vendas que são feitas dos seus produtos. E, para resolver esse problema, ela teve a ideia de informatizar o cadastro dessas notas fiscais para que facilitasse futuras consultas delas e também o controle sobre as vendas.

Um dos pedidos da *UberDist* é que o sistema fosse web, pois ele precisaria consultar as notas fiscais de fora da empresa e também como uma forma de garantir que todos os usuários utilizarão sempre a versão mais atual da aplicação, o que em uma aplicação web é sempre garantido. Além disso, uma outra característica dessa aplicação é que ela deve possuir uma interface rica, onde a usabilidade seja bem parecida com uma aplicação para *Desktop*.

Seus livros de tecnologia parecem do século passado?



Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**. Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

[Casa do Código, Livros de Tecnologia.](#)

2.2 JAVA EE WEB PROFILE

Para desenvolvermos o projeto da *UberDist*, decidimos utilizar **JSF** em conjunto com **CDI**, e **JPA** para nos auxiliar na persistência dos dados. Como gerenciador de banco de dados, onde armazenaremos as informações, utilizaremos o **MySQL**.

Tanto JSF, quanto CDI e JPA são especificações do Java EE. Sendo assim, será preciso usar algumas implementações desta plataforma para rodar a aplicação. Podemos até utilizar um *Servlet Container* como base, para depois configurar e integrar todas as implementações das especificações manualmente. Assim seria possível usar, por exemplo, *Apache Tomcat* como *Servlet Container*, *Mojarra* como implementação JSF, *Weld* para o CDI e o *Hibernate* para o JPA. Além de ser bem trabalhoso fazer isso, ficaria sob total responsabilidade do desenvolvedor integrar corretamente todas essas bibliotecas e suas possíveis dependências.

Para facilitar, há servidores compatíveis com o padrão Java EE, já pré configurados e prontos para serem usados. A ideia é fornecer uma infraestrutura adequada e pronta, para que o desenvolvedor não preocupe-se com detalhes de configuração. O Java EE cresceu bastante e os servidores que implementam este padrão já trazem uma série de bibliotecas para resolver diversos problemas. Como consequência, os servidores ficaram cada vez mais pesados e difíceis de usar, principalmente para quem precisa apenas das especificações voltadas para a Web.

Em uma tentativa de simplificar o Java EE, foi definido um **perfil específico** para quem precisar trabalhar especificamente com aplicações web. Esse perfil foi chamado de **Web Profile**. O *Web Profile* implementa um conjunto de especificações que a grande maioria das aplicações web precisa. Assim, o desenvolvedor pode usufruir de servidores mais leves, e quando realmente precisar de outras tecnologias, fica fácil migrar para um profile mais completo.

2.3 SERVIDOR DE APLICAÇÃO GLASSFISH

Utilizaremos no decorrer do curso o *Servidor de aplicação Glassfish* com o *Web Profile*. No entanto, é importante salientar que há poucas configurações específicas para o Glassfish. Dessa forma, poderíamos utilizar qualquer outro *servidor de aplicação* completo (ou *Web Profile*) como por exemplo o *JBoss (Wildfly)*, *Apache TomEE*, *Oracle WebLogic*, *IBM WebSphere*, dentre outros.

PERSISTÊNCIA COM JPA, HIBERNATE E EJB LITE

A Caelum possui o curso Persistência com JPA, Hibernate e EJB lite (FJ-25), onde é abordada com muito detalhe a JPA com o Hibernate e EJB lite usando o servidor de aplicação JBoss AS 7. O treinamento foca na persistência, inclusive fazendo otimizações e diversas configurações e customizações que são possíveis de serem realizadas.

2.4 MODELO

Conversando com os responsáveis pelo domínio da aplicação, percebemos que será preciso manter um cadastro dos produtos da empresa, onde eles precisarão saber o nome, a descrição e o preço do produto. Com relação às notas fiscais, precisarão do CNPJ do cliente e a data da emissão da nota. Cada nota poderá possuir diversos produtos, que são conhecidos como itens da nota. Cada item da nota deverá identificar a qual produto se refere e qual o seu valor unitário no dia da emissão da nota.

Podemos traduzir esse problema com o seguinte diagrama de classes que representa o modelo da nossa aplicação:



Para implementarmos as classes desse modelo, poderíamos definir um pacote padrão como `br.com.caelum.notasfiscais.modelo`. Utilizando a JPA, teríamos algo como o seguinte:

```
@Entity
public class Produto {

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

    private String nome;

    private String descricao;

    private Double preco;

    //getters / setters
}

@Entity
public class Item {

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

```

    @ManyToOne
    private NotaFiscal notaFiscal;

    @ManyToOne
    private Produto produto;

    private Integer quantidade;

    private Double valorUnitario;

    //getters / setters
}

@Entity
public class NotaFiscal {

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

    private String cnpj;

    @Temporal(TemporalType.DATE)
    private Calendar data;

    @OneToMany(cascade=CascadeType.PERSIST, mappedBy="notaFiscal")
    private List<Item> itens;

    //getters / setters
}

```

Agora é a melhor hora de respirar mais tecnologia!



Se você está gostando dessa apostila, certamente vai aproveitar os **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum. Programação, Mobile, Design, Infra, Front-End e Business! Ex-aluno da Caelum tem 15% de desconto, siga o link!

[Conheça a Alura Cursos Online.](#)

2.5 IMPLEMENTANDO O DAO

Para encapsularmos toda a API da JPA para a persistência dos dados, usaremos o padrão DAO (*Data Access Object*). Para cada entidade teremos um DAO específico que possuirá os métodos de persistência e de pesquisa. Além dos DAOs, criaremos uma classe que nos fornecerá as instâncias do EntityManager para podermos utilizar a JPA. As implementações estão a seguir:

```

public class JPAUtil {
    private static EntityManagerFactory emf =

```

```

        Persistence.createEntityManagerFactory("notas");

    public EntityManager getEntityManager() {
        return emf.createEntityManager();
    }
}

public class ProdutoDao {

    public void adiciona(Produto produto) {
        EntityManager manager = new JPAUtil().getEntityManager();
        manager.getTransaction().begin();

        //persiste o objeto
        manager.persist(produto);

        manager.getTransaction().commit();
        manager.close();
    }

    public void remove(Produto produto) {
        EntityManager manager = new JPAUtil().getEntityManager();
        manager.getTransaction().begin();

        manager.remove(manager.merge(produto));

        manager.getTransaction().commit();
        manager.close();
    }

    public void atualiza(Produto produto) {
        EntityManager manager = new JPAUtil().getEntityManager();
        manager.getTransaction().begin();

        manager.merge(produto);

        manager.getTransaction().commit();
        manager.close();
    }

    public List<Produto> listaTodos() {
        EntityManager manager = new JPAUtil().getEntityManager();

        CriteriaQuery<Produto> query = manager.getCriteriaBuilder()
            .createQuery(Produto.class);
        query.select(query.from(Produto.class));

        List<Produto> lista = manager.createQuery(query).getResultList();

        manager.close();

        return lista;
    }

    public Produto buscaPorId(Long id) {
        EntityManager manager = new JPAUtil().getEntityManager();

        Produto produto = manager.find(Produto.class, id);

        manager.close();

        return produto;
    }
}

```

```
    }  
}
```

TRATAMENTO DE TRANSAÇÕES E CICLO DE VIDA DOS ENTITY MANAGERS

É importante perceber que o tratamento de transações, exceções e abertura/ fechamento dos Entity Managers são pontos cruciais para a escalabilidade e performance da sua aplicação. Por enquanto, faremos dessa maneira simples, abrindo e fechando managers e transações sempre que necessário, mas nos capítulos posteriores veremos como fazer isso de uma maneira mais elegante e escalável.

Por fim, precisaremos do arquivo de configurações da JPA para indicarmos o endereço do banco de dados, e as informações para autenticar-se no banco. Esse arquivo é o `persistence.xml`, que deve ser criado no diretório `META-INF` no classpath da sua aplicação (`src`). Nele também ficarão os dados específicos da implementação/ provedor da JPA. Como utilizaremos o Glassfish, o provedor de persistência (*persistence provider*) será o **EclipseLink**, a implementação referencial da JPA:

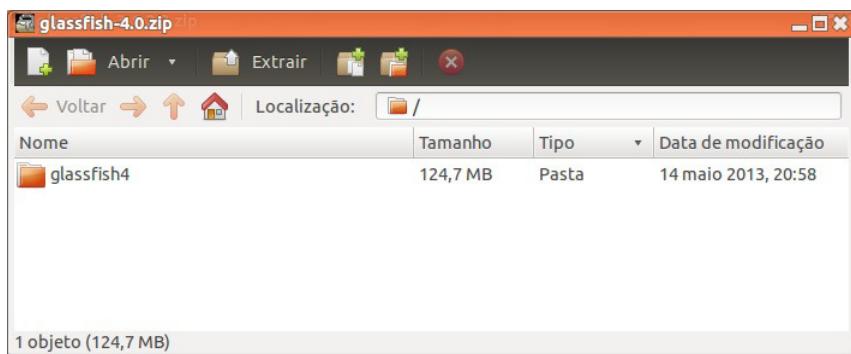
```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"  
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence  
        http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"  
    version="2.0">  
  
<persistence-unit name="notas" transaction-type="RESOURCE_LOCAL">  
  
    <class>br.com.caelum.notasfiscais.modelo.Item</class>  
    <class>br.com.caelum.notasfiscais.modelo.NotaFiscal</class>  
    <class>br.com.caelum.notasfiscais.modelo.Produto</class>  
    <class>br.com.caelum.notasfiscais.modelo.Usuario</class>  
  
    <properties>  
        <property name="javax.persistence.jdbc.driver"  
            value="com.mysql.jdbc.Driver"/>  
        <property name="javax.persistence.jdbc.url"  
            value="jdbc:mysql://localhost/fj26"/>  
        <property name="javax.persistence.jdbc.user" value="root"/>  
        <property name="javax.persistence.jdbc.password" value="" />  
  
        <!-- EclipseLink atualiza as tabelas do banco automaticamente -->  
        <property name="eclipselink.ddl-generation" value="create-tables" />  
        <property name="eclipselink.ddl-generation.output-mode"  
            value="database" />  
    </properties>  
    </persistence-unit>  
</persistence>
```

2.6 EXERCÍCIOS: INSTALANDO O GLASSFISH E CRIANDO O PROJETO

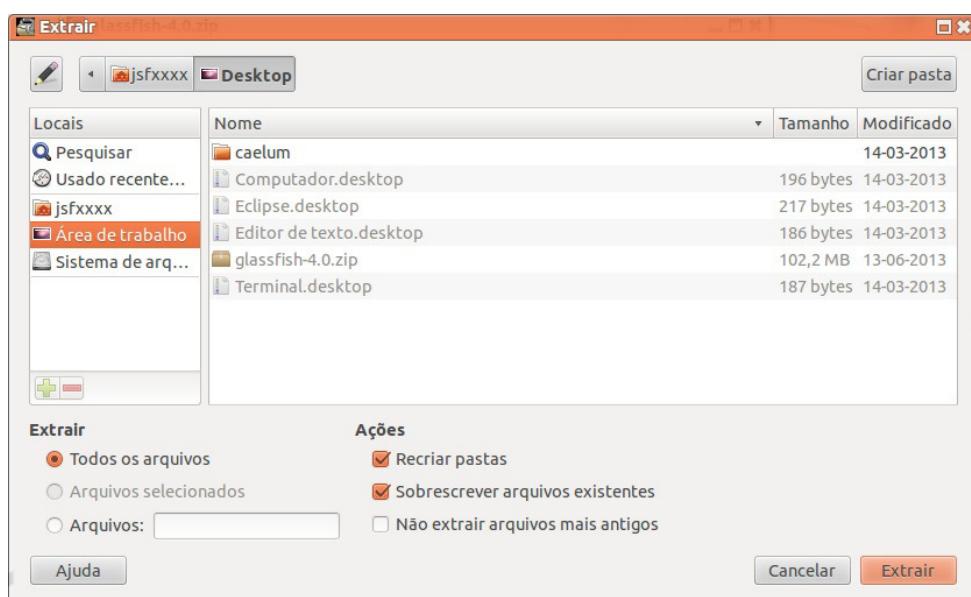
Todos os arquivos utilizados nos exercícios desta apostila estão disponíveis em:
<https://goo.gl/v1i4X1>.

1. Primeiramente, precisamos instalar o Glassfish em nossas máquinas.

- Vá no Desktop e entre na pasta *Caelum* e em seguida na pasta 26.
- Descompacte o arquivo `glassfish-4.x.zip` no seu Desktop. Para isso dê um duplo clique no arquivo para abrir o *Archive Manager* em seguida clique no botão *Extract*.



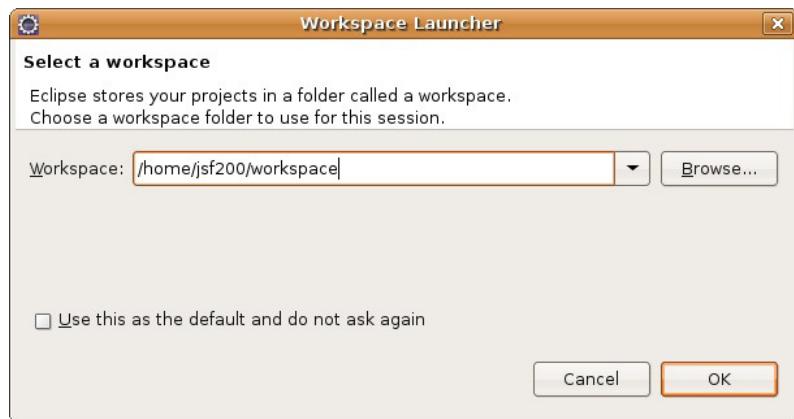
- Na janela que abriu, selecione no lado esquerdo o item *Área de Trabalho* (ou *Desktop*) e clique em *Extract*.



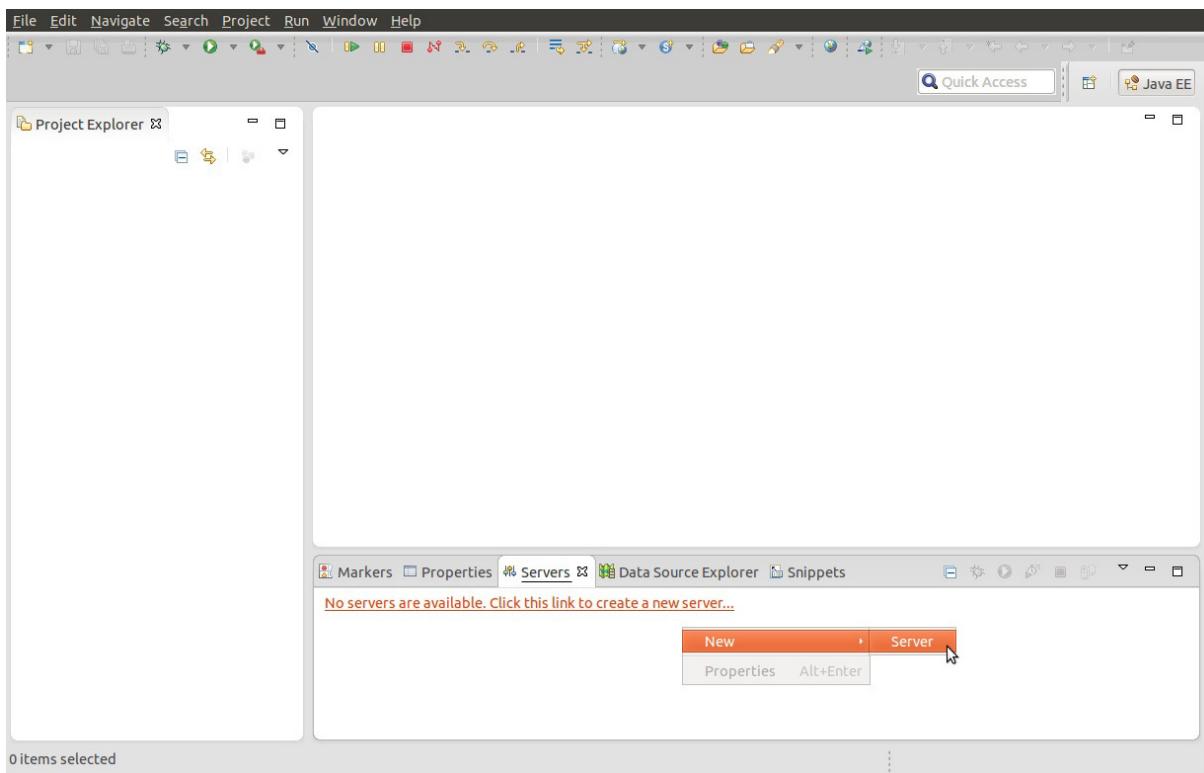
- Falta copiar o JAR do MySQL para a pasta do *Glassfish*, na pasta *Caelum/26*, copie o JAR do MySQL e cole no diretório `_glassfish4/glassfish/domains/domain1/lib/ext_`
- Pronto, você já instalou o *Glassfish*.

2. O próximo passo é configurar o Glassfish no Eclipse, para que possamos controlá-lo facilmente:

- Abra o Eclipse, e ao ser perguntado sobre o caminho da sua workspace apenas confirme.

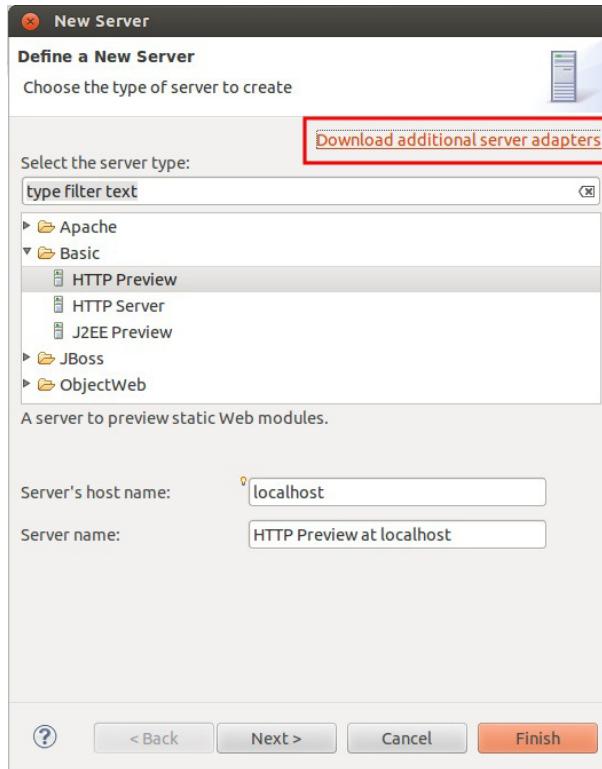


- Se uma janela de boas vindas for exibida, feche-a ou clique no link *Workbench* exibido nela.
- Dentro do Eclipse, abra a aba *Servers*. Para isso, vá no menu *Window* -> *Show View* e escolha a opção *Servers* e ela será aberta na parte inferior do seu Eclipse.
- Dentro da aba *Servers* clique com o botão direito do mouse e escolha *New* -> *Server* ou, caso seja o primeiro servidor configurado, clique direto no link *Click this link to create a new server....*

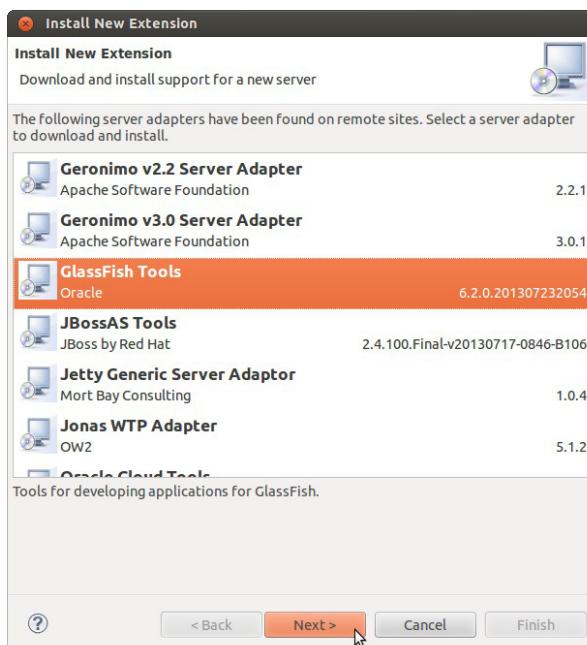


- Na janela *Define a New Server*, caso o *adapter* do **GlassFish 4.0** já esteja na lista, pule para o passo **i** e continue com a configuração. Caso contrário, clique no link *Download additional server*

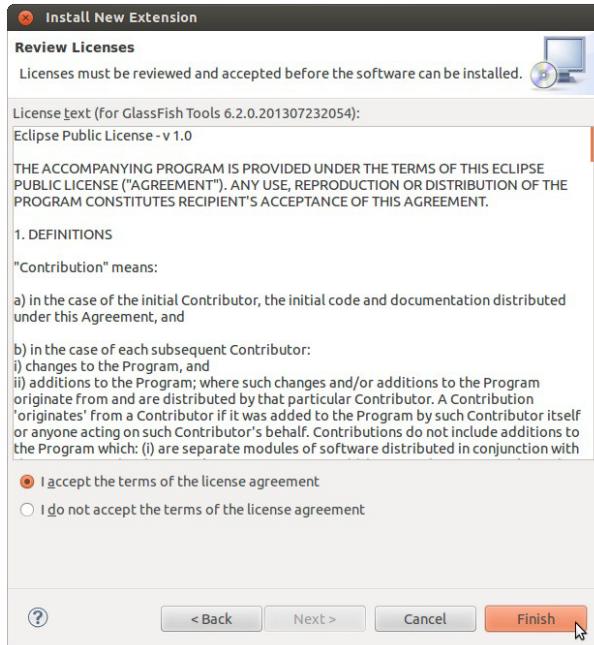
adapters:



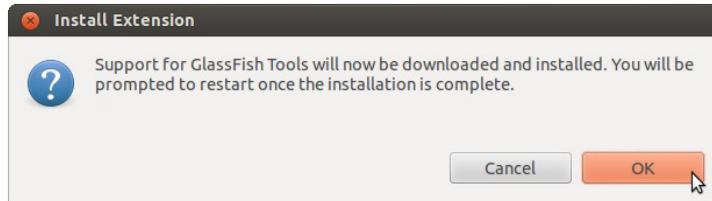
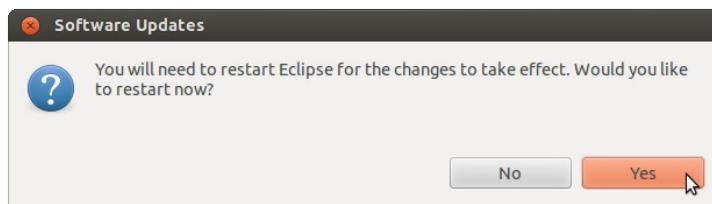
- Na janela *Install New Extension*, selecione o *adapter GlassFish Tools* e clique em *Next >*:



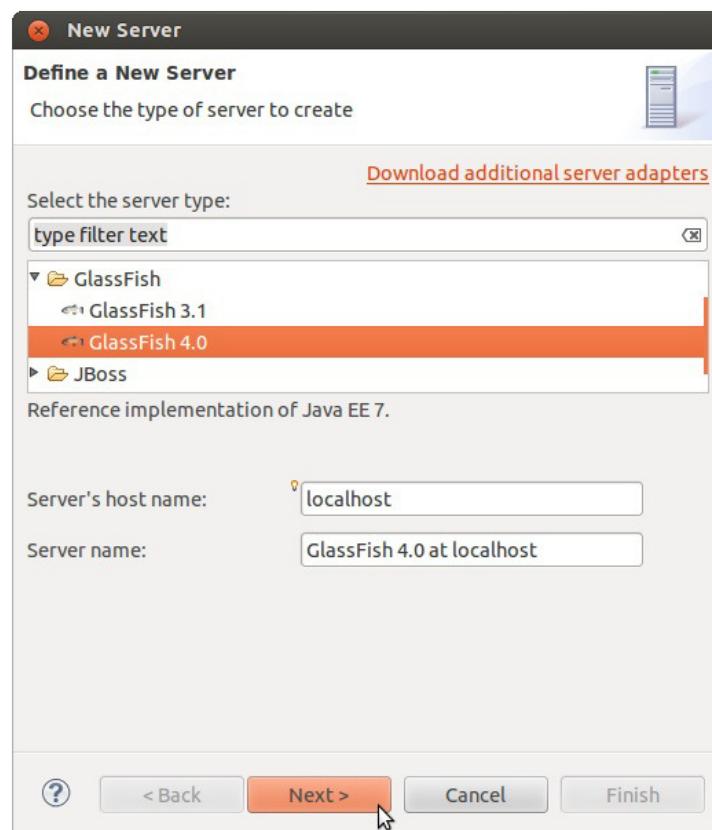
- Aceite os termos da licença e clique em *Finish*.



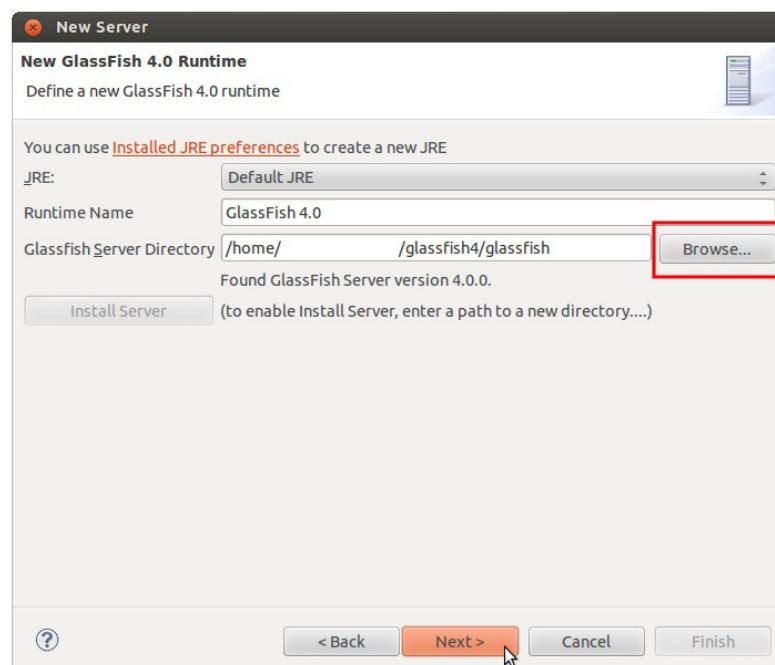
Dê **OK** nas duas telas seguintes que são de confirmações do download e instalação do adapter, e de *restart* do Eclipse.



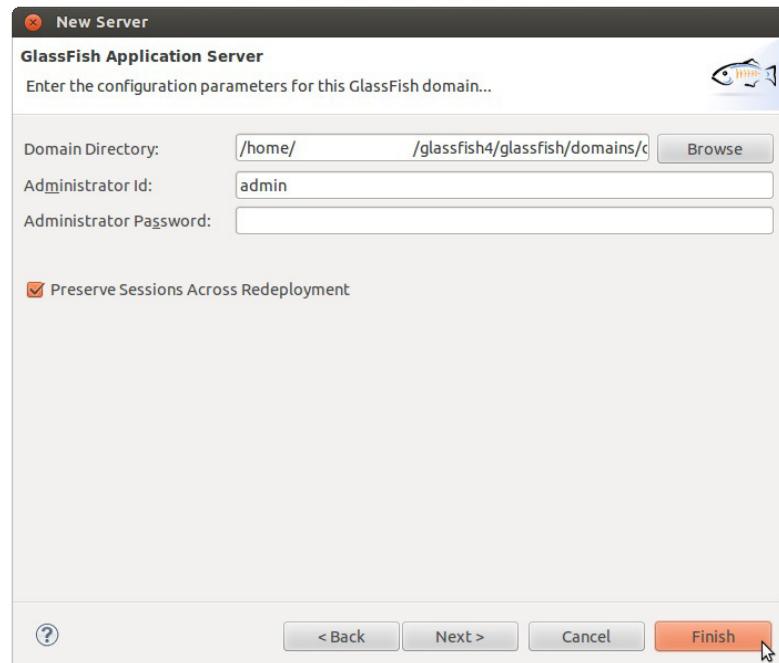
- Depois do *restart* do Eclipse, volte na aba *Servers* e reinicie o processo de adição de um novo servidor.
- Dentro da Janela *New Server* escolha *Glassfish 4.0* e clique em *Next*.



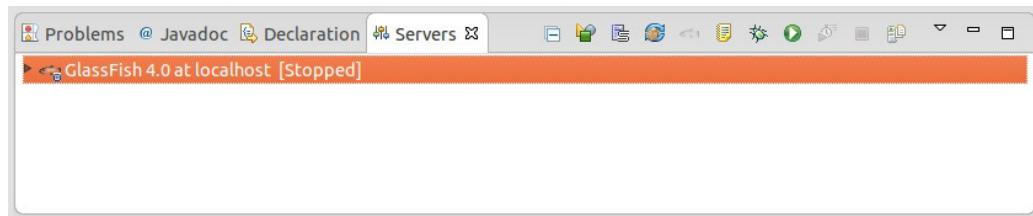
- No próximo passo dizemos ao Eclipse em qual diretório instalamos o Glassfish 4.0. Clique no botão *Browse...* e escolha a pasta **glassfish** que estará dentro do diretório na qual você descompactou o *Glassfish4*. Daí, clique em *Next*.



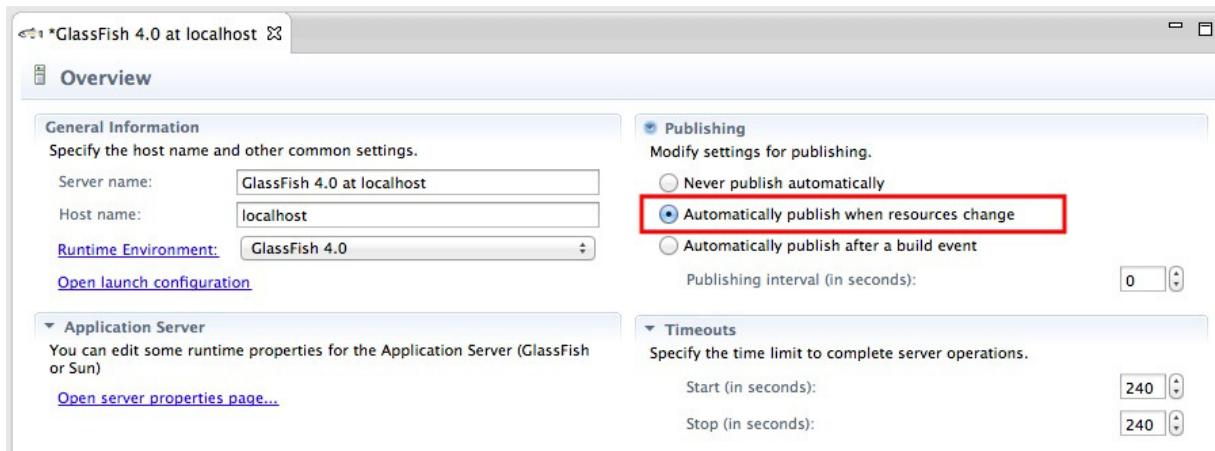
- Agora deixaremos a configuração padrão, ou seja, o *Administrator Id* como **admin** e o *Administrator Password* em branco. Logo após clique em *Finish*.



- Agora o **GlassFish 4.0** já aparece na aba *Servers* no estado **[Stopped]** .

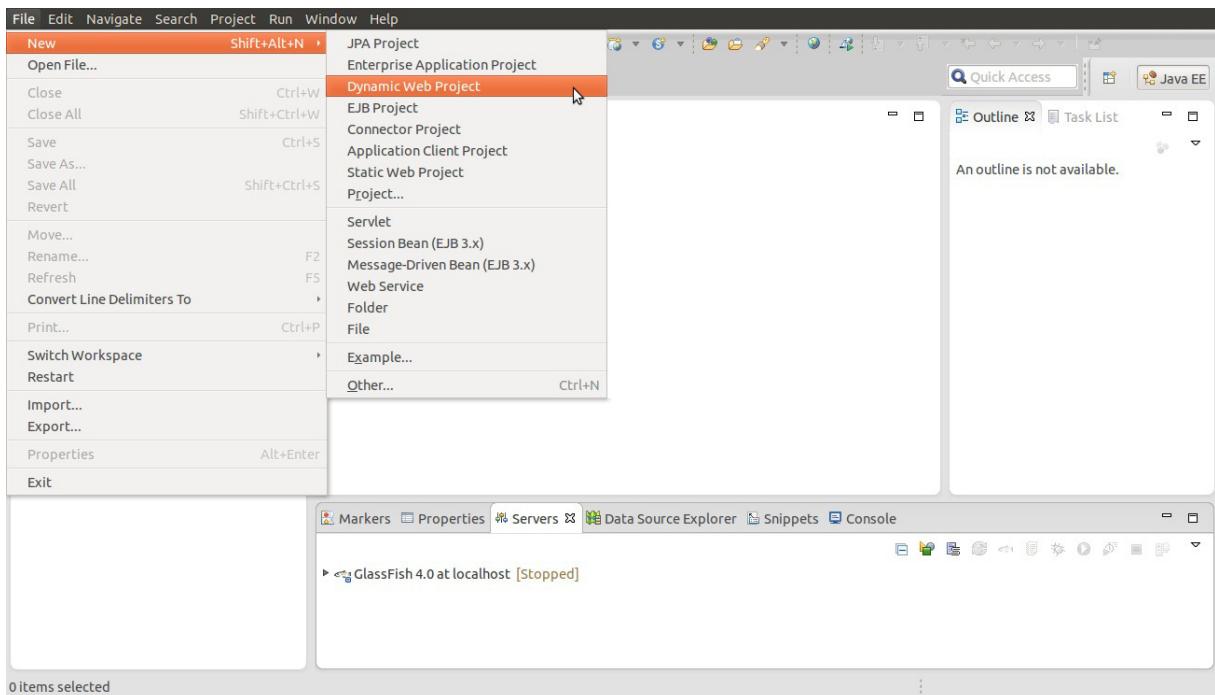


- Falta configurar o **Hot Deploy**. Dê um duplo clique no nome do servidor. Na janela que será aberta, abra a opção *Publishing* e escolha **Automatically publish when resources change**:

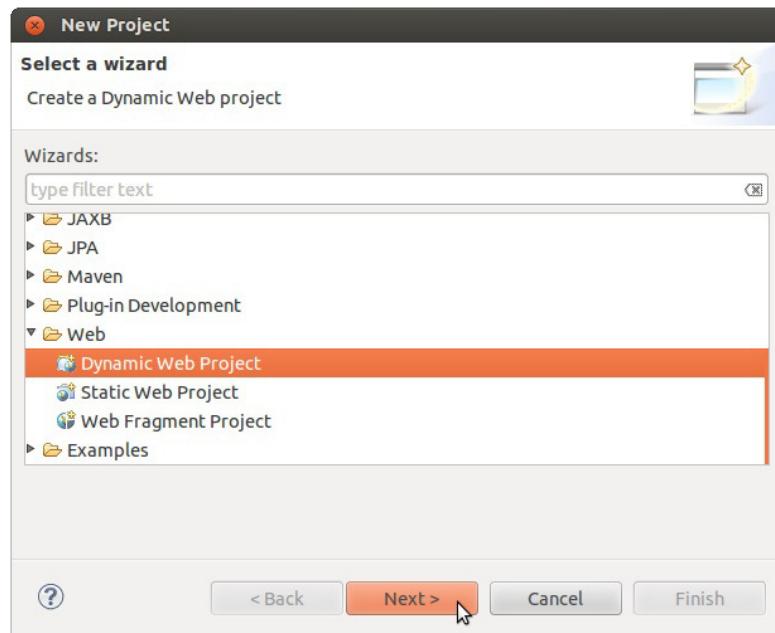


3. Agora, para podermos começar a trabalhar, criaremos o nosso projeto dentro do Eclipse.

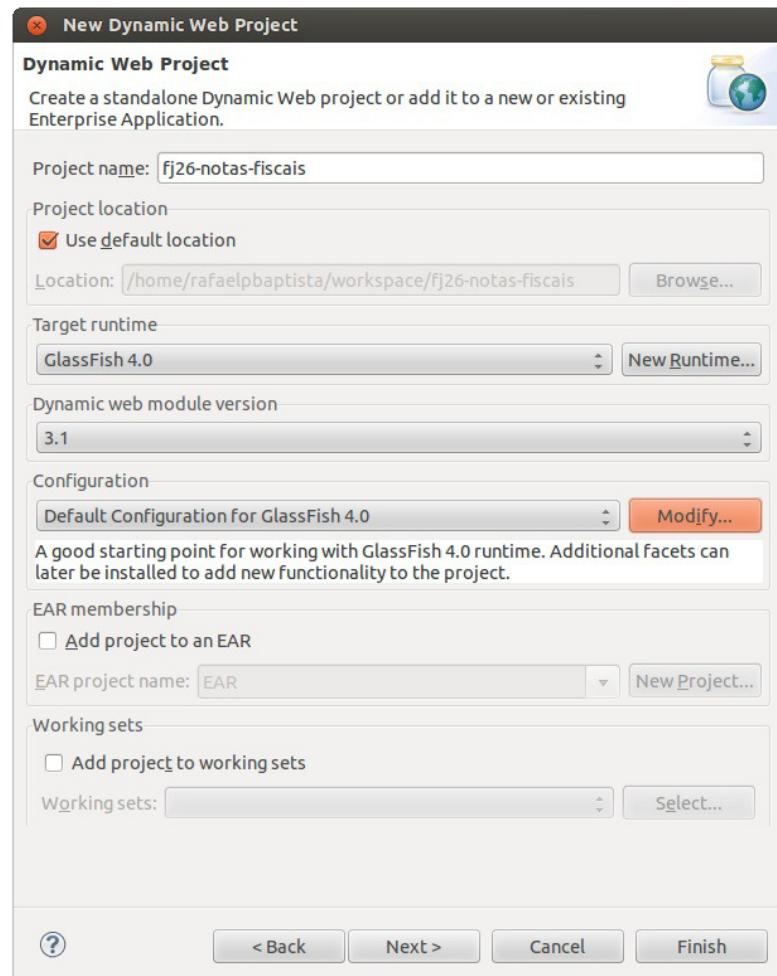
- Já no Eclipse, perspectiva Java EE, vá no menu *File* -> *New* -> *Dynamic Web Project*



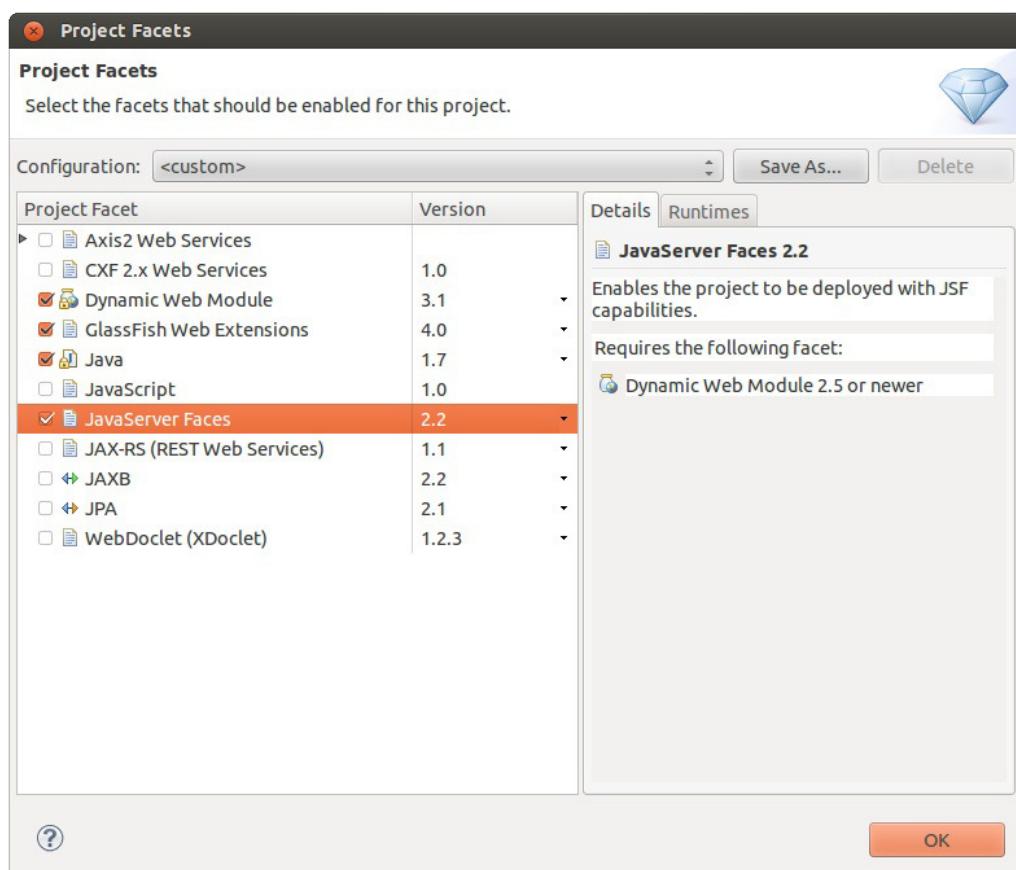
Caso esteja usando a perspectiva Java, vá no menu *File* -> *New* -> *Other*. Na janela seguinte, escolha então *Dynamic Web Project*



- Em *Project name* coloque **fj26-notas-fiscais** .
- Na seção *Configuration*, clique no botão **Modify...:**

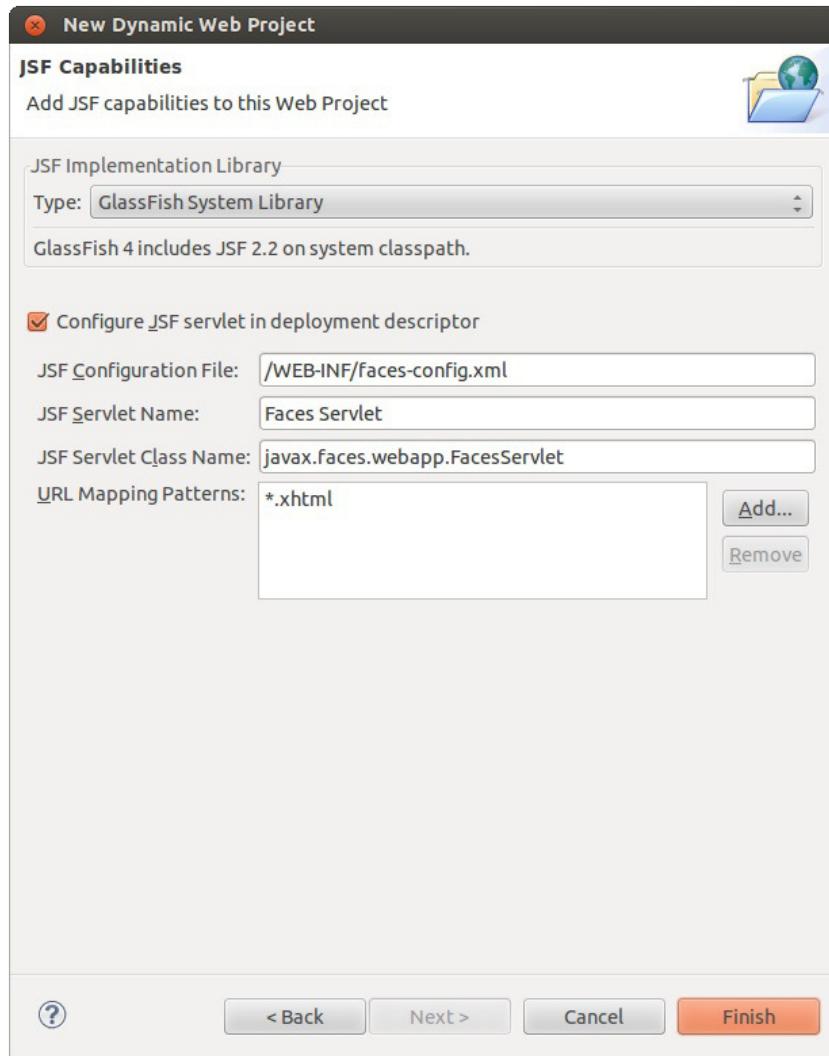


- Abrirá a janela *Project Facets* onde você deverá marcar o box do *JavaServer Faces*, versão 2.2 e clicar *Ok*:



- De volta à tela de criação do projeto, clique em **Next** duas vezes e marque a opção **Generate web.xml deployment descriptor** para que o Eclipse gere o arquivo `web.xml`.
- Clique em **Next** para chegar à tela de configuração do JSF. Nessa tela, deixe selecionada a opção **GlassFish System Library** (isso indica para o Eclipse que utilizaremos os JARs do JSF incluídos no próprio servidor *GlassFish*).

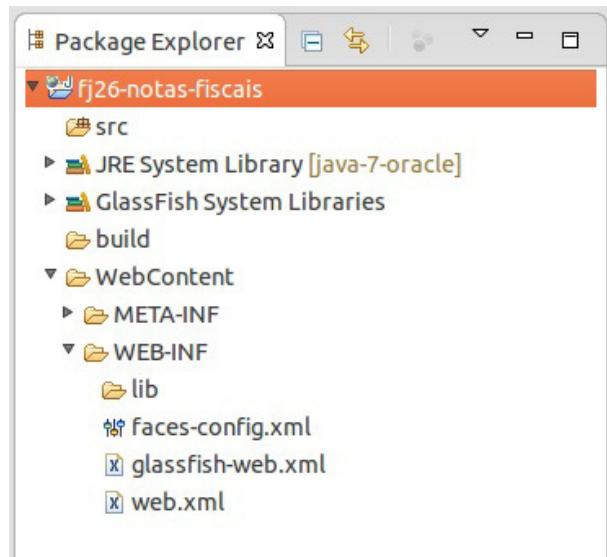
Ainda nessa tela, na parte *URL Mapping Patterns*, **remova** o mapeamento `/faces/*` (se existir) e **adicone** um novo mapeamento: `*.xhtml`



- Clique em *Finish* e o projeto está criado. Se estiver na perspectiva **Java** e for perguntado sobre a mudança para a perspectiva **Java EE**, clique em *No*.



- Assim ficará a estrutura do seu projeto:

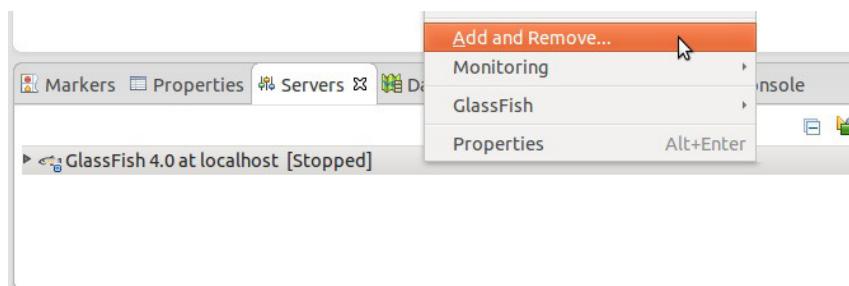


VERSÕES DO ECLIPSE

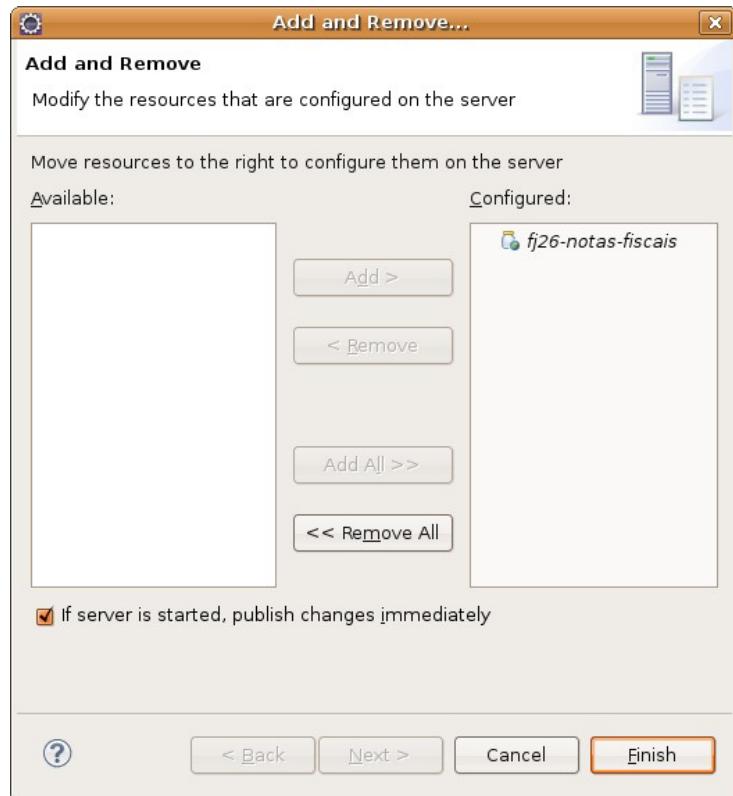
O suporte nativo a JSF 2 foi adicionado ao Eclipse na versão 3.6. Em versões anteriores, não havia um bom modo de ter autocomplete nos arquivos do facelets. Uma alternativa era o uso do plugin JBoss Tools.

4. Adicione o projeto ao servidor Glassfish configurado antes. Para isso:

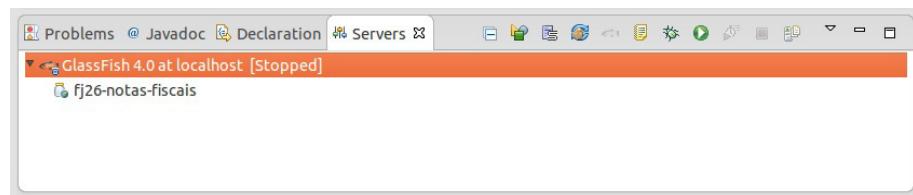
- Clique com o botão direito no servidor dentro da aba *Servers* e escolha a opção *Add and Remove...* para associarmos o nosso projeto com o *Glassfish*.



- Passe o projeto para a coluna da direita clicando no botão *Add* depois clique em *Finish*.

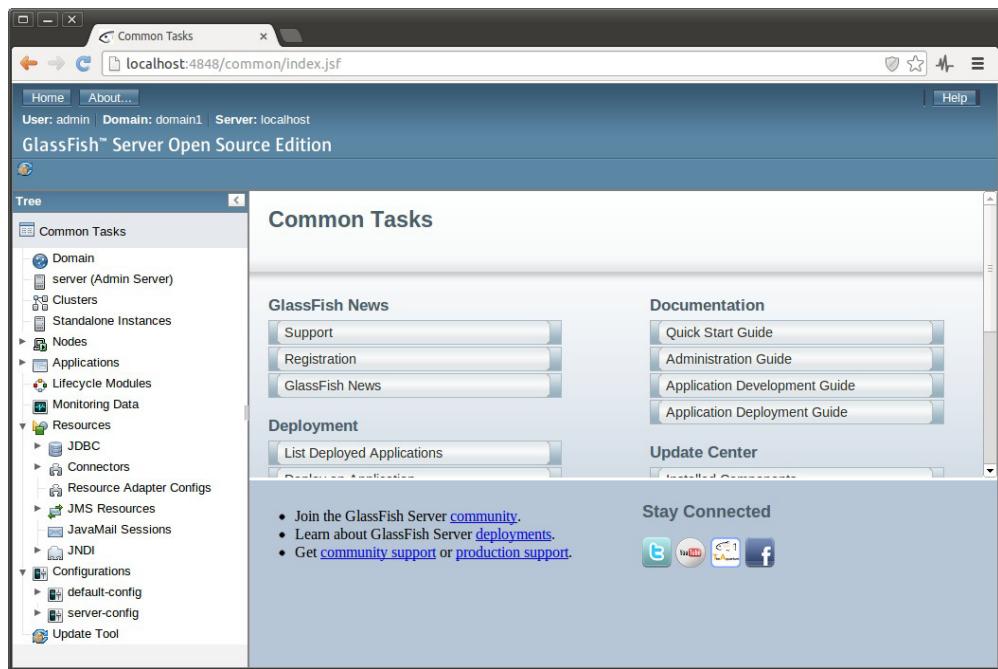


- Clicando na seta em frente ao nome do servidor é possível ver o projeto já incluído ao servidor.

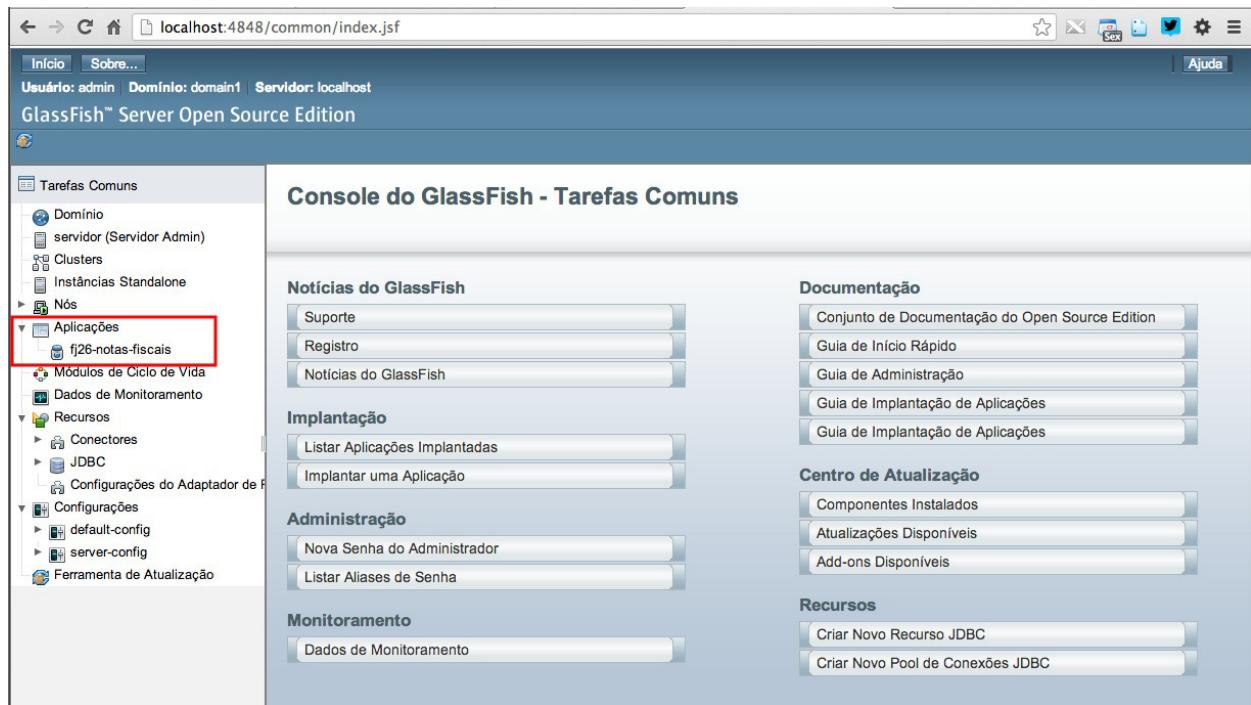


- Caso o servidor ainda esteja com status [**Stopped**], clique com o botão direito sobre o nome do servidor e selecione **Start**. Aguarde até o estado [**Started, Synchronized**].

5. Acesse `http://localhost:4848/` e veja a página de administração do Glassfish:



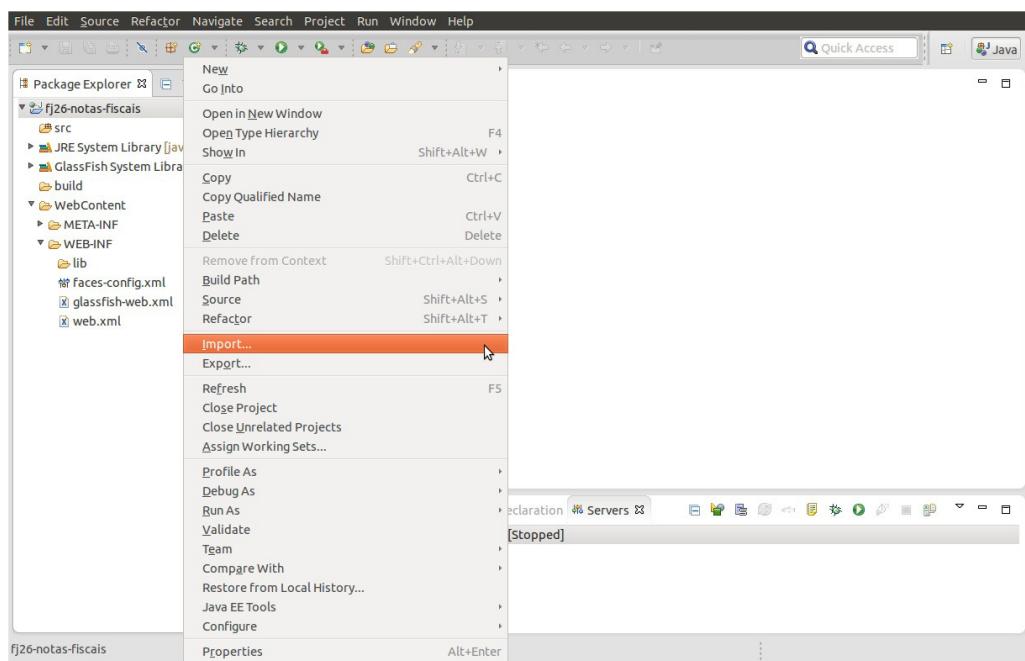
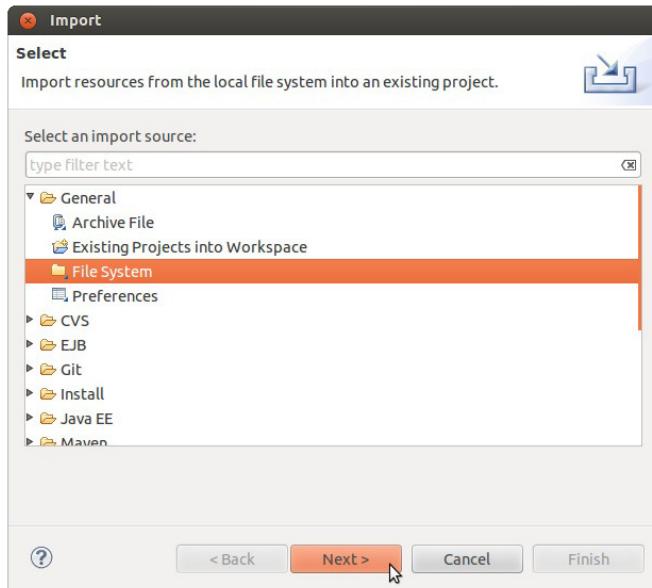
6. Na árvore de navegação do lado esquerdo da página, clicando na opção **Aplicações** você verá que nossa aplicação já foi implantada no servidor.



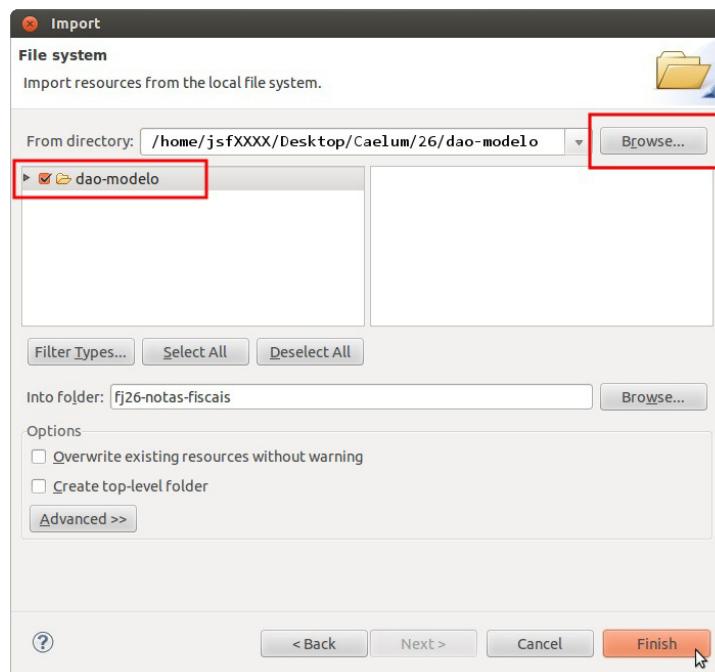
7. Por fim, precisamos das classes de modelo e do DAO. Para nos preocuparmos mais com o desenvolvimento usando JSF, e como não é necessário saber JPA para usar JSF, vamos importar as classes já digitadas:

- No Eclipse, clique com o botão direito no seu projeto e escolha a opção *Import -> General -> File*

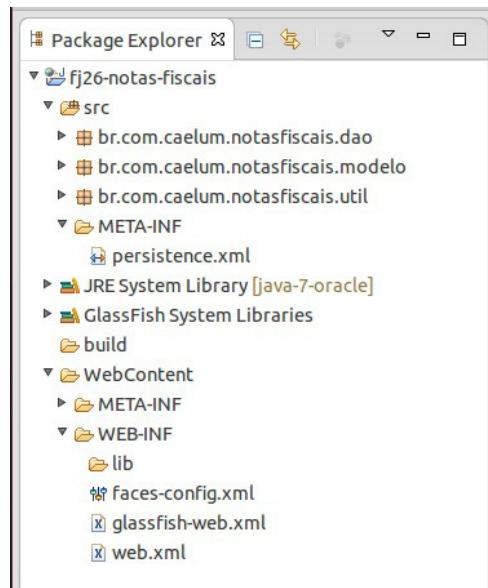
System.



- Escolha o diretório **dao-modelo**, na pasta da *Caelum*, diretório 26. Marque o checkbox para importar tudo do diretório e pressione *Finish*. **ATENÇÃO:** Tenha cuidado para não selecionar outra pasta se não **dao-modelo**.



- Após isso, seu projeto deve ter ficado com uma estrutura parecida com a seguinte:



2.7 PARA SABER MAIS: EM CASA

Para criar o projeto em casa, você precisará de:

- Eclipse IDE for Java EE Developers:** <http://www.eclipse.org/downloads>
- GlassFish Server Open Source Edition 4.x:** <https://glassfish.java.net/download.html>
- MySQL Community Server 5.x:** <http://dev.mysql.com/downloads/mysql/>

As classes iniciais do modelo você pode obter durante o curso na Caelum e importá-las no seu projeto.

Editora Casa do Código com livros de uma forma diferente



Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.

Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.](#)

JAVASERVER FACES

"Nem todos os caminhos são para todos os caminhantes" -- Johann Wolfgang von Goethe

3.1 MOTIVAÇÃO: HTTP STATELESS

Existe um grande problema com o desenvolvimento de aplicações Web: a característica stateless do protocolo HTTP.

Isso quer dizer que a cada nova requisição, a cada clicar do mouse em botões e links, o servidor já não sabe em que ponto da conversação estava com o cliente. Sem usar nenhum recurso diferente, o servidor não consegue nem mesmo saber que é o mesmo cliente que está acessando um determinado link. Aí que entram os cookies e sessões, para que pelo menos o servidor saiba com quem está falando.

Apesar de ser fácil descobrir quem é o cliente, ainda assim, a cada *refresh* de página, o estado da interface com o usuário é totalmente perdido: em um JSP você recria todo um formulário e ainda mais, precisa "repopular" todos os campos com valores que provavelmente já foram atribuídos algumas "páginas" atrás.

Repare que isso simplesmente não ocorre em uma aplicação Desktop: os valores contidos nos componentes visuais estão na memória do computador, e quando você clica em um botão, a não ser que você explicitamente apague, os dados anteriores são mantidos em seus respectivos campos. Não há necessidade de "reconstruir" a interface.

Já conhece os cursos online Alura?

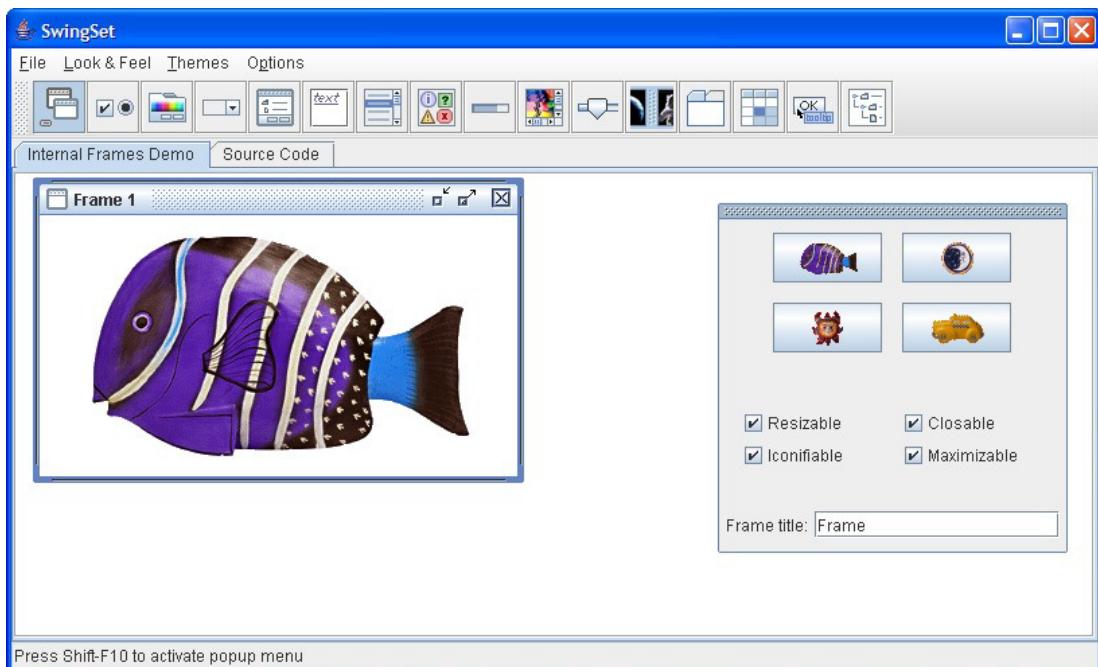


A Alura oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Programação, Front-end, Mobile, Design & UX, Infra e Business, com um plano que dá acesso a todos os cursos. Ex aluno da Caelum tem 15% de desconto neste link!

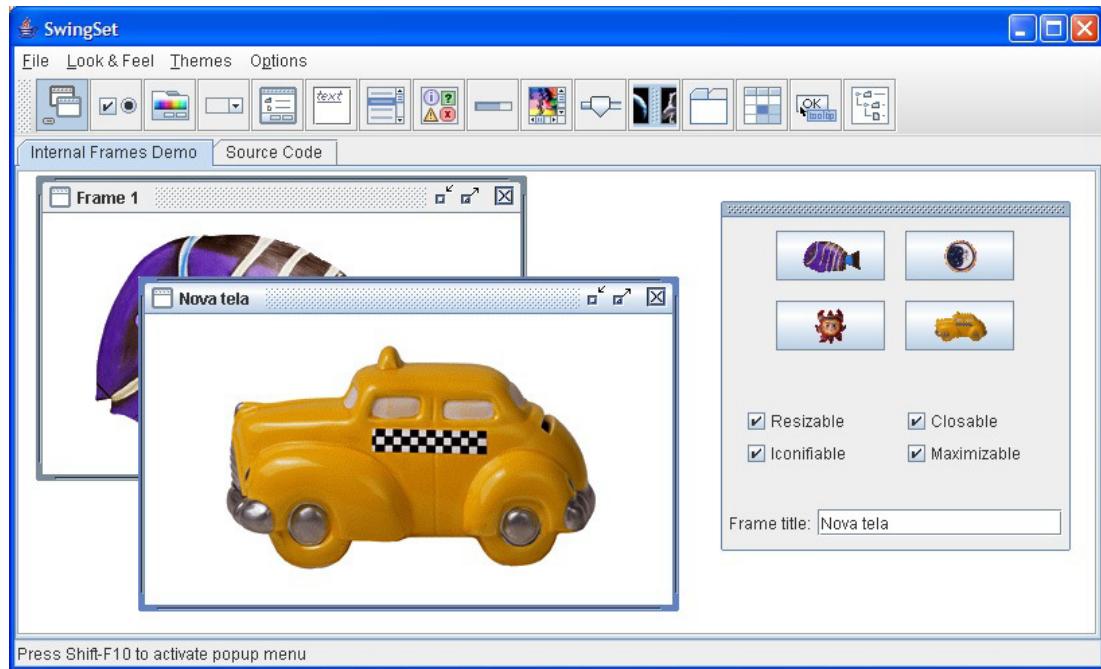
[Conheça os cursos online Alura.](#)

3.2 MOTIVAÇÃO: SWING, VISUAL BASIC E DELPHI

Considere a simples tela do demo de Swing que acompanha o SDK:



Quando você digita na caixa de texto "*Frame Title*" e clica no carrinho, o que ocorre? Um novo frame abre com o desenho do carro, como na figura abaixo:



Surpreendente? Obviamente não. Mas pare um pouco para pensar em como você faria algo parecido desenvolvendo para a web? Logo após que o usuário clicou para aparecer a imagem do carrinho:

- preciso buscar de algum lugar que na tela já havia um peixe;
- preciso buscar as informações do nome da nova tela;
- **preciso desenhar o táxi com o nome da tela escolhida;**
- preciso mudar o campo Frame Title para o nome da tela que eu criei.

Apenas a tarefa em negrito é realmente uma novidade. O resto é tudo restauração do estado da tela (*ViewState*) anterior. E, com certeza, esse é o pior trabalho.

Pense nesse outro caso: quando você cria uma tabela na web e pede para que uma linha seja removida (não importa se do banco de dados ou não). Na tela seguinte você precisa gerar novamente toda a tabela, sem aquele elemento. Vai precisar que, por exemplo, um `foreach` percorra os elementos restantes e gere cada linha da tabela.

Se fosse em Swing, Visual Basic ou Delphi, bastaria você chamar o componente e remover a linha e, caso necessário, remover o objeto correspondente do banco de dados ou da memória. Não haveria necessidade de programar para que a tabela fosse novamente construída a partir de alguns dados. Ela já está lá quando trabalhamos com uma tecnologia desktop. **Grande parte da programação na web envolve restaurar o estado de telas anteriores (*ViewState*).**

Mais ainda, em RADs de desenvolvimento como as do Delphi e Visual Basic, você pode editar as propriedades do componente, como na figura:

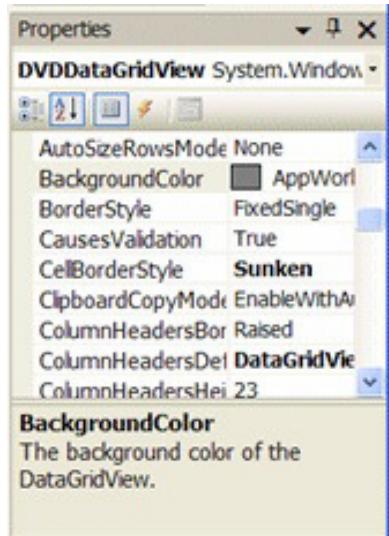


Figura 3.3: ajustando as propriedades de um componente no VB

Cada componente tem seu conjunto de propriedades, e podemos associá-lo aos dados de algum objeto que estará por trás (*Backing bean*, *data source*, etc).

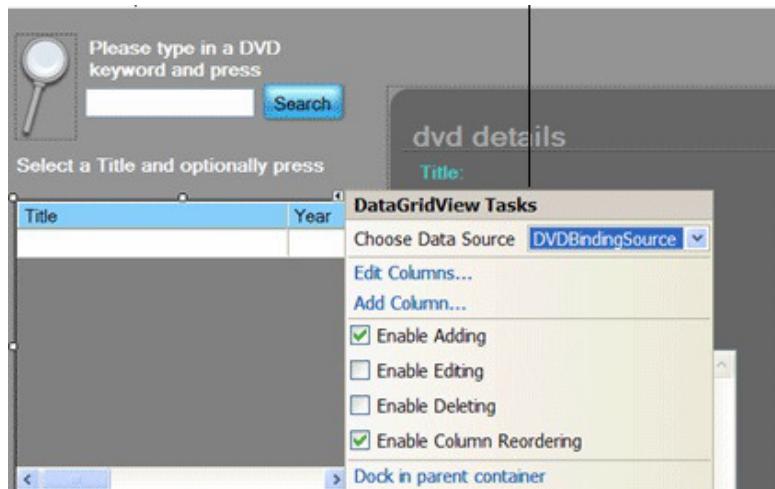


Figura 3.4: fazendo o binding do componente a fonte de dados no VB

Resumindo, toda interface com usuário é feita através da associação (binding) de valores à componentes visuais, e ajustando suas diversas propriedades.

Além disso, em uma tela, não existem `fors` e `ifs`: decidimos se um campo vai aparecer ou não por uma propriedade "visible" do componente, por exemplo.

3.3 JAVASERVER FACES (JSF)

JSF é uma tecnologia que nos permite criar aplicações Java para Web utilizando componentes visuais, fazendo com que o desenvolvedor não se preocupe com Javascript e HTML. Basta adicionarmos

os componentes (calendários, tabelas, formulários, etc) que os mesmos serão renderizados e exibidos em formato HTML. Além disso o estado dos componentes será sempre guardado automaticamente (como veremos mais a frente), fazendo o papel de *Stateful*. Isso nos permite, por exemplo, criar *wizards* e navegar nos vários passos dessa *wizard* com o estado das telas sendo mantidos, sem ter de resgatá-los novamente.

Outra característica marcante na arquitetura do JSF é a separação que fazemos entre as camadas de apresentação e de aplicação. Pensando no modelo MVC, o JSF possui uma camada de visualização que, por padrão, baseia-se nos *Facelets* e um conjunto de classes conhecidas como *Managed Beans*; veremos ambos com bastante detalhes.

Grande parte do legado que existe atualmente no mercado refere-se ao JSF 1.2. Porém, essa versão possui alguns problemas que abordaremos durante o curso, o que deixava nosso código muito acoplado ao JSF, além de alguns problemas de performance que tínhamos que resolver com nosso próprio código.

Devido a esses numerosos problemas, a comunidade Java começou a lançar soluções para os mesmos. O uso de JSP com o JSF sempre foi um problema devido a incompatibilidade de seus ciclos de vida. O *Facelets* resolveu este problema e na versão seguinte do JSF foi incorporado como camada de visualização padrão. Além disso, diversas bibliotecas adicionando suporte ao Ajax nas aplicações JSF, assim como os conceitos do JBoss Seam, criado por Gavin King, que resolvem vários problemas de código acoplado, injeção de dependências, validação, navegação também foram incorporados na nova versão. Veremos esses e muitos outros conceitos em detalhes no decorrer do curso.

O JSF 2 foi lançado em Julho de 2009 através da JSR 314. No lançamento do Java EE 6 em Dezembro de 2009, passou a ser oficial na plataforma Java EE. Já o JSF 2.2 (JSR 344), que utilizaremos durante esse curso, teve seu primeiro release estável lançado em Maio de 2013, mas só em Junho de 2013, com o lançamento oficial do JavaEE 7, foi oficializado na plataforma JavaEE. Utilizaremos a RI (Reference Implementation) que foi batizada de Projeto Mojarra e pode ser encontrada em <http://javaserverfaces.java.net/>

3.4 PRIMEIRO EXEMPLO

Agora que nossa aplicação de Controle de Notas Fiscais foi criada e preparada no capítulo anterior, vamos criar nossa primeira tela utilizando o JSF. Antes disso precisamos configurar mais alguns detalhes. O JSF é uma especificação e como foi dito, utilizaremos a implementação do Projeto Mojarra. Para isto, precisamos dos JARs da implementação Mojarra. Como utilizaremos o servidor de aplicação Glassfish, não será preciso baixar os JARs, pois os mesmos já estão embutidos no servidor.

JSF DENTRO DO SERVLET CONTAINER

Caso queiramos utilizar o JSF dentro de um *servlet container* como Apache Tomcat ou Jetty, será preciso baixar os JARs do site da implementação Mojarra:

Stable Versions of Mojarra <http://javaserverfaces.java.net/download.html>

Feito isso precisamos configurar o servlet do JSF no `web.xml` da aplicação. Esse Servlet é responsável por receber as requisições e delegá-las ao *core* do JSF. Para configurá-lo basta adicionar as seguintes configurações no `web.xml`:

```
<servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>*.xhtml</url-pattern>
</servlet-mapping>
```

Ao utilizar o Eclipse com suporte a JSF 2 para a criação do projeto, como fizemos no capítulo anterior, essa configuração no `web.xml` já é feita automaticamente.

Como utilizaremos *Facelets*, as nossas páginas estarão em arquivos com a extensão `.xhtml`. Dessa forma, criaremos um arquivo chamado `index.xhtml`.

VIEW NO JSF 1.2

Antes da versão 2 do JSF, a camada de visualização era usualmente construída com páginas JSP usando *taglibs* especiais do JSF. Com isso, todo programador acostumado com as tecnologias Web tradicionais do Java conseguia aprender e usar o JSF.

Com o tempo, começaram a surgir alternativas para a *View* com recursos mais poderosos e integrados ao JSF. O **Facelets** foi criado pela Sun, ainda na época do JSF 1.x, e logo ganhou bastante mercado. A partir do JSF 2, foi instituído como biblioteca padrão para *View* e o uso de JSP tornou-se *deprecated*.

Devido ao fato do JSF ser todo baseado em componentes, precisamos utilizá-los para manter o estado das nossas telas. Os componentes do JSF são usados como tags novas na nossa tela. O JSF já define dois conjuntos principais de tags que vamos utilizar, a ****core**** e a ****html****. A `html` possui os componentes necessários para montarmos a nossa tela gerando o HTML adequado. Enquanto a

`core` possui diversos componentes para "estender" outros componentes, como por exemplo, adicionar tratamento de eventos. Para importar todas essas tags, no nosso arquivo `.xhtml`, basta declararmos como *namespace XML* no nosso arquivo. Dessa forma, teremos:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:h="http://java.sun.com/jsf/html"
xmlns:f="http://java.sun.com/jsf/core">

</html>
```

Agora precisamos colocar uma mensagem de boas vindas para o usuário. Como todo arquivo HTML, tudo o que será renderizado no navegador deve ficar dentro de uma Tag `body`. Utilizaremos a Tag `h:head` para definirmos o cabeçalho da página contendo o título dela e também a Tag `h:body` na qual colocaremos o conteúdo da nossa página.

Um primeiro exemplo pode ser uma tela simples com um campo de texto e um botão que não faz nada:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:h="http://java.sun.com/jsf/html"
xmlns:f="http://java.sun.com/jsf/core">

<h:head>
    <title>Sistema de Notas Fiscais</title>
</h:head>

<h:body>
    <h:form>
        <h:inputText />
        <h:commandButton value="Clique-me" />
    </h:form>
</h:body>

</html>
```

Repare no uso das tags `h:form`, `h:inputText` e `h:commandButton`. Elas gerarão o código HTML necessário para o exemplo funcionar.



Saber inglês é muito importante em TI

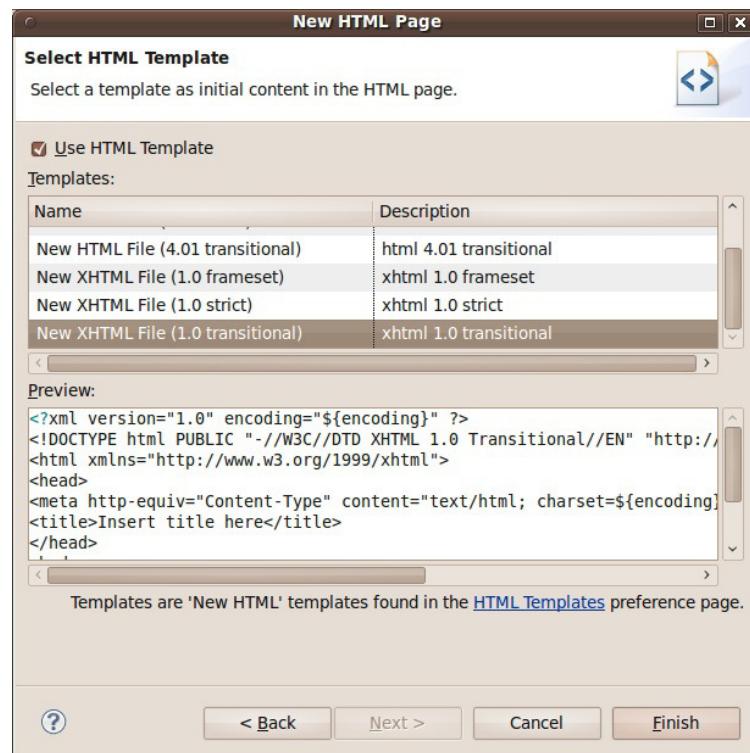
galandra

O **Galandra** auxilia a prática de inglês através de *flash cards* e *spaced repetition learning*. Conheça e aproveite os preços especiais.

[Pratique seu inglês no Galandra.](#)

3.5 EXERCÍCIOS: PRIMEIRA PÁGINA

1. Em **WebContent**, através do menu **File -> New -> Other -> HTML** crie um arquivo chamado **index.xhtml**. Após definir o nome do arquivo clique em **Next** e você verá a seguinte tela:



Selecione a mesma opção da imagem acima.

Implemente nosso primeiro código JSF com um campo de texto e um botão:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:f="http://java.sun.com/jsf/core"
xmlns:h="http://java.sun.com/jsf/html">
```

```

<h:head>
    <title>Sistema de Notas Fiscais</title>
</h:head>

<h:body>
    <h:form>
        <h:inputText />
        <h:commandButton value="Clique-me" />
    </h:form>
</h:body>
</html>

```

2. Inicie o Glassfish e acesse a URL: <http://localhost:8080/fj26-notas-fiscais/index.xhtml> O resultado deve ser esse:



3. Verifique o código fonte gerado pela página. Repare que não é nada mais que simples HTML.

3.6 CRIANDO O FORMULÁRIO DE CADASTRO

Para fazermos o cadastro de produtos, vamos precisar criar um formulário com os campos da entidade. Vamos utilizar para isso algumas tags da taglib *html* do JSF.

O primeiro passo é delimitarmos os campos que irão compor nosso formulário. Para isso, utilizaremos a Tag `h:form`. Essa tag gerará o equivalente à Tag `form` do HTML.

A Tag do JSF para fazer um campo de texto é a tag `h:inputText`. Além disso, temos o campo com a descrição do produto. Como esse campo poderá aceitar textos bem longos, vamos criar um `h:inputTextarea` que gerará o equivalente à Tag `<textarea>` do HTML.

Temos que criar o botão para submeter o formulário. Faremos isso através da Tag `h:commandButton`. Por fim, só falta criarmos os *labels* dos campos. Para isso, podemos utilizar a tag `h:outputLabel`, que gerará o equivalente à tag `label` do HTML. Podemos também indicar qual é o campo ao qual o label se refere dando um `id` para o campo e dizendo que o label é referente àquele campo através do atributo `for`.

Dessa forma, poderíamos ter o seguinte código para definir o formulário para cadastrar produtos:

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:h="http://java.sun.com/jsf/html"
xmlns:f="http://java.sun.com/jsf/core">

<h:head>
    <title>Sistema de Notas Fiscais</title>

```

```

</h:head>
<h:body>
    <h:form>
        <h:panelGrid columns="2">

            <h:outputLabel value="Nome:" for="nome" />
            <h:inputText id="nome" />

            <h:outputLabel value="Descrição:" for="descricao" />
            <h:inputTextarea id="descricao" />

            <h:outputLabel value="Preço:" for="preco" />
            <h:inputText id="preco" />

            <h:commandButton value="Gravar" />

        </h:panelGrid>
    </h:form>
</h:body>
</html>

```

ORGANIZANDO E ALINHANDO OS COMPONENTES NA TELA

Repare que no exemplo anterior utilizamos a Tag `h:panelGrid`. Ela organiza os componentes dentro do painel em tabelas contendo o número de colunas que você determinar no atributo `columns`.

Muitas pessoas não gostam dessa abordagem, pois gera tabelas para dados que não são tabulares, no caso o formulário. Para resolver esse problema, pode-se aplicar também estilos CSS em seus formulários, fazendo assim com que seu formulário seja *Tableless*.

Os componentes simples de entrada de dados disponíveis no JSF são:

<code><h:inputText/></code>	<input type="text" value="Texto"/>
<code><h:inputSecret/></code>	<input type="password" value="....."/>
<code><h:inputTextarea/></code>	<input type="text" value="Texto"/>
<code><h:inputHidden/></code>	<code><input type='hidden' value='Escondido' /></code>
<code><h:selectBooleanCheckbox/></code>	<input checked="" type="checkbox"/> Sim ou não
<code><h:selectOneMenu/></code>	<input type="button" value="Opção 3"/>
<code><h:selectOneRadio/></code>	<input type="radio"/> Opção 1 <input checked="" type="radio"/> Opção 2 <input type="radio"/> Opção 3
<code><h:selectManyCheckbox/></code>	<input checked="" type="checkbox"/> Opção 1 <input type="checkbox"/> Opção 2 <input checked="" type="checkbox"/> Opção 3
<code><h:selectManyMenu/></code>	<input type="button" value="Opção 1"/> <input type="button" value="Opção 2"/> <input type="button" value="Opção 3"/>

3.7 EXERCÍCIOS: A PÁGINA DE CADASTRO DE PRODUTOS

- Crie uma nova página no diretório `WebContent`, chamada `produto.xhtml`, para conter nosso cadastro de produtos.

A estrutura dessa página é parecida com nosso formulário anterior (com `html`, `h:head`, `h:body` e um `h:form`), a diferença é que os campos do formulário devem ser os da entidade `Produto`:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:f="http://java.sun.com/jsf/core"
xmlns:h="http://java.sun.com/jsf/html">

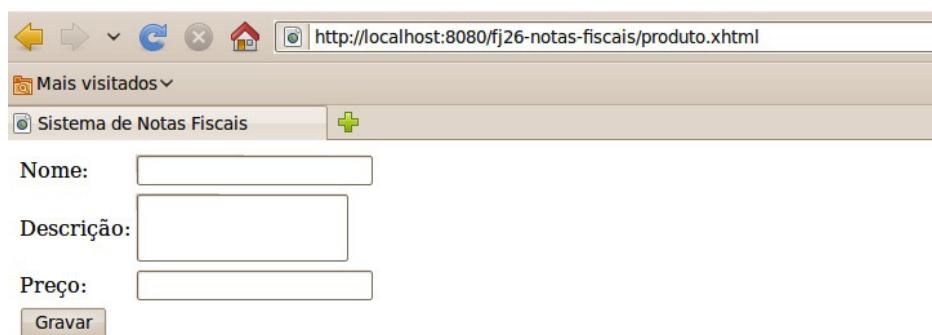
<h:head>
    <title>Sistema de Notas Fiscais</title>
</h:head>
<h:body>
    <h:form>
        <h:panelGrid columns="2">
            <h:outputLabel value="Nome:" for="nome" />
            <h:inputText id="nome" />

            <h:outputLabel value="Descrição:" for="descricao" />
            <h:inputTextarea id="descricao" />

            <h:outputLabel value="Preço:" for="preco" />
            <h:inputText id="preco"/>

            <h:commandButton value="Gravar" />
        </h:panelGrid>
    </h:form>
</h:body>
</html>
```

- Acesse a página em `http://localhost:8080/fj26-notas-fiscais/produto.xhtml`. Repare que a tela ainda não possui comportamento. Vamos adicioná-lo em seguida.



Seus livros de tecnologia parecem do século passado?



Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**. Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

[Casa do Código, Livros de Tecnologia.](#)

3.8 NOVIDADE DO JAVA EE 7: JSF2.2 E HTML 5

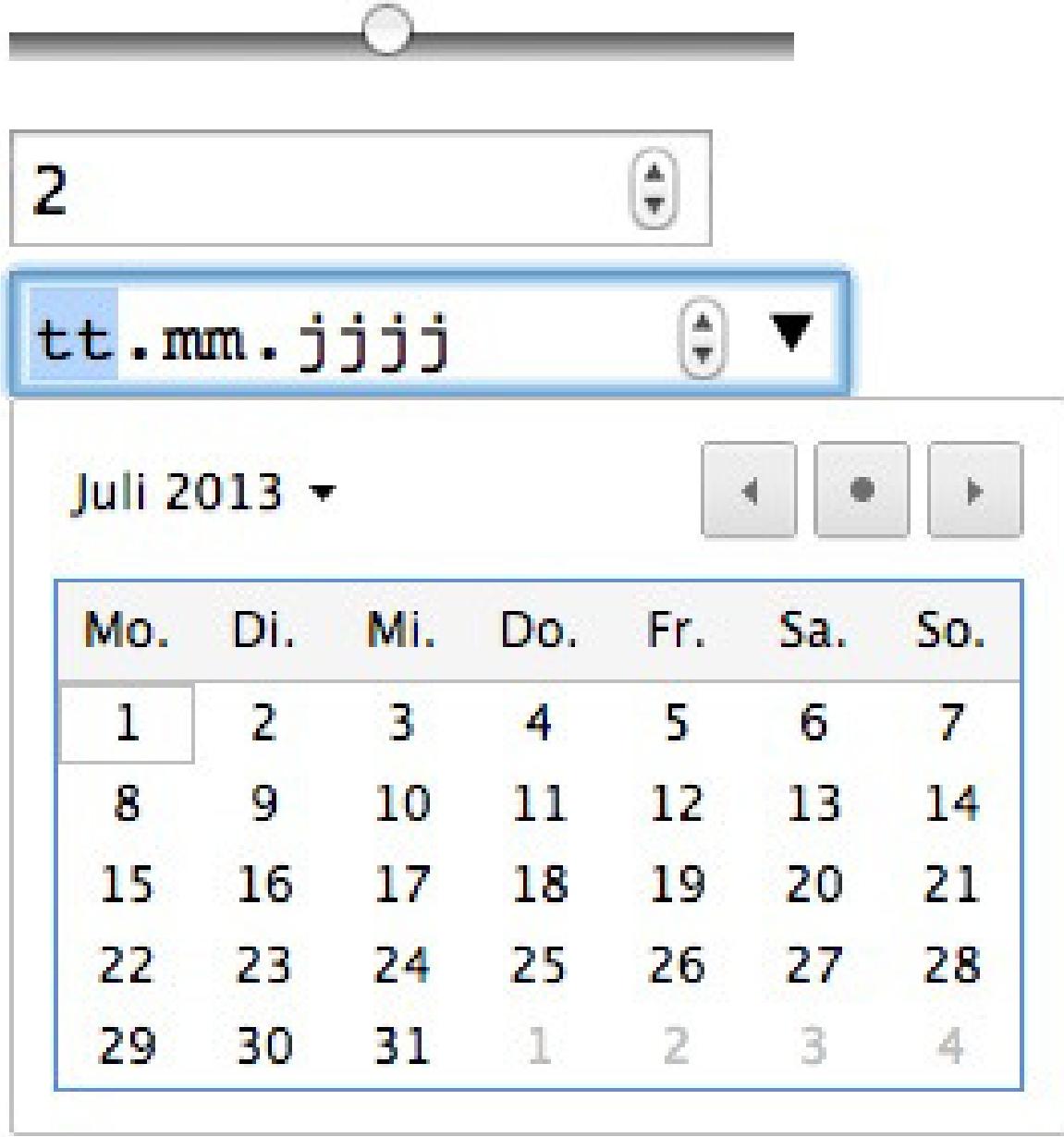
O HTML 5 já vem com uma série de componentes prontos que facilitam o desenvolvimento web. O JSF 2.2 suporta o HTML 5 através de uma técnica que simplesmente repassa os atributos de um componente para o navegador, deixando que ele os interprete, algo que não era possível nas implementações anteriores ao JSF 2.2.

Para tal, foi criado um novo componente **f:passThroughAttribute** que deve ser associado com um *input*. Nele podemos especificar todos os atributos que queremos repassar para o navegador, os quais o JSF deve ignorar:

```
<h:form>
    <h:inputText id="range">
        <f:passThroughAttribute name="type" value="range"/>
    </h:inputText>

    <h:inputText id="numero">
        <f:passThroughAttribute name="type" value="number"/>
    </h:inputText>

    <h:inputText id="data">
        <f:passThroughAttribute name="type" value="date"/>
    </h:inputText>
</h:form>
```



3.9 MANAGED BEANS

Recebendo dados e adicionando comportamentos

Agora que já temos a tela pronta, precisamos recuperar os dados digitados no formulário para que, em seguida, possamos gravá-los no banco de dados.

O primeiro passo é criarmos uma classe que receberá os valores da requisição e as guardará. Essa classe, no JSF, chamamos de **Managed Bean**, e é um simples POJO (*Plain Old Java Object*).

POJO (PLAIN OLD JAVA OBJECT)

POJO é um termo criado por Martin Fowler, Rebecca Parsons e Josh MacKenzie que serve para definir um objeto simples. Segundo eles, o termo foi criado pois ninguém usaria objetos simples nos seus projetos pois não existia um nome extravagante para ele.

Os objetos dessa classe terão seus métodos invocados a partir de comandos na web.

O JSF tenta trazer à Web um estilo de programação similar ao do Desktop, onde o clique de um botão invoca um método em seu *Managed Bean*. Como os "eventos" de algumas outras linguagens de programação.

Para declarar um *Managed Bean*, basta criarmos uma classe e anotá-la com `@ManagedBean` :

```
@ManagedBean  
public class ProdutoBean {  
  
    // vamos adicionar códigos aqui  
  
}
```

A anotação `@ManagedBean` surgiu na versão 2.0 do JSF. Até a versão 1.2, essa declaração era feita utilizando um arquivo XML, chamado `faces-config.xml`. Dessa forma, para declararmos o mesmo *Managed Bean* na versão 1.2 do JSF tínhamos que fazer a seguinte declaração no `faces-config.xml` :

```
<faces-config ...>  
    <managed-bean>  
        <managed-bean-name>produtoBean</managed-bean-name>  
        <managed-bean-class>  
            br.com.caelum.notasfiscais.mb.ProdutoBean  
        </managed-bean-class>  
        <managed-bean-scope>request</managed-bean-scope>  
    </managed-bean>  
</faces-config>
```

Perceba que o que declaramos no XML, tanto o escopo (a Tag `managed-bean-scope`) e o nome (a Tag `managed-bean-name`), não foram feitas via anotação. O que acontece é que quando não especificamos o nome do *Managed Bean*, o padrão é utilizar o nome da classe com a **primeira letra em minúsculo**. Poderíamos mudar o nome do nosso *Managed Bean* através do parâmetro `name` na notação, dessa forma ficaria:

```
@ManagedBean(name="managedBeanProduto")  
public class ProdutoBean { ... }
```

Com relação ao escopo, quando ele não é definido, o JSF assume como padrão o escopo de `request`. Para defini-lo, basta anotar o seu *Managed Bean* com uma anotação referente ao escopo que queira utilizar, por exemplo, para o escopo de sessão, bastaria utilizar a anotação `@SessionScoped`.

```
@SessionScoped  
@ManagedBean(name="managedBeanProduto")  
public class ProdutoBean { ... }
```

ESCOPOS NO JSF

É muito importante conhecer os escopos do JSF, pois eles podem influenciar bastante no comportamento das nossas aplicações. Muitas vezes, a escolha do escopo para o seu Managed Bean pode inclusive impactar na escalabilidade da aplicação.

Durante o curso, veremos todos os escopos disponíveis pelo JSF em seu momento adequado.

No nosso caso, como não precisamos manter os dados do produto após a requisição, vamos deixar o *Managed Bean* no escopo *request*.

PRECEDÊNCIA ENTRE XML E ANOTAÇÕES

No JSF 2.0, você pode usar tanto as configurações via anotação, quanto via XML. No caso de ambas existirem, a configuração via XML sobrescreverá a configuração das anotações.

O próximo passo é representar os dados do formulário no nosso *Managed Bean*. Como nosso formulário possui dados de produto, como o nome, descrição e o preço, nada mais justo do que criarmos na nossa classe `ProdutoBean` três atributos para representar os valores.

```
@ManagedBean  
public class ProdutoBean {  
    private String nome;  
    private String descricao;  
    private Double preco;  
}
```

No entanto, essa abordagem não é interessante e pouco orientada a objetos, pois nós já possuímos uma classe que representa o produto. Bastaria utilizarmos a nossa classe para isso, dessa forma teríamos no `ProdutoBean` :

```
@ManagedBean  
public class ProdutoBean {  
    private Produto produto;  
}
```

3.10 LIGANDO MANAGED BEANS A COMPONENTES VISUAIS

Precisamos dizer no nosso formulário que os campos criados na página `produto.xhtml` irão

preencher os seus respectivos atributos no *Managed Bean*. Para isso, no arquivo `produto.xhtml` devemos indicar quais campos dentro do atributo `produto` do *Managed Bean* `produtoBean`, devem ser preenchidos, como por exemplo, o atributo `nome`.

Para fazer a **ligação** entre os valores dos componentes e os atributos do *Managed Bean*, usamos a **Expression Language do JSF**. Esta ligação é um dos principais conceitos do JSF, e é conhecido como **binding**.

Ao fazer o *binding*, o valor do componente passa a ser **representado** pela propriedade do *Managed Bean*. Desta forma, ao alterar o valor do componente, a propriedade do *Managed Bean* é automaticamente modificada pelo JSF.

O contrário também é válido: ao alterar a propriedade do *Managed Bean*, o valor do componente também muda. Os dois estão realmente **ligados**. Bem diferente do que acontece com um JSP comum.

Essa indicação pode ser feita através do atributo `value` que é passado para a Tag `h:inputText` que representa o nome do produto. Dessa forma, para ligarmos o `inputText` do nome do produto que está no `ProdutoBean`, fazemos o seguinte:

```
<h:inputText value="#{produtoBean.produto.nome}" />
```

O que acontece na linha acima é que, dentro da instância do *Managed Bean* `produtoBean`, estamos acessando a propriedade `produto`. Dentro do `produto`, queremos atribuir o `nome`. Para que o JSF faça isso, no seu `produtoBean`, teremos que adicionar um método *getter* para o `produto`. Em seguida, na instância de `Produto` que foi recuperada, será invocado o *setter* do `nome`. Um cuidado que temos que tomar é que como o JSF invocará o *getter* para o `produto` no `produtoBean`, o `produto` precisará estar instanciado, caso contrário tomaremos uma `NullPointerException` quando o JSF invocar o *setter* para o atributo `nome`.

Para termos esse comportamento, basta utilizarmos a *Expression Language* do JSF, como a seguir:

```
<h:outputLabel value="Nome:" />
<h:inputText value="#{produtoBean.produto.nome}" />
```

E também não esquecemos de criar o *getter* do `produto` no `ProdutoBean`:

```
@ManagedBean
public class ProdutoBean {
    private Produto produto = new Produto();

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

Precisamos colocar a ação para o nosso botão. Como já sabemos o conceito do *binding*, fica fácil imaginar que precisamos associar o botão com o método que fará a gravação do `produto` no banco de dados. Para isso, basta criarmos no nosso `ProdutoBean` um método que utilizará o `ProdutoDao` para

isso. Então, poderíamos ter o seguinte método `grava` :

```
@ManagedBean  
public class ProdutoBean {  
    private Produto produto = new Produto();  
  
    public void grava() {  
        ProdutoDao dao = new ProdutoDao();  
        dao.adiciona(produto);  
    }  
  
    // getter  
}
```

Um último detalhe é que agora ao acessarmos a página `produto.xhtml` no navegador e realizarmos a gravação do produto, o formulário vai continuar preenchido. Para resolvemos isso e limparmos o formulário, basta fazermos o atributo `produto` apontar para uma instância nova de `Produto`, dessa forma, como existe o *binding*, os campos ficarão vazios.

```
@ManagedBean  
public class ProdutoBean {  
    private Produto produto = new Produto();  
  
    public void grava() {  
        ProdutoDao dao = new ProdutoDao();  
        dao.adiciona(produto);  
        this.produto = new Produto();  
    }  
  
    // getter  
}
```

LIMPANDO O FORMULÁRIO

Pode parecer estranho termos que fazer o `new` para limpar o formulário que estava previamente preenchido. No entanto, faz bastante sentido quando pensamos que o JSF guarda o estado dos componentes. Lembre que os componentes visuais estão **ligados** com os objetos do *Managed Bean*, dessa forma o nosso objeto controla o que será visualizado na tela.

Por fim, basta fazer o *binding* para o método `grava` no atributo `action` do botão que criamos no `produto.xhtml` :

```
<h:commandButton value="Gravar" action="#{produtoBean.grava}" />
```

Quando o usuário preenche o formulário e o submete, uma nova requisição é gerada no servidor. Como o formulário possui binding com o `ProdutoBean` e ele é *request scoped*, uma nova instância do *managed bean* é criada para essa requisição. Os dados são então populados no objeto `Produto` dentro do bean. Logo em seguida, a ação `grava` é chamada e o objeto é inserido no banco. Para montar a resposta de volta para o cliente (nesse caso, um formulário vazio), o JSF consulta novamente o *managed*

bean para saber quais valores devem ser colocados na tela. Por isso, a importância de criar um novo Produto vazio no final do método grava . Só quando o *request* é completamente tratado e sua resposta é gerada que a instância do ProdutoBean é descartada.

Agora é a melhor hora de respirar mais tecnologia!

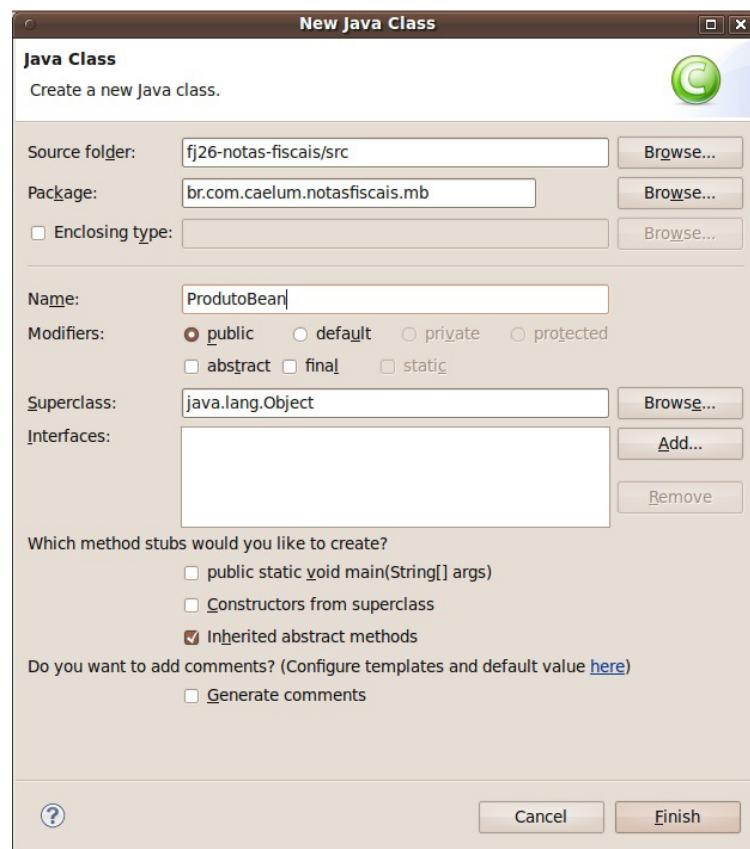


Se você está gostando dessa apostila, certamente vai aproveitar os **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum. Programação, Mobile, Design, Infra, Front-End e Business! Ex-aluno da Caelum tem 15% de desconto, siga o link!

[Conheça a Alura Cursos Online.](#)

3.11 EXERCÍCIOS: GRAVAÇÃO DE PRODUTOS

1. Crie o *Managed Bean* para gerenciar o cadastro dos produtos dentro do pacote br.com.caelum.notasfiscais.mb . Chame a classe de ProdutoBean :



Ela deve ser anotada com `@ManagedBean` do JSF 2 e, por enquanto, terá apenas um `Produto` e seu `getter`.

ATENÇÃO: Cuidado, a anotação `@ManagedBean` é do pacote `javax.faces.bean`:

```
import javax.faces.bean.ManagedBean;

@ManagedBean
public class ProdutoBean {
    private Produto produto = new Produto();

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

2. Faça o *binding* dos componentes do `produto.xhtml` com o atributo novo que criamos no *Managed Bean*. Basta usar o atributo `value` nos *inputs* apontando para a propriedade dentro do *Managed Bean*. Edite então os *inputs* **acrescentando o `value` em cada um deles**:

```
<h:outputLabel value="Nome:" for="nome" />
<h:inputText id="nome" value="#{produtoBean.produto.nome}" />

<h:outputLabel value="Descrição:" for="descricao" />
<h:inputTextarea id="descricao" value="#{produtoBean.produto.descricao}" />

<h:outputLabel value="Preço:" for="preco" />
<h:inputText id="preco" value="#{produtoBean.produto.preco}" />
```

3. O próximo passo é criarmos o método que fará a gravação de um produto no banco de dados. Para isso, vamos criar o método `grava` no *Managed Bean* `ProdutoBean` que usará o `ProdutoDao`:

```
public void grava() {
    ProdutoDao dao = new ProdutoDao();
    dao.adiciona(produto);
    this.produto = new Produto();
}
```

4. Por fim, precisamos fazer o *binding* do botão com o novo método `grava` que acabamos de criar. Altere então o código do seu botão em `produto.xhtml` para usar o atributo `action`:

```
<h:commandButton value="Gravar" action="#{produtoBean.grava}" />
```

5. Crie o banco de dados:

```
mysql -u root
create database fj26;
```

Reinic peace o servidor e tente cadastrar um produto.

6. Para termos certeza de que o produto foi cadastrado, podemos acessar o MySQL e verificar se o registro está cadastrado no banco de dados. Para isso, acesse o terminal e digite:

```
mysql -u root fj26;
```

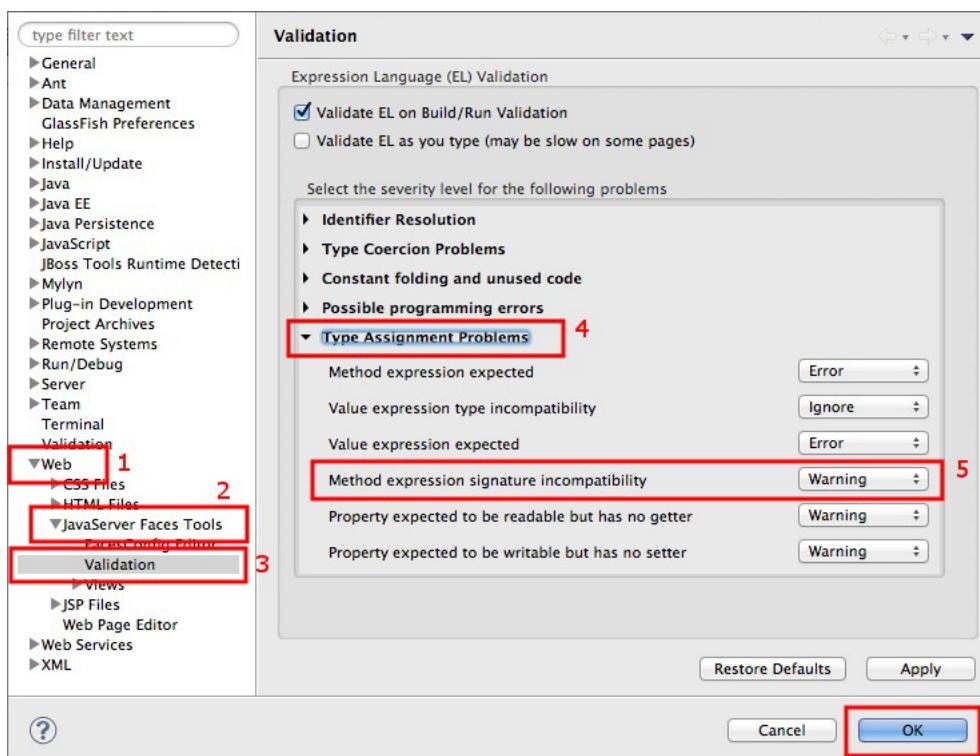
```
select * from PRODUTO;
```

BUG NO PLUGIN WTP DO ECLIPSE

Em algumas versões do Eclipse, por conta de um Bug na validação dos facelets quando fazem a chamada de métodos do *Managed Bean* via *Expression Language*, pode ser necessária uma mudança na configuração que define a maneira que o Eclipse fará a validação desse código. Caso você esteja com um erro do tipo: `Method must have signature "String method(), String method(), String method(String), String method(String, String), ..."` but has signature "void method()"

faça a configuração abaixo:

- Acesse o menu: *Window > Preferences*;
- Depois, siga o caminho nas árvores de configurações: *Web > JavaServer Faces Tools > Validation > Type Assignment Problems > Method expression signature incompatibility* e troque o tratamento de *error* para *Ignore* ou *Warning*;
- Depois é só pressionar o botão "Ok" e pronto.



3.12 LISTAGEM COM DATATABLE

Agora que já temos pronto o cadastro dos produtos da nossa aplicação, precisamos visualizá-los na

nossa tela. Para isso, vamos fazer uma listagem dos produtos cadastrados abaixo do formulário. O primeiro passo para fazermos essa listagem é conhecer o componente que vai exibir os produtos na tela. Esse componente é o `h:dataTable`.

Para utilizarmos o `h:dataTable`, primeiramente temos que indicar de qual lugar virão os dados. Indicamos esses valores no atributo `value` do `h:dataTable` que espera receber uma referência para um método `getter` que retorne um `java.util.List` ou um `array`.

Dessa forma, para declarar o `h:dataTable` passaremos via *Expression Language* uma referência para um `getter`:

```
<h:dataTable value="#{produtoBean.produtos}">  
</h:dataTable>
```

A *Expression Language* `#{{produtoBean.produtos}}` está apontando para um método chamado `getProdutos` no *Managed Bean* `produtoBean` que nós ainda precisamos implementar. Ao invocar o método `getProdutos`, devolveremos uma lista de produtos que será buscada no banco de dados.

Nesse ponto, surge um detalhe crucial para o bom funcionamento da sua aplicação JSF. Quando uma requisição é enviada para o JSF, ele faz diversas tarefas internamente, as quais estudaremos mais adiante, e que acabam por invocar os métodos `getters` do seu *Managed Bean* e isso pode causar problemas de performance caso você dispare consultas no banco de dados no seu método `getter`.

Teoricamente, um método `getter`, indica uma ligação com um *Managed Bean*, e ele não deveria ter cálculos ou buscas em um banco de dados, justamente pela necessidade do JSF de invocá-lo N vezes dentro do mesmo `request`. Um truque simples para resolver o problema, é guardar a lista em um atributo do *managed bean* na primeira vez que ela for buscada no banco:

```
@ManagedBean  
public class ProdutoBean {  
    private List<Produto> produtos;  
  
    //outros métodos e atributos  
  
    public List<Produto> getProdutos() {  
        if (produtos == null) {  
            System.out.println("Carregando produtos...");  
            produtos = new ProdutoDao().listaTodos();  
        }  
  
        return produtos;  
    }  
}
```

Para determinarmos como os dados serão exibidos na tela, precisamos alterar um pouco o nosso `h:dataTable`. O `h:dataTable` funciona como um `for`, no qual você diz qual a `Collection` será percorrida e qual a variável em que ele armazenará o item atual durante a iteração. No `h:dataTable`, definimos essa variável temporária através do atributo `var`, no qual podemos dar um nome qualquer,

como produto :

```
<h:dataTable value="#{produtoBean.produtos}" var="produto">  
</h:dataTable>
```

Agora que já temos o item que está sendo iterado no momento, no caso representado pela variável `produto`, precisamos determinar as colunas que vão existir em nossa tabela. Para isso, adicionamos a Tag `h:column` dentro da nossa tabela para cada coluna existente na mesma. Dentro da `h:column`, colocamos o valor que queremos que seja renderizado na tela, podendo utilizar a *Expression Language* para recuperar os valores do item iterado naquele momento:

```
<h:dataTable value="#{produtoBean.produtos}" var="produto">  
  
    <h:column>  
        #{produto.nome}  
    </h:column>  
  
    <h:column>  
        #{produto.descricao}  
    </h:column>  
  
    <h:column>  
        #{produto.preco}  
    </h:column>  
  
</h:dataTable>
```

Por fim, precisamos mostrar o cabeçalho das colunas para podermos identificar facilmente do que se trata cada coluna. Para adicionarmos o cabeçalho, utilizaremos a Tag `f:facet`. A Tag `f:facet` modifica um critério de visualização do seu componente pai, ou seja, se colocarmos uma tag `f:facet` dentro de um `h:column` ela modificará a coluna, que é seu componente pai.

Para indicarmos qual característica da coluna queremos mudar, basta indicá-la para o seu `f:facet` através do atributo `name`. No caso da coluna, é possível definir o `header` (cabeçalho) e o `footer` (rodapé):

```
<h:dataTable value="#{produtoBean.produtos}" var="produto">  
  
    <h:column>  
        <f:facet name="header">Nome</f:facet>  
        #{produto.nome}  
    </h:column>  
  
    <h:column>  
        <f:facet name="header">Descrição</f:facet>  
        #{produto.descricao}  
    </h:column>  
  
    <h:column>  
        <f:facet name="header">Preço</f:facet>  
        #{produto.preco}  
    </h:column>  
  
</h:dataTable>
```

Ao acessarmos a nossa página que contém o formulário e logo abaixo a listagem, e gravarmos alguns produtos, reparamos um grave problema. A listagem dos produtos nunca é atualizada após a gravação de novos produtos. Como podemos resolver esse problema?

Esse é um problema fácil de perceber, mas que quando começamos com JSF pode nos custar algumas horas para entendermos e corrigirmos corretamente. A questão é que o *binding* do `h:dataTable` está feito com o `getProdutos`, que nada mais faz do que retornar a lista de produtos que até então só foi buscada quando uma requisição foi feita para a aplicação. Precisamos recuperar a nova listagem no banco de dados (que agora conterá um produto novo que acabou de ser adicionado) logo após a gravação do mesmo no banco de dados. Para isso, basta adicionar ao fim do método `adiciona` uma chamada para o `ProdutoDao` que fará essa busca:

```
public void grava() {  
    ProdutoDao dao = new ProdutoDao();  
    dao.adiciona(produto);  
    this.produto = new Produto();  
    this.produtos = dao.listaTodos();  
}
```

Repare que essa é uma forma bem mais próxima à maneira *Desktop* de se desenvolver aplicações, onde controlamos via código o comportamento do que e como as informações serão exibidas.

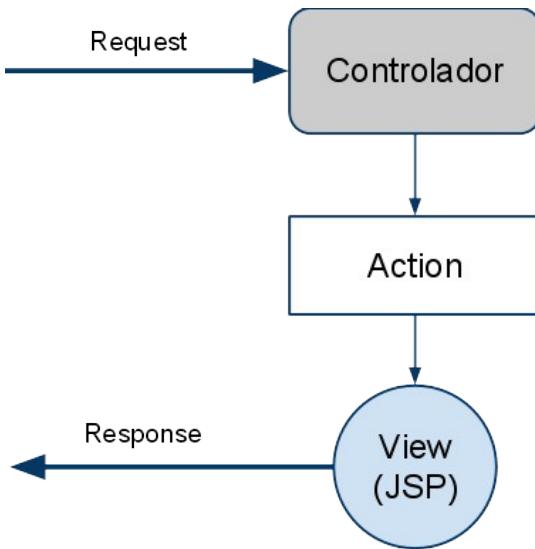
3.13 DIFERENTES ESTILOS DE MVC: O MVC PUSH E O PULL

Um fator de fundamental importância para entendermos qualquer framework web é compreender o padrão MVC e como ele favorece a separação de responsabilidades de uma aplicação. Para compreendermos melhor como o MVC é empregado ao trabalharmos com o JSF, precisamos entender como ele é utilizado em outros frameworks. Vamos tomar como base o clássico framework Struts; no entanto, a ideia pode ser aplicada para praticamente qualquer outro framework.

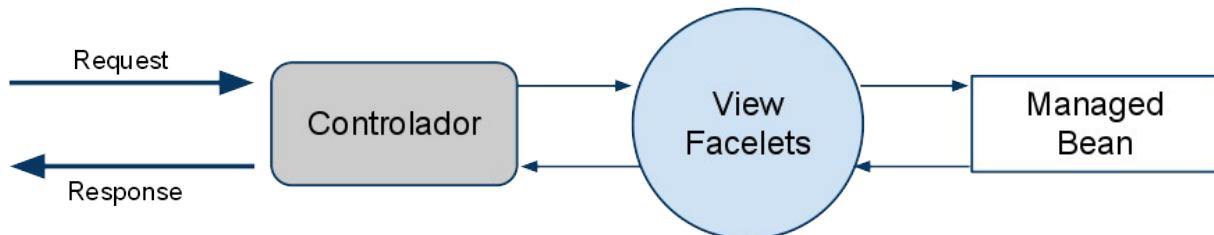
Quando uma requisição é disparada para uma aplicação, ela sempre será tratada por um Servlet, que conhecemos como sendo o controller da nossa aplicação. No caso do Struts, sua função é descobrir qual Ação (*Action*) deve ser realizada. A *Action*, que é uma classe com algumas características similares ao *Managed Bean* do JSF, tem como função invocar as classes de Model, como os *beans*, contendo regras de negócio e DAOs.

Ao fim da execução, a *Action* devolve um sinal ao framework indicando qual foi o resultado da sua execução, para que possa ser decidido qual view deve ser renderizada para o cliente. Porém, na grande maioria das vezes, a *View* precisa de alguns dados para exibir, como é o caso de uma listagem. Esses dados são disponibilizados para a *View* pela *Action*. Ou seja, é como se a *Action* "empurrasse" os dados para a *View*, e ela os recebesse e utilizasse. Esse estilo de MVC é conhecido como *MVC Push*, nos quais os dados são empurrados para a *View* e é encontrado em frameworks como o Struts, VRaptor e SpringMVC.

E pode ser visto no diagrama a seguir:



Porém, o JSF funciona de forma diferente, apesar de também utilizar o MVC para a separação das responsabilidades. Ao enviar uma requisição para o JSF, estamos acessando o XHTML, ou seja, a View é quem recebe a requisição (a requisição ainda passa pelo framework, ou seja continua passando pelo Controller, no entanto, o primeiro passo dentro do framework é a execução da View). Conforme a View vai sendo processada, ela vai demandando as informações que precisa para os Managed Beans. Isso se dá através dos *bindings* feitos entre os componentes e os próprios beans. Os Managed Beans por sua vez, delegam para as classes de Modelo. Repare que continuamos tendo o MVC, mas estruturado de uma maneira diferente, no qual a View pede tudo o que precisa ao Managed Bean. Esse estilo de MVC é o MVC Pull, onde a View busca os dados necessários.



Editora Casa do Código com livros de uma forma diferente



Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.

Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.](#)

3.14 EXERCÍCIOS: LISTAGEM DE PRODUTOS

- Para fazermos a listagem dos produtos, adicione na sua classe `ProdutoBean` o método `getProdutos` e o atributo `produtos` que é uma `List<Produto>`. Seu método deve fazer a busca com o DAO e verificar se a busca já não foi feita para evitar várias chamadas ao banco:

```
@ManagedBean  
public class ProdutoBean {  
    private List<Produto> produtos;  
  
    public List<Produto> getProdutos() {  
        if (produtos == null) {  
            System.out.println("Carregando produtos...");  
            produtos = new ProdutoDao().listaTodos();  
        }  
  
        return produtos;  
    }  
}
```

- Adicione, abaixo do seu formulário, no `produto.xhtml`, os componentes para gerar a tabela:

```
<!-- aqui em cima vai o formulário para gravar os produtos -->  
  
<h:dataTable value="#{produtoBean.produtos}" var="produto">  
  
    <h:column>  
        <f:facet name="header">Nome</f:facet>  
        #{produto.nome}  
    </h:column>  
  
    <h:column>  
        <f:facet name="header">Descrição</f:facet>  
        #{produto.descricao}  
    </h:column>  
  
    <h:column>  
        <f:facet name="header">Preço</f:facet>  
        #{produto.preco}
```

```
</h:column>
```

```
</h:dataTable>
```

3. Acesse `produto.xhtml` no seu navegador e veja o resultado final. Deve ser algo como:



Adicione produtos no banco de dados. Algo está errado?

O problema é que a lista de produtos não é atualizada após fazermos a gravação. Para resolver isso, basta adicionar no seu método `grava` o código para buscar a lista atualizada do banco de dados após a inserção:

```
public void grava() {
    ProdutoDao dao = new ProdutoDao();
    dao.adiciona(produto);
    this.produto = new Produto();
    this.produtos = dao.listaTodos();
}
```

3.15 EXERCÍCIOS OPCIONAIS

1. Para praticar o conceito do `f:facet` e do *binding* dos componentes visuais com o *Managed Bean*, faça com que a coluna do preço tenha um rodapé onde deve ser exibido um somatório dos valores dos produtos. Para isso, altere o seu *Managed Bean* livremente.

3.16 REMOVENDO UM PRODUTO E PARÂMETROS DE EL

Agora que conseguimos visualizar os produtos cadastrados no sistema, precisamos também removê-los, caso seja necessário.

O primeiro passo para conseguirmos fazer a remoção dos produtos é adicionar uma coluna na nossa tabela, contendo um link para efetuar a remoção. Para isso, utilizaremos a Tag `h:commandLink`:

```
<h:column>
    <f:facet name="header">Ações</f:facet>
    <h:commandLink action="#{produtoBean.remove(produto)}" value="Remover" />
</h:column>
```

Repare na chamada ao método `remove` do *managed bean* a passagem de um parâmetro! A partir da

EL 2.2 é possível passar parâmetros para métodos na *expression language*.

A ação para o botão remover é parecida com a de gravar, mas agora recebe um argumento com o produto a ser removido:

```
public void remove(Produto produto) {  
    ProdutoDao dao = new ProdutoDao();  
    dao.remove(produto);  
    this.produtos = dao.listaTodos();  
}
```

Um último detalhe é que, no JSF, todos os botões e links devem estar dentro de algum formulário (exceto os `h:button` e `h:link`, introduzidos no JSF 2.0, que falaremos mais a frente). Por isso, devemos envolver a nossa tabela com um formulário (`h:form`):

```
<h:form>  
    <h:dataTable value="#{produtoBean.produtos}" var="produto">  
  
        <!-- colunas vão aqui -->  
  
    </h:dataTable>  
</h:form>
```

Já conhece os cursos online Alura?



A Alura oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Programação, Front-end, Mobile, Design & UX, Infra e Business, com um plano que dá acesso a todos os cursos. Ex aluno da Caelum tem 15% de desconto neste link!

[Conheça os cursos online Alura.](#)

3.17 EXERCÍCIOS: REMOÇÃO DE PRODUTOS

1. Abra seu arquivo `produto.xhtml` e crie uma nova coluna com o *link* de remover. Esse link deve invocar o método `remove` que já criaremos no *managed bean*. Não esqueça também de **envolver a tabela toda em um formulário** (`h:form`):

```
<h:form>  
    <h:dataTable value="#{produtoBean.produtos}" var="produto">  
  
        <!-- outras colunas aqui -->  
  
        <h:column>  
            <f:facet name="header">Ações</f:facet>
```

```

        <h:commandLink action="#{produtoBean.remove(produto)}"
                      value="Remover" />
    </h:column>

</h:dataTable>
</h:form>

```

2. Implemente o método `remove` na classe `ProdutoBean` recebendo o `Produto` para remoção e usando o `ProdutoDao` para remover no banco. Não esqueça de refazer a listagem no final para que a tela seja atualizada.

```

public void remove(Produto produto) {
    ProdutoDao dao = new ProdutoDao();
    dao.remove(produto);
    this.produtos = dao.listaTodos();
}

```

3. Reinicie o servidor e acesse o endereço <http://localhost:8080/fj26-notas-fiscais/produto.xhtml> e exclua alguns produtos.

3.18 ALTERAÇÃO DE PRODUTOS E O SETPROPERTYACTIONLISTENER

Para completarmos a tela de produtos, precisamos permitir alterá-los. Para isso, a abordagem que utilizaremos é a de clicar em um link chamado "Alterar" e os dados do produto deverão aparecer no formulário do cliente.

Para fazermos isso, criaremos mais um `h:commandLink` dessa vez chamado "Alterar". Ao clicar no link, precisamos pegar o item que ele clicou, e exibi-lo no formulário. Repare que o formulário possui um *binding* com o atributo `produto`. Dessa forma, quaisquer alterações que façamos no atributo `produto` serão refletidas automaticamente na tela.

O que precisamos é uma forma de carregar dados arbitrários dentro do atributo `produto` quando o usuário clicar no link de alterar.

É possível ter esse comportamento usando um recurso chamado `PropertyActionListener`. Através da tag `f:setPropertyActionListener` é possível carregar um valor qualquer para dentro de alguma propriedade do *managed bean* quando o botão for clicado:

```

<h:commandLink value="Alterar">
    <f:setPropertyActionListener value="#{produto}"
                                 target="#{produtoBean.produto}" />
</h:commandLink>

```

O atributo `value` indica o valor que queremos atribuir (no caso, o produto atual) e o atributo `target` indica em qual propriedade do *managed bean* queremos fazer isso (no caso, a propriedade `produto` que já é usada para exibir os dados na tela).

Repare também que não temos um método a ser invocado quando o link é clicado (diferentemente da remoção e adição). No caso da edição, queremos apenas carregar os dados do objeto selecionado, e o

`f:setPropertyActionListener` é suficiente para isso.

Agora que os dados do produto foram exibidos no formulário e o usuário pode editá-los, quando ele finalizar a edição, ele pode clicar no botão salvar. No entanto, como saberemos se o usuário está adicionando um novo produto ou alterando algum já existente, visto que queremos reaproveitar o mesmo botão e método do *managed bean*?

Uma das formas de fazermos isso é verificarmos no método `grava` se o `id` do produto foi enviado. Caso tenha sido, significa que ele já existia no banco de dados, portanto, devemos alterá-lo. Caso contrário, precisamos adicioná-lo no banco de dados:

```
public void grava() {
    ProdutoDao dao = new ProdutoDao();

    if (produto.getId() == null) {
        dao.adiciona(produto);
    } else {
        dao.atualiza(produto);
    }
    produtos = dao.listaTodos();
    this.produto = new Produto();
}
```

Além disso, é preciso guardar o `id` do produto dentro do formulário. Como o usuário não deveria alterar o valor, usaremos o componente `h:inputHidden`:

```
<h:inputHidden value="#{produtoBean.produto.id}" />
```

Mas, como saber visualmente se estamos na verdade adicionando um produto novo ou editando um existente? Uma maneira elegante seria ter um título HTML na página:

```
<h2>Novo Produto</h2>
```

ou

```
<h2>Editar Produto</h2>
```

Mas como alternar o título dependendo da situação? Em JSP normal faríamos um `c:if`, mas o JSF tem um recurso mais interessante. Se transformarmos esses títulos em componentes, podemos usar o atributo `rendered` que todos os componentes têm para indicar quando renderizá-los. O `rendered` recebe uma expressão booleana onde podemos verificar o `id` do produto:

```
<h2>
    <h:outputText value="Novo Produto"
        rendered="#{empty produtoBean.produto.id}"/>
    <h:outputText value="Editar Produto"
        rendered="#{not empty produtoBean.produto.id}"/>
</h2>
```

UTILIZANDO IF PARA DETERMINAR SE O COMPONENTE SERÁ EXIBIDO OU NÃO

No JSF 1.2, onde podíamos utilizar JSP ao invés de Facelets para desenvolver a camada da visualização, era comum encontrar códigos que utilizavam a JSTL para decidir se um componente seria exibido ou não. Algo como o seguinte:

```
<c:if test="#{produtoBean.produto.id != null}">
    <h:outputText value="JSF" />
</c:if>
```

Essa é uma prática totalmente desencorajada, pois existem diversas incompatibilidades entre o ciclo de vida do JSP e do JSF e dependendo do caso, essa abordagem podia não atingir o comportamento desejado. Portanto, sempre que for necessário decidir se um componente deve ou não ser exibido, utilize o atributo `rendered`, pois ele se adequa totalmente ao JSF.

3.19 EXERCÍCIOS: ALTERANDO PRODUTOS

1. Altere o método `grava` do nosso `ProdutoBean` para que ele suporte também a atualização de produtos. Basta usar o método `atualiza` do `ProdutoDao` caso o objeto possua um `id`:

```
public void grava() {
    ProdutoDao dao = new ProdutoDao();

    if (produto.getId() == null) {
        dao.adiciona(produto);
    } else {
        dao.atualiza(produto);
    }
    produtos = dao.listaTodos();
    this.produto = new Produto();
}
```

2. **Adicione** no `dataTable` do `produto.xhtml` o *link* para a alteração do produto. Você pode criar uma nova coluna ou adicionar o *link* na coluna *Ações* onde já está a remoção.

```
<h:commandLink value="Alterar">
    <f:setPropertyActionListener value="#{produto}" 
        target="#{produtoBean.produto}" />
</h:commandLink>
```

Se você não tem um método `setProduto` no *managed bean*, **crie-o agora**. Ele é essencial para que o `f:setPropertyActionListener` consiga preencher a propriedade.

3. Coloque o título do formulário logo abaixo da tag `h:form`. Use o atributo `rendered` para mostrar a mensagem correta.

```
<h2>
    <h:outputText value="Novo Produto"
        rendered="#{empty produtoBean.produto.id}" />
```

```
<h:outputText value="Editar Produto"  
rendered="#{not empty produtoBean.produto.id}" />  
</h2>
```

4. Também use o componente `h:inputHidden` para guardar o `id` do produto no formulário. Coloque o componente logo abaixo da tag `h:form`:

```
<h:form>  
    <h:inputHidden value="#{produtoBean.produto.id}" />  
  
    <!-- demais componentes -->  
  
</h:form>
```

5. Acesse a página `produto.xhtml` no navegador e altere alguns produtos.

Saber inglês é muito importante em TI

galandra

O **Galandra** auxilia a prática de inglês através de *flash cards* e *spaced repetition learning*. Conheça e aproveite os preços especiais.

[Pratique seu inglês no Galandra.](#)

3.20 EXERCÍCIOS OPCIONAIS

1. Crie um novo botão no formulário para "Cancelar a edição".

AJAX COM JSF 2

"Pensa como pensam os sábios, mas fala como falam as pessoas simples" -- Aristóteles

4.1 AJAX

AJAX (*Asynchronous Javascript and XML*) sem dúvida é umas das tecnologias que possibilitam alcançarmos os conceitos que envolvem a web 2.0. Muitas pessoas pensam em um primeiro momento que AJAX é uma ferramenta que utilizamos em nosso código, mas na verdade, ela é apenas uma maneira de usar tecnologias já existentes, como XML e JavaScript providas pelos navegadores, e que nos permitem fazer requisições assíncronas para um servidor web.

Uma aplicação web tradicional funciona com o usuário fazendo uma requisição HTTP para um servidor web. O servidor processa a requisição e retorna uma resposta na maioria das vezes em formato HTML. Enquanto a requisição está sendo processada o usuário fica esperando a resposta, enquanto poderia continuar interagindo com a aplicação.

Outro problema de trabalharmos no modelo tradicional da web é a questão do tráfego de rede. Imagine que precisamos adicionar um item que estamos comprando em nosso carrinho de compras: se utilizássemos o modelo tradicional, faríamos uma requisição para "apenas" adicionar o produto no carrinho de compras em uma sessão no servidor, porém nossa página que já está carregada, seria recarregada. Perceba que a página que já estava aberta anteriormente terá que ser recarregada completamente, apenas para adicionar alguns poucos dados diferentes.

Esses problemas podem ser resolvidos através de AJAX, que por permitir fazer requisições assíncronas e enviar apenas as informações necessárias para que a tarefa seja executada sem interferir na navegação que o usuário possuía com a página original. No nosso exemplo anterior, queremos adicionar o produto no carrinho, e não recarregar toda a página novamente.

Seus livros de tecnologia parecem do século passado?



Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**. Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

[Casa do Código, Livros de Tecnologia.](#)

4.2 AJAX E JSF

As versões anteriores ao JSF 2.0 não possuíam formas nativas de se trabalhar com AJAX. Devido a exigência do mercado para que tivéssemos aplicações utilizando tal tecnologia foram criadas ferramentas por terceiros que nos permitiam criarmos componentes que suportem AJAX. As mais conhecidas e utilizadas são **JSF Matrix** e o **Ajax4JSF**, este último foi integrado ao **RichFaces**, que é uma ferramenta de componentes JSF mais robustos.

Vendo a necessidade que o mercado tinha, o *Expert Group* da versão 2.0 da JSF resolveu tornar o AJAX parte da especificação, para que seus usuários não precisassem recorrer a ferramentas de terceiros.

TRABALHANDO COM AJAX NAS VERSÕES ANTERIORES DO JSF

Nas versões anteriores ao JSF 2.0 também é possível trabalhar com AJAX. No entanto, para conseguirmos utilizarmos AJAX precisamos usar componentes externos, pois o JSF não possuía suporte nativo.

Um desses componentes mais comuns é o Ajax4JSF, mas existem diversos outros que podem ser encontrados e utilizados.

4.3 COMO UTILIZAR AJAX NAS APLICAÇÕES JSF

Em nossa aplicação a funcionalidade de remover um produto está gerando um tráfego de rede desnecessário e que poderia ser evitado se fosse utilizado AJAX. Vamos então alterá-lo para mostrar como é simples a utilização de AJAX em aplicações que utilizam JSF.

Pensando em uma página HTML que precisa utilizar uma biblioteca Javascript, precisamos importar

a biblioteca para a nossa página. No HTML fazemos isso com a Tag:

```
<script src="caminho/arquivo.js" type="text/javascript"></script>
```

No entanto, como estamos utilizando o JSF, temos componentes prontos que fazem essa importação da forma com que o JSF trabalha. O componente que possui esse comportamento é o `h:outputScript`. Para esse componente, passamos o atributo `name` que indica qual é o arquivo que desejamos importar. Temos também um atributo chamado `library` que indica em qual diretório, ou pacote caso o arquivo a ser importado esteja dentro de um, deve-se localizar o arquivo. E por fim, um atributo chamado `target`, que indica dentro de qual Tag esse componente será renderizado. Dessa forma temos:

```
<h:outputScript name="jsf.js" library="javax.faces" target="head"/>
```

Nessa biblioteca, está definido o método `jsf.ajax.request` que faz um requisição assíncrona, mantendo a página aberta. Esse método coleta todos os dados que serão passados para a requisição, prepara-os e registra na resposta um método de *callback* para ser chamado quando a requisição é terminada. Esse método de *callback* pode ser usado para alterarmos alguma informação na página já existente e que não precisou ser carregada novamente.

Agora que já importamos a biblioteca de AJAX, basta alterarmos nosso *link* para que a requisição seja feita de maneira assíncrona. Para isso, basta adicionar dentro do `commandLink` a Tag `f:ajax`:

```
<h:commandLink action="#{produtoBean.remove(produto)}" value="Remover">
    <f:ajax />
</h:commandLink>
```

Pronto! Agora a requisição para exclusão do produto será feita através de uma requisição assíncrona. Porém, se deixarmos o código como está, o produto será removido, mas a tabela não será atualizada. Para resolver esse problema, vamos avisar ao JSF que tudo que está no formulário será renderizado novamente após a requisição, setando o atributo `render`. O objetivo do atributo `render` é indicar para o JSF qual parte da sua tela deverá ser renderizada novamente após a execução do AJAX.

Nesse caso, vamos indicar que tudo que existe dentro do formulário que envolve o link clicado deverá ser renderizado. Para isso, utilizaremos o valor `@form` que referencia o form que envolve o componente que disparou o evento.

```
<h:commandLink action="#{produtoBean.remove(produto)}" value="Remover">
    <f:ajax render="@form" />
</h:commandLink>
```

É possível também indicar que qualquer outro elemento da tela deve ser renderizado após a execução da requisição via AJAX. Para isso, basta identificar o elemento através de um `id` e referenciar esse `id` no atributo `render` da Tag AJAX precedido de um `:`. Nesse caso, se quiséssemos atualizar um elemento cujo `id` é `formProduto` faríamos:

```
<f:ajax render=":formProduto" />
```

Ou podemos renderizar mais de um elemento apontando seus `ids` separados por espaço:

```
<f:ajax render=":formProduto :outroFormulario" />
```

É importante ressaltar que a sintaxe `:formProduto` usa um ID absoluto. Ou seja, na hierarquia da tela, logo abaixo do `h:body`, deve ter algum elemento que possui o ID `formProduto`. Assim podemos definir um caminho para chegar algum componente encadeado: `:formProduto:campoNome`. Também é possível usar o um ID relativo, por exemplo `campoNome` (sem `:`). Nesse caso o JSF procura um ID dentro do mesmo formulário.

Um outro ponto interessante é quando precisamos submeter dados de um formulário para que uma `action` seja executada, que é o caso do cadastro de produtos. Nesse caso, ao adicionarmos AJAX ao botão `Gravar` com a Tag `f:ajax` os dados não são submetidos, consequentemente quando a ação é executada são gravados dados nulos no banco de dados. Para contornarmos isso, precisamos indicar quais dados queremos submeter para nossa `action`. Isso é feito através do atributo `execute` da Tag `f:ajax`, onde podemos indicar qual o formulário deve ser submetido:

```
<h:form id="formProduto">  
    <!-- aqui estão os inputs -->  
    <h:commandButton value="Gravar" action="#{produtoBean.grava}">  
        <f:ajax execute="@form" />  
    </h:commandButton>  
  
</h:form>
```

4.4 EXERCÍCIOS: REMOÇÃO DE PRODUTOS COM AJAX

1. Altere a remoção de produtos para funcionar via AJAX. Adicione o componente `<f:ajax>` ao link:

```
<h:commandLink action="#{produtoBean.remove(produto)}" value="Remover">  
    <f:ajax render="@form" />  
</h:commandLink>
```

2. Salve a alteração que foi feita e acesse no browser: `http://localhost:8080/fj26-notas-fiscais/produto.xhtml`

Remova algum produto. Repare que a tela inteira não foi renderizada novamente, só o item que foi removido da tabela. Será que teve alguma requisição sendo feita? Verifique no console do navegador.

Agora é a melhor hora de respirar mais tecnologia!



Se você está gostando dessa apostila, certamente vai aproveitar os **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum. Programação, Mobile, Design, Infra, Front-End e Business! Ex-aluno da Caelum tem 15% de desconto, siga o link!

[Conheça a Alura Cursos Online.](#)

4.5 PARA SABER MAIS: A TAG

A tag `f:ajax` assim como a maioria das tags utilizadas no JSF devem ser colocadas ou dentro de uma tag ou envolvendo outras Tags.

O código a seguir mostra um exemplo de como envolver tags e aplicar as capacidades do AJAX em cada uma delas individualmente:

```
<f:ajax>
    <h:inputText value="#{produtoBean.produto.nome}" />
    <h:commandButton value="Gravar" action="#{produtoBean.grava}" />
</f:ajax>
```

O código anterior faz exatamente o mesmo que o código a seguir:

```
<h:inputText value="#{produtoBean.produto.nome}">
    <f:ajax event="change" />
</h:inputText>
<h:commandButton value="Gravar" action="#{produtoBean.grava}">
    <f:ajax event="click" />
</h:commandButton>
```

Ou seja, o `inputText` será atualizado no servidor via AJAX quando seu valor for mudado pelo cliente e o campo perder o foco. Já o `commandButton`, disparará sua ação quando for clicado.

A tabela abaixo mostra alguns atributos que podemos colocar dentro da tag `f:ajax`:

- **execute**: representa os valores que serão submetidos na requisição através dos *client ids* separados por espaço;
- **render**: determinam quais componentes de árvore serão renderizados novamente;
- **listener**: invoca algum método com a assinatura **void método(AjaxBehaviorEvent e)** no *managed bean*;
- **immediate**: Boolean. Se `false` a ação ocorrerá na fase *Invoke App.*, se `true` ocorrerá na *Apply Req. Values*;

- **disabled**: Booleano. Se for `true` desliga as funcionalidades do AJAX;
- **event**: indica qual evento invocará a requisição AJAX. Exemplos: `mouseout`, `dblclick`, `change`, `focus`, etc.

A lista abaixo mostra algumas palavras especiais que podem ser usados dentro desses dos atributos `execute` e `render`.

- **@all** :
 - No `execute` : Todo componente da página é enviado na requisição;
 - No `render` : Todo componente será renderizado novamente após a requisição;
- **@form** :
 - No `execute` : Submete todos componentes da tag `<h:form>` em que a tag `<f:ajax>` está;
 - No `render` : Renderiza após a requisição todos componentes contidos na tag `<h:form>` em que a tag `<f:ajax>` está;
- **@none** :
 - No `execute` : Nenhum componente é submetido;
 - No `render` : Não renderiza nenhum componente após a requisição;
- **@this** :
 - No `execute` : Submete apenas o componente ou componentes em que a tag `<f:ajax>` está aplicada;
 - No `render` : Renderiza após a requisição apenas o componente ou componentes em que a tag `<f:ajax>` está aplicada;

4.6 EXERCÍCIOS: SUBMETENDO FORMULÁRIOS COM AJAX

1. Baseado nas informações das tabelas anteriores, faça com que o formulário de cadastro e alteração dos produtos seja submetido via AJAX.

Dicas: Lembre-se de que é possível referenciar elementos através dos seus `ids` e que, para submeter dados via AJAX, é necessário utilizar o atributo `execute` da Tag `f:ajax`.

2. Faça também a alteração de produtos com AJAX.

Dica: o *link* de "Alterar" da tabela deverá disparar a renderização do formulário de cadastro que será carregado com os dados.

APLICANDO LAYOUTS À APLICAÇÃO COM CSS

"Do sublime ao ridículo, há nada mais que um passo." -- Napoleão Bonaparte

5.1 MELHORANDO A INTERFACE GRÁFICA

Agora que desenvolvemos as funcionalidades básicas para que os funcionários da UberDist consigam cadastrar os produtos, precisamos adicionar um *layout* simples para que a visualização da aplicação fique mais agradável. Além disso, incluiremos também o logo da empresa na página e um rodapé com os dados da empresa.

Para fazermos o estilo da nossa aplicação utilizaremos CSS (*Cascading Stylesheet*). Ao usar JSF, é essencial o uso de CSS para estilo já que temos pouco controle do HTML gerado. É a melhor forma de especificar os *layouts*.

APRENDENDO CSS A FUNDO

É importante lembrar que o foco desse capítulo não é criar o CSS e sim integrar um CSS já pronto com a aplicação. CSS é ensinado em detalhes junto com HTML e Javascript no curso Desenvolvimento Web com HTML, CSS e JavaScript (WD-43).

Editora Casa do Código com livros de uma forma diferente



Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.

Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.](#)

5.2 INTEGRANDO CSS COM O JSF

O primeiro passo para utilizarmos CSS em nossas páginas é importá-lo nas páginas que o usaremos. Para isso, podemos utilizar o componente do JSF chamado `h:outputStylesheet` .

Por padrão, no JSF 2.0 os arquivos de CSS devem estar dentro de um diretório chamado `resources` na sua aplicação Web ou então dentro de um diretório chamado `META-INF/resources` .

Considerando que exista um arquivo chamado `style.css` dentro de um diretório chamado `resources/css` podemos importar esse arquivo da seguinte forma no JSF:

```
<h:outputStylesheet library="css" name="style.css" />
```

O atributo `library` representa o diretório dentro da pasta `resources` na qual o arquivo se encontra, e o nome do arquivo é informado através do atributo `name` .

Agora que já importamos o CSS para a nossa página, precisamos utilizar as definições contidas dentro dele nos nossos componentes. As classes definidas dentro do CSS podem ser utilizadas nos componentes visuais do JSF através do atributo `styleClass` existente nos componentes.

Podemos também utilizar um estilo CSS em um `h:outputText` :

```
<h:outputText value="Algum Texto" styleClass="classeCSS" />
```

Também é possível definir os `ids` dos componentes através do atributo `id` . Dessa forma, podemos referenciar os estilos CSS da nossa aplicação através do `id` dos componentes também.

```
<h:outputText value="Algum Texto" styleClass="classeCSS"
               id="texto"/>
```

Um outro ponto no qual é bastante comum precisarmos utilizar estilos são nas linhas da tabela, na qual cada linha possui uma cor alternadamente, dando o efeito de "zebrado". Isso pode ser feito através

do atributo `rowClasses`. Nele é possível informar diversas classes CSS que serão utilizadas para formar as linhas da tabela gerada, separando-os por vírgula.

Dessa forma, considerando que em nosso CSS tenhamos duas classes, uma chamada `par` e outra chamada `impar`, poderíamos criar o nosso `dataTable` da seguinte forma:

```
<h: dataTable value="#{produtoBean.produtos}" var="produto"
    rowClasses="par,impar">
```

COMPATIBILIDADE ENTRE NAVEGADORES

Um problema comum na Web é garantir que o Site funcione em todos os principais navegadores. O JSF ajuda muito com isso ao encapsular boa parte do trabalho com HTML e JavaScripts. Mas se vamos usar um *layout* customizado com CSS, poderemos ter problemas com isso.

O *layout* usado nesse curso suporta todos os principais navegadores modernos. Para ficar mais simples, o CSS não foi pensado em compatibilidade com navegadores antigos. Mas é bom ficar atento ao requisito de cada sistema para saber quais navegadores devem ser suportados.

5.3 ADICIONANDO IMAGENS NA APLICAÇÃO

Também é possível adicionarmos imagens às páginas. Para isso, basta utilizar a Tag `h:graphicImage`. Essa Tag vai gerar em HTML uma Tag `img`. Para indicar qual a imagem que será utilizada, é preciso passar um atributo `name` para a Tag, indicando qual o nome do arquivo que será utilizado. Além disso, também é possível indicar em qual diretório a imagem estará, dentro do diretório `resources`. Para isso utiliza-se o atributo `library`, como abaixo:

```
<h:graphicImage library="imagens" name="logo.png" />
```

5.4 EXERCÍCIOS: MELHORANDO A INTERFACE GRÁFICA

1. Vamos aplicar alguns estilos melhores à interface de nossa aplicação. Faça esse exercício com calma, preste bastante atenção na digitação e nos enunciados.
 - No diretório `WebContent` crie uma pasta chamada **resources**.
 - Vá ao Desktop e entre na pasta `Caelum/26`.
 - Copie os diretórios `css` e `imagens` para dentro da pasta `resources` que você acabou de criar (copie os diretórios mesmo, não só os arquivos dentro deles).
2. Importe o arquivo `.css` para a nossa página. Para isso, no arquivo `produto.xhtml`, dentro da Tag `h:head` adicione:

```

<h:head>
    <title>Sistema de Notas Fiscais - UberDist</title>
    <h:outputStylesheet library="css" name="style.css" />
</h:head>

```

3. **Inmediatamente após** da Tag `h:body` adicione uma `div` para adicionarmos o logo da empresa:

```

<h:body>
    <div id="cabecalho">
        <h:graphicImage library="imagens" name="logo-uber.png"
            id="logoCompany" />
    </div>

    <!-- continuação da página -->
</h:body>

```

Atualize a página no navegador, você já começará a ver algum resultado.

4. **Abaixo do cabeçalho** que foi recém adicionado, crie uma outra `div` para envolver o formulário e a tabela, fechando-a antes do fechamento da Tag `h:body`.

```

<div id="conteudo">

    <!-- formulário com os campos e a tabela -->

</div>

```

Atualize novamente a página.

5. Nossa *designer* já preparou o *layout* para ser *tableless*, seguindo as boas práticas. Portanto, precisamos remover o uso do componente `<h:panelGrid>`, usado antes para um *layout* simples. Esse componente gera tabelas HTML, o que não é recomendado.

Remova a abertura e o fechamento da tag `h:panelGrid`, atualize a página e veja o resultado.

6. No seu `dataTable` adicione os atributos `styleClass` e `rowClasses` como abaixo:

```

<h:dataTable value="#{produtoBean.produtos}" var="produto"
    styleClass="dados" rowClasses="impar,par">

```

Atualize novamente a página.

7. **Adicione** um novo bloco para o rodapé antes do fechamento da Tag `h:body`.

```

<div id="rodape">
    Copyright 2013.
    Todos os Direitos reservados a ---Coloque seu nome aqui---.
</div>

```

Atualize novamente a página.

8. Para separarmos melhor as duas partes da tela (a parte superior com o formulário de cadastro e a parte inferior com a listagem) vamos adicionar um subtítulo para a lista também. Antes do `form` que envolve a listagem, adicione:

<h2>Listagem de Produtos</h2>

9. Para agruparmos os componentes do nosso formulário de cadastro dos produtos, vamos utilizar a Tag `fieldset` envolvendo os elementos do formulário abaixo do título da página:

```
<h:form>
    <!-- aqui fica o titulo h2 -->

    <fieldset>
        <legend>Dados do Produto</legend>

        <!-- aqui estão os campos do formulário -->
    </fieldset>
</h:form>
```

10. Acesse a página pelo navegador. O resultado final deverá ser similar à imagem seguinte:

The screenshot shows a web application interface. At the top, there is a blue header bar with the word "UBERDIST" in white. Below it, the main content area has a title "Novo Produto". Underneath, there is a section titled "Dados do Produto" containing three input fields: "Nome", "Descrição", and "Preço", each with a corresponding text input box. Below these fields is a "Gravar" button. Further down, there is another section titled "Listagem de Produtos" which displays a table with four columns: Nome, Descrição, Preço, Remover, and Alterar. The table contains four rows of data. At the bottom of the page, there is a blue footer bar with the text "Copyright 2010. Todos os Direitos reservados a —Seu nome aqui—.".

Nome	Descrição	Preço	Remover	Alterar
aaaa		0.0	Remover	Alterar
		0.0	Remover	Alterar
asdasd		0.0	Remover	Alterar
asdasd		0.0	Remover	Alterar

Já conhece os cursos online Alura?



A Alura oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Programação, Front-end, Mobile, Design & UX, Infra e Business, com um plano que dá acesso a todos os cursos. Ex aluno da Caelum tem 15% de desconto neste link!

[Conheça os cursos online Alura.](#)

ENTENDENDO JSF A FUNDO

"A ciência nunca resolve um problema sem criar mais dez" -- George Bernard Shaw

Ao término desse capítulo, você será capaz de:

- Descrever o que é a árvore de componentes do JSF;
- Descrever as fases do processamento da requisição no JSF e o impacto de cada uma delas.

6.1 REDUZINDO A COMPLEXIDADE DE OUTRAS APIs

Quando estamos começando a aprender a trabalhar com a Web em Java, um dos primeiros assuntos que aprendemos são as `Servlets`, cujo objetivo é abstrair a complexidade de lidarmos diretamente com o protocolo HTTP. No entanto, a própria API de `Servlets` ainda nos faz trabalhar de forma não natural em alguns momentos, como por exemplo, ao termos que tratar o `HttpServletRequest` para buscar os dados da requisição como uma `String` e termos que fazer a conversão manualmente.

Pelo pouco que já vimos de JSF, percebemos que toda aquela complexidade é muito reduzida.

Sempre que uma requisição é submetida através de uma aplicação desenvolvida em JSF, diversas tarefas são executadas internamente pelo framework, para que seja possível que a requisição seja processada corretamente. Para aproveitar ao máximo os recursos do JSF, é imprescindível que se conheça quais são as tarefas executadas internamente pelo seu motor. Essa sequência de tarefas é o que chamamos de **Ciclo de Vida de uma requisição JSF**.

Saber inglês é muito importante em TI

galandra

O **Galandra** auxilia a prática de inglês através de *flash cards* e *spaced repetition learning*. Conheça e aproveite os preços especiais.

[Pratique seu inglês no Galandra.](#)

6.2 A ÁRVORE DE COMPONENTES

A base para todo o processamento que o JSF faz internamente é uma estrutura que representa no servidor todos os componentes visuais que foram utilizados nos arquivos `xhtml` : **a árvore de componentes.**

O armazenamento da árvore de componentes é um dos pontos mais críticos ao se utilizar o JSF, pois pode fazer com que a sua aplicação seja ou não escalável.

O ponto chave é que a árvore de componentes ao ser mantida no servidor, é armazenada na sessão do usuário, e dessa forma, ocupando mais memória no servidor. Uma outra alternativa é armazenar a árvore de componentes no lado do cliente, e dessa forma ela é passada a cada requisição através de campos do tipo `hidden`, onerando a banda do servidor.

Ambas as abordagens possuem pontos positivos e negativos. No caso de armazenar a árvore de componentes na sessão, o ponto negativo é o fato de ocupar memória do servidor. E como a árvore de componentes é gerada para todos os usuários, esse consumo facilmente fica grande. O ponto positivo é que, para o cliente, é imperceptível. Ao armazenar a árvore de componentes no cliente, a vantagem é que deixa de sobrecarregar o servidor. No entanto, o ponto negativo é que, ao ter que enviar a árvore de componentes durante cada requisição, as requisições tendem a demorar mais tempo para serem feitas, dado que mais dados serão transmitidos.

O ARMAZENAMENTO DA ÁRVORE DE COMPONENTES NO JSF 2

No JSF 2, o armazenamento da árvore de componentes foi melhorado com relação as versões anteriores, com o intuito de utilizar menos recursos.

Nas versões anteriores, todos os componentes eram armazenados completamente, o que por muitas vezes causava problemas de performance. Na versão 2, é armazenado apenas os estados que foram alterados dos componentes, consumindo menos recursos.

6.3 ALTERANDO A FORMA DE ARMAZENAMENTO DA ÁRVORE DE COMPONENTES

Por padrão, o JSF armazena a árvore de componentes na sessão do usuário. Esse comportamento pode ser facilmente alterado para armazená-la no cliente. Para isso, basta editar o seu `web.xml` e adicionar o seguinte parâmetro:

```
<context-param>
    <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
    <param-value>client</param-value>
    <!-- aqui é possível colocar a opção "server" (padrão) -->
```

</context-param>

6.4 COMPREENDENDO O CICLO DE VIDA DE UMA REQUISIÇÃO DO JSF

Quando uma requisição é enviada para uma aplicação JSF, dispare-se um ciclo que é o mesmo para todas as requisições que são feitas. Esse ciclo é extremamente importante para o JSF, pois possui diversas fases críticas.

São 6 (seis) as fases do **ciclo de vida do JSF**:

- *Restore View* (Restauração da View)
- *Apply Request Values* (Aplicar os valores da requisição)
- *Process Validations* (Processar validações)
- *Update Model Values* (Atualizar valor dos modelos)
- *Invoke Application* (Invocar a lógica)
- *Render Response* (Renderizar a resposta)

Vamos explicar passo a passo essas fases.

Seus livros de tecnologia parecem do século passado?



Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**. Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

[Casa do Código, Livros de Tecnologia.](#)

6.5 PRIMEIRA FASE: RESTAURAÇÃO DA VIEW

A primeira ação que o JSF toma ao receber uma requisição é recuperar o estado da árvore de componentes anterior. Caso seja uma primeira requisição, o JSF vai criar uma nova árvore de componentes para que seja usada futuramente. O estado da árvore de componentes é recuperado de acordo com o local especificado para armazená-la. No caso de ser armazenado na sessão, internamente o JSF faz algo parecido com:

```
ArvoreDeComponentesDoJSF arvore =
    (ArvoreDeComponentesDoJSF) session.getAttribute("arvore");
```

6.6 SEGUNDA FASE: APlicar os Valores da Requisição

Agora que o JSF já restaurou a árvore de componentes, ele recupera os valores que vieram na requisição e aplica-os aos seus respectivos componentes. No entanto, esses valores ainda estão representados como `Strings`, ou seja, eles ainda estão sem nenhuma significância para nossos modelos. Pensando em código, seria o equivalente a:

```
String paramNome = request.getParameter("produto.nome");
String paramPreco = request.getParameter("produto.preco");
```

6.7 TERCEIRA FASE: CONVERTER E PROCESSAR VALIDAÇÕES

Após recuperar os valores, e associá-los com os componentes corretos, o JSF os converte para o tipo adequado que o componente espera receber (ou faz conversões customizadas, que veremos mais adiante no curso). Após a conversão, o valor já estará com o tipo correto e portanto ele estará pronto para ser validado. Atingimos esse comportamento com o código abaixo:

```
int idade = Integer.parseInt(paramIdade);
if (idade <= 18) {
    throw new ValidacaoDoJSFException("Idade é menor do que 18 anos");
}
```

Agora é a melhor hora de respirar mais tecnologia!



Se você está gostando dessa apostila, certamente vai aproveitar os **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum. Programação, Mobile, Design, Infra, Front-End e Business! Ex-aluno da Caelum tem 15% de desconto, siga o link!

[Conheça a Alura Cursos Online.](#)

6.8 QUARTA FASE: ATUALIZAÇÃO DOS MODELOS

Após a conversão e a validação serem feitas corretamente, os valores estão prontos para serem aplicados aos modelos aos quais estão associados. Dessa forma, as propriedades existentes dentro do seu *Managed Bean* serão populadas de acordo com as informações que vieram na requisição. O JSF sabe para quais objetos enviar os valores através do *binding* que foi feito na criação das telas. O que o JSF faz internamente é criar um objeto e populá-lo, similar com o código abaixo:

```
Produto produto = produtoBean.getProduto();
produto.setPreco(precoProduto);
```

```
produto.setDescricao(descricaoProduto);
```

6.9 QUINTA FASE: INVOCAÇÃO DA LÓGICA

Com o *Managed Bean* devidamente populado, o JSF agora pode, com segurança, invocar a lógica determinada pela requisição. A navegação para outras páginas (que aprenderemos em seguida) também é feita nessa fase, logo após a execução da lógica. Portanto, a partir de uma instância do *Managed Bean*, o JSF invoca o método desejado:

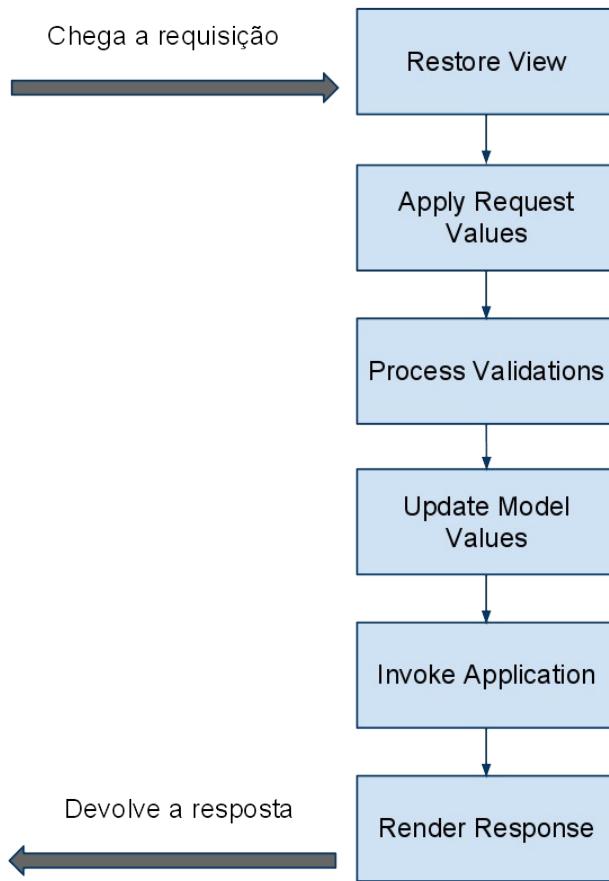
```
produtoBean.salva();
```

6.10 SEXTA FASE: RENDERIZAR RESPOSTA

Agora que o processamento foi feito, o último passo é devolver a resposta para o usuário. Portanto, é nesse passo que o XHTML de resposta é renderizado para o usuário. Nesta etapa ocorrem basicamente duas coisas: a primeira é a navegação, onde o JSF confere se a resposta que o sistema tem que gerar é a árvore atual ou se tem que ir para uma página diferente. Depois de descobrir qual árvore será a resposta, o JSF começa a escrever o HTML de resposta. Uma última tarefa extremamente importante que o JSF executa nesse instante é guardar a árvore de componentes atual para utilização nas requisições futuras. Para isso, ele passa componente por componente, avaliando as diferenças entre seus estados iniciais e os atuais, armazenando as diferenças. Em resumo, o JSF faz algo como:

```
arvore = requisicao.geraArvore();
session.setAttribute("arvore", arvore);
```

Por fim, a sequência da execução das fases pode ser visualizada no diagrama a seguir:



Editora Casa do Código com livros de uma forma diferente



Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.

Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.](#)

6.11 EXERCÍCIOS: FASES DO JSF

1. Para podermos visualizar que o JSF executa internamente diversas tarefas, vamos forçar para que aconteça algum problema nessa tarefa.

Tente adicionar um novo produto com um preço que contenha alguma letra. Repare que o mesmo não vai ser adicionado no banco de dados, por quê?

2. Adicione logo no começo do seu método grava do ProdutoBean um System.out com uma mensagem de log:

```
System.out.println("Será que vai passar por aqui?");
```

Teste de novo e observe. Passou? Se não passou, é sinal de que alguma das fases não está sendo invocada. Qual?

3. O problema é que digitamos o valor do produto de forma inválida. A requisição está sendo interrompida na **fase de validação** (terceira fase). Dessa forma, a atualização dos modelos e a invocação da lógica não são executadas.

Futuramente, vamos aprender a tratar problemas de conversão e validação. É importante sempre tratar a validação no JSF, ou você pode perder-se, já que ao clicar em um botão nada acontece, nem mesmo uma mensagem de erro.

4. (avançado, opcional) Troque no seu *web.xml* para que a árvore de componentes seja guardada no cliente em vez do servidor. Para que isso funcione corretamente, suas classes envolvidas precisam implementar a interface `java.io.Serializable`. Depois, acesse o formulário do seu sistema e repare um atributo *hidden* que está sendo passado em toda requisição. O que é esse atributo?

```
<context-param>
    <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
    <param-value>client</param-value>
</context-param>
```

ATENÇÃO: Lembre-se de voltar à configuração para armazenar o estado no servidor.

6.12 DEBUGANDO O CICLO DE VIDA DO JSF COM UM PHASELISTENER

O JSF possui um recurso que nos permite executar algum código antes e/ou depois que alguma fase ocorra. Esse recurso é conhecido como **PhaseListener**.

Implementá-lo é simples, basta seguirmos a interface `PhaseListener` do próprio JSF e teremos 3 (três) métodos para implementar: `beforePhase`, `afterPhase` e `getPhaseId`. Nos dois primeiros colocamos o código que queremos executar antes ou depois de alguma fase e o terceiro método devolve qual fase do JSF nosso listener irá escutar, ou podemos devolver `PhaseId.ANY_PHASE` para indicar que queremos ser notificados antes e depois de todas as fases. Dessa forma, para termos um simples *Listener* que imprima quando uma fase começou e terminou, podemos criar uma classe chamada `CicloDeVidaListener`:

```
public class CicloDeVidaListener implements PhaseListener {
    public void beforePhase(PhaseEvent event) {
```

```

        System.out.println("Antes da fase: " + event.getPhaseId());
    }

    public void afterPhase(PhaseEvent event) {
        System.out.println("Depois da fase: " + event.getPhaseId());
    }

    public PhaseId getPhaseId() {
        return PhaseId.ANY_PHASE;
    }
}

```

Apenas reforçando que no método `getPhaseId`, ao devolvermos `PhaseId.ANY_PHASE`, indicamos que o *Listener* deve ser disparado em todas as fases. Com isso, todas as fases imprimirão seu início e seu fim. Por fim, é necessário habilitar o `PhaseListener` em nossa aplicação. Para isso, basta declará-lo no arquivo `faces-config.xml`:

```

<faces-config ...>
    <lifecycle>
        <phase-listener>
            br.com.caelum.notasfiscais.listener.CicloDeVidaListener
        </phase-listener>
    </lifecycle>
</faces-config>

```

Agora, ao acessarmos a página `produtos.xhtml` teremos a seguinte saída:

```

INFO: Antes da fase: RESTORE_VIEW 1
INFO: Depois da fase: RESTORE_VIEW 1
INFO: Antes da fase: RENDER_RESPONSE 6
INFO: Carregando produtos...
INFO: [EL Info]: 2013-08-16 12:43:40.625--ServerSession(1237141914)--
  EclipseLink, version: Eclipse Persistence Services -
  2.5.0.v20130507-3faac2b
INFO: [EL Info]: connection: 2013-08-16 12:43:40.914--ServerSession
  (1237141914)-- file:/Users/jsfxxxx/glassfish4-web/glassfish/domains/
  domain1/eclipseApps/fj26-notas-fiscais/WEB-INF/classes/_notas login
  successful
INFO: [EL Fine]: sql: 2013-08-16 12:43:40.936--ServerSession(1237141914)--
  Connection(612921479)--SELECT ID FROM ITEM WHERE ID <> ID
INFO: [EL Fine]: sql: 2013-08-16 12:43:40.973--ServerSession(1237141914)--
  Connection(612921479)--SELECT ID FROM NOTAFISCAL WHERE ID <> ID
INFO: [EL Fine]: sql: 2013-08-16 12:43:41.057--ServerSession(1237141914)--
  Connection(612921479)--SELECT ID FROM PRODUTO WHERE ID <> ID
INFO: [EL Fine]: sql: 2013-08-16 12:43:41.096--ServerSession(1237141914)--
  Connection(612921479)--SELECT ID FROM USUARIO WHERE ID <> ID
INFO: [EL Fine]: sql: 2013-08-16 12:43:41.102--ServerSession(1237141914)--
  Connection(612921479)--ALTER TABLE ITEM ADD CONSTRAINT FK_ITEM_PRODUTO_ID
  FOREIGN KEY (PRODUTO_ID) REFERENCES PRODUTO (ID)
INFO: [EL Fine]: sql: 2013-08-16 12:43:41.306--ServerSession(1237141914)--
  SELECT 1
INFO: [EL Fine]: sql: 2013-08-16 12:43:41.37--ServerSession(1237141914)--
  Connection(612921479)--SELECT ID, DESCRICAO, NOME, PRECO FROM PRODUTO
INFO: Depois da fase: RENDER_RESPONSE 6

```

Repare que as fases 2, 3, 4 e 5 não foram executadas. Isso acontece, pois nenhum ação (submissão de formulário) foi realizada e, com isso, não há dados para serem recuperados nem convertidos.

No entanto, se tentarmos gravar um produto novo, teremos a seguinte saída:

```

INFO: Antes da fase: RESTORE_VIEW 1
INFO: Depois da fase: RESTORE_VIEW 1
INFO: Antes da fase: APPLY_REQUEST_VALUES 2
INFO: Depois da fase: APPLY_REQUEST_VALUES 2
INFO: Antes da fase: PROCESS_VALIDATIONS 3
INFO: Depois da fase: PROCESS_VALIDATIONS 3
INFO: Antes da fase: UPDATE_MODEL_VALUES 4
INFO: Depois da fase: UPDATE_MODEL_VALUES 4
INFO: Antes da fase: INVOKE_APPLICATION 5
INFO: [EL Fine]: sql: 2013-08-16 12:48:42.402--ClientSession(445407382)--
    Connection(612921479)--SELECT LAST_INSERT_ID()
INFO: [EL Fine]: sql: 2013-08-16 12:48:42.41--ServerSession(1237141914)--
    Connection(612921479)--SELECT ID, DESCRICAO, NOME, PRECO FROM PRODUTO
INFO: Depois da fase: INVOKE_APPLICATION 5
INFO: Antes da fase: RENDER_RESPONSE 6
INFO: Depois da fase: RENDER_RESPONSE 6

```

Repare que, agora, todas as fases são executadas e, na quinta fase, o insert é realizado, ou seja, a ação foi invocada. Por fim, podemos tentar também visualizar o que acontece ao tentarmos cadastrar um produto cujo valor falhe na conversão. O resultado será algo similar ao seguinte:

```

INFO: Antes da fase: RESTORE_VIEW 1
INFO: Depois da fase: RESTORE_VIEW 1
INFO: Antes da fase: APPLY_REQUEST_VALUES 2
INFO: Depois da fase: APPLY_REQUEST_VALUES 2
INFO: Antes da fase: PROCESS_VALIDATIONS 3
INFO: Depois da fase: PROCESS_VALIDATIONS 3
INFO: Antes da fase: RENDER_RESPONSE 6
INFO: Carregando produtos...
INFO: [EL Fine]: sql: 2013-08-16 12:56:19.253--ServerSession(477205733)--
    Connection(825461719)--SELECT ID, DESCRICAO, NOME, PRECO FROM PRODUTO
INFO: Depois da fase: RENDER_RESPONSE 6

```

Note que, depois da fase de validações, o JSF pulou diretamente para a renderização da resposta.

6.13 EXERCÍCIOS: OBSERVANDO AS FASES DO JSF

1. Crie um `PhaseListener` do JSF que apenas loga a invocação de cada fase (antes e depois). Faça a classe `CicloDeVidaListener` no pacote `br.com.caelum.notasfiscais.listener`:

```

public class CicloDeVidaListener implements PhaseListener {

    public void afterPhase(PhaseEvent event) {
        System.out.println("Depois da fase: " + event.getPhaseId());
    }

    public void beforePhase(PhaseEvent event) {
        System.out.println("Antes da fase: " + event.getPhaseId());
    }

    public PhaseId getPhaseId() {
        return PhaseId.ANY_PHASE;
    }
}

```

Configure sua classe no `faces-config.xml` do JSF:

```
<faces-config ...>
```

```
<lifecycle>
    <phase-listener>
        br.com.caelum.notasfiscais.listener.CicloDeVidaListener
    </phase-listener>
</lifecycle>
</faces-config>
```

2. Teste sua aplicação de diferentes formas e observe no console do Eclipse a saída em cada um dos casos.
 - Quais fases executam na primeira vez que acessamos a página?
 - Submeta um `Produto` com dados corretos; quais fases são executadas?
 - Submeta um `Produto` com valor inválido (usando uma letra); o que acontece?
3. Remova a configuração do listener no XML para não poluir a saída do console nos próximos exercícios.

Já conhece os cursos online Alura?



A Alura oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Programação, Front-end, Mobile, Design & UX, Infra e Business, com um plano que dá acesso a todos os cursos. Ex aluno da Caelum tem 15% de desconto neste link!

[Conheça os cursos online Alura.](#)

LOGIN E NAVEGAÇÃO

"O primeiro problema para todos, homens e mulheres, não é aprender, mas desaprender" -- Gloria Steinem

7.1 CRIANDO A FUNCIONALIDADE DE LOGIN

Precisamos fazer a funcionalidade de login para o sistema da UberDist. Os usuários serão armazenados no banco de dados, portanto, precisaremos de uma classe mapeada com as anotações da JPA.

```
@Entity
public class Usuario {

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

    private String login;

    private String senha;

    // getters e setters

}
```

Para desenvolvemos a tela de login, utilizaremos os mesmos conceitos aprendidos até o momento. Portanto, o primeiro passo é termos o arquivo XHTML para desenvolvemos a tela, que vamos chamar de `login.xhtml`.

Para desenvolvemos o formulário de login, utilizaremos os componentes do JSF, entre eles o componente que renderiza um campo para digitar senhas, o `h:inputSecret`. Dessa forma, a página de login terá o seguinte código:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">

    <h:body>
        <h:form id="formlogin">
            <h:panelGrid columns="2" styleClass="campos">

                <h:outputLabel value="Login:" for="login" />
                <h:inputText id="login" />

                <h:outputLabel value="Senha:" for="senha" />
                <h:inputSecret id="senha" />
            
```

```

        <h:commandButton value="Efetuar Login" />

        </h:panelGrid>
    </h:form>
</h:body>
</html>

```

Ao clicarmos no botão "Efetuar Login", queremos validar se o login e senha informados estão cadastradas no banco de dados. Para isso, será preciso fazer uma consulta no banco de dados para verificar se os dados existem. Uma forma de fazer essa consulta é criando uma classe DAO para encapsular a pesquisa, dessa forma teríamos:

```

public class UsuarioDao {

    public boolean existe(Usuario usuario) {
        EntityManager em = new JPAUtil().getEntityManager();
        em.getTransaction().begin();

        Query query = em.createQuery("from Usuario u where u.login = "
            + ":pLogin and u.senha = :pSenha");
        query.setParameter("pLogin", usuario.getLogin());
        query.setParameter("pSenha", usuario.getSenha());

        boolean encontrado = !query.getResultList().isEmpty();

        em.getTransaction().commit();
        em.close();

        return encontrado;
    }
}

```

O próximo passo é fazer o *binding* dos componentes com um *Managed Bean* que receberá os dados para executar o processamento. Para isso, vamos criar um *Managed Bean* chamado `LoginBean`. Esse *Managed Bean* deverá ter um atributo do tipo `Usuario` como abaixo:

```

@ManagedBean
public class LoginBean {
    private Usuario usuario = new Usuario();

    // apenas o getter
}

```

Podemos fazer o *binding* com o atributo `usuario`. Para isso, basta adicionar o atributo `value` nos campos `login` e `senha` do `login.xhtml`.

```

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">

    <h:head>
        <title>Sistema de Notas Fiscais - UberDist</title>
    </h:head>

    <h:body>
        <h:form id="formlogin">
            <h:panelGrid columns="2" styleClass="campos">

```

```

<h:outputLabel value="Login:" for="login" />
<h:inputText id="login" value="#{loginBean.usuario.login}" />

<h:outputLabel value="Senha:" for="senha" />
<h:inputSecret id="senha" value="#{loginBean.usuario.senha}" />

<h:commandButton value="Efetuar Login" />

</h:panelGrid>
</h:form>
</h:body>
</html>

```

Agora que já temos o DAO, precisamos criar o método do nosso *Managed Bean* que será invocado ao clicarmos no botão "Efetuar Login". Esse invocará o método existe da classe UsuarioDao para saber se o usuário existe mesmo.

Para isso, vamos criar um método na classe LoginBean chamado efetuaLogin que processará a requisição do usuário para fazer a validação e efetuar o login:

```

public void efetuaLogin() {
    UsuarioDao dao = new UsuarioDao();
    boolean loginValido = dao.existe(this.usuario);
    // vamos usar a variável loginValido futuramente
}

```

E também fazermos o *binding* do método efetuaLogin com o componente commandButton :

```
<h:commandButton value="Efetuar Login" action="#{loginBean.efetuaLogin}" />
```

Saber inglês é muito importante em TI

galandra O Galandra auxilia a prática de inglês através de *flash cards* e *spaced repetition learning*. Conheça e aproveite os preços especiais.

[Pratique seu inglês no Galandra.](#)

7.2 EXERCÍCIO: CRIANDO O FORMULÁRIO DE LOGIN

- Vamos criar o formulário de login e um *Managed Bean* para guardar as informações do usuário que vai fazer o login. Primeiramente, crie uma classe LoginBean no pacote br.com.caelum.notasfiscais.mb que tenha o método efetuaLogin :

```

@ManagedBean
public class LoginBean {
    private Usuario usuario = new Usuario();

```

```

public void efetuaLogin() {
    UsuarioDao dao = new UsuarioDao();
    boolean loginValido = dao.existe(this.usuario);
    System.out.println("O login era valido? " + loginValido);
}

public Usuario getUsuario() {
    return this.usuario;
}
}

```

2. Crie o arquivo `login.xhtml` na pasta `WebContent` do seu projeto com o seguinte conteúdo que criará um formulário contendo os campos para realizar o login:

```

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">

    <h:head>
        <title>Sistema de Notas Fiscais - UberDist</title>
    </h:head>

    <h:body>
        <div id="conteudo">
            <h2>Login no sistema</h2>
            <h:form id="formlogin">
                <h:panelGrid columns="2" styleClass="campos">

                    <h:outputLabel value="Login:" for="login" />
                    <h:inputText id="login"
                                 value="#{loginBean.usuario.login}" />

                    <h:outputLabel value="Senha:" for="senha" />
                    <h:inputSecret id="senha"
                                   value="#{loginBean.usuario.senha}" />

                    <h:commandButton value="Efetuar Login"
                                     action="#{loginBean.efetuaLogin}" />
                </h:panelGrid>
            </h:form>
        </div>
    </h:body>
</html>

```

3. Acesse a página pelo endereço: `http://localhost:8080/fj26-notas-fiscais/login.xhtml`, o resultado deve ser similar à imagem abaixo:



4. Acesse o banco de dados através do terminal e verifique se existe um usuário cadastrado:

```
mysql -u root fj26;  
  
select * from USUARIO;
```

Caso não exista, crie um executando um `insert` na tabela:

```
insert into USUARIO (login, senha) values ('admin', 'admin123');
```

5. Utilize o usuário cadastrado no banco de dados para fazer o login. O login será feito com sucesso?
Por quê? O que está errado?

7.3 NAVEGAÇÃO CONDICIONAL

Mas nesse ponto temos um problema. O que fazer se o login for válido e caso contrário, como devemos proceder?

O ideal é que, se o login for inválido, o usuário permaneça na mesma página e, caso seja validado com sucesso, ele deverá ser enviado para a tela principal do sistema.

7.4 NAVEGAÇÃO ENTRE TELAS

Dependendo do resultado da validação, precisamos decidir o caminho que o usuário seguirá no fluxo do sistema.

O componente `commandButton` aceita no seu atributo `action` um valor que indica qual é a página que deverá ser renderizada em seguida. Por exemplo, para indicarmos que queremos ir da página `login.xhtml` para a página `produto.xhtml` bastaria indicarmos no atributo `value` com `produto`:

```
<h:commandButton value="Efetuar Login" action="produto"/>
```

O que o JSF faz é procurar um arquivo `.xhtml` com o nome indicado no atributo `action`. Dessa forma, o arquivo `produto.xhtml` será renderizado. Esse nome indicado no atributo `action` também é conhecido como **outcome**.

Essa solução de informarmos o `outcome` no atributo `action` ainda não é suficiente para redirecionarmos o usuário após o login ser feito. Precisamos que, após a execução do método, o `outcome` adequado seja informado, ou seja, se quisermos ir para o `produto.xhtml` o `outcome` deverá ser `produto`, caso seja para ficar na mesma página, no caso a `login.xhtml` a saída deverá ser `login`.

Logo, o `outcome` nesse caso deve ser condicional. Conseguimos resolver isso pelo método `efetuaLogin` do `LoginBean`, através do retorno do método. O método pode retornar uma `String` indicando qual o `outcome` deverá ser utilizado. Dessa forma, poderíamos fazer:

```
public String efetuaLogin() {  
    UsuarioDao dao = new UsuarioDao();  
    boolean loginValido = dao.existe(this.usuario);
```

```

        if (loginValido) {
            return "produto";
        } else {
            this.usuario = new Usuario();
            return "login";
        }
    }
}

```

Seus livros de tecnologia parecem do século passado?



Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**. Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

[Casa do Código, Livros de Tecnologia.](#)

7.5 GUARDANDO DADOS EM ESCOPO DE SESSÃO

A funcionalidade de login está quase finalizada, mas ainda precisamos mostrar na página principal qual é o `login` do usuário que está usando o sistema atualmente.

Uma primeira ideia para resolver esse problema é simplesmente adicionar na `produto.xhtml` o `login` do usuário que está associado no `LoginBean`.

```
Logado como: #{loginBean.usuario.login}
```

Mas um detalhe bastante sutil vai impedir que isso funcione. Ao pressionar o link `Alterar` selecionando algum `Produto` para alteração, o `login` do usuário simplesmente desaparece, por quê?

O problema é que o `LoginBean` está no escopo de `request` e, dessa forma, os dados dele são perdidos nas requisições seguintes. Para resolvemos esse problema, basta alterarmos o escopo do `LoginBean` para `SessionScoped`.

```

@SessionScoped
@ManagedBean
public class LoginBean {
    // ...
}

```

A NAVEGAÇÃO NAS VERSÕES ANTERIORES DO JSF

Nas versões anteriores a 2.0, a navegação era feita através de declarações em XML. No caso, para dizer que a partir da página `login.xhtml` queremos ir para a página `produto.xhtml` quando o resultado do processamento é `produto` tínhamos que escrever o seguinte código:

```
<navigation-rule>
    <from-view-id>/login.xhtml</from-view-id>
    <navigation-case>
        <from-outcome>produto</from-outcome>
        <to-view-id>/produto.xhtml</to-view-id>
    </navigation-case>
</navigation-rule>
```

Isso, a princípio, pode parecer desagradável. No entanto, IDEs modernas proveem editores visuais para gerar o XML de navegação. Outro ponto importante é que uma aplicação JSF não possui muita navegação por se assemelhar com uma aplicação Desktop. No caso, é muito mais comum ao invés de trocarmos de página, renderizarmos ou não determinados componentes.

7.6 EXERCÍCIO: COMPLETANDO O LOGIN

- Precisamos fazer com que, ao se logar com sucesso, o usuário seja redirecionado para o `produto.xhtml` para isso, vamos fazer com que o método `efetuaLogin`, da classe `LoginBean`, retorne uma `String` indicando qual o caminho a seguir de acordo com o resultado da validação do login. Para isso, altere o seu método `efetuaLogin` como abaixo:

```
public String efetuaLogin() {
    UsuarioDao dao = new UsuarioDao();
    boolean loginValido = dao.existe(this.usuario);
    if (loginValido) {
        return "produto";
    } else {
        this.usuario = new Usuario();
        return "login";
    }
}
```

- Para sabermos o nome do usuário logado no momento, adicione dentro da `div` de cabeçalho do `produto.xhtml`, logo em seguida à imagem do logotipo da empresa, o seguinte conteúdo:

```
<div id="usuarioLogado">
    Logado como: #{loginBean.usuario.login}
</div>
```

- Tente fazer o login novamente de forma errada e repare que agora o usuário só é redirecionado para o `produto.xhtml` quando o fizer o login correto.
- Faça o login com sucesso e repare que agora o nome do usuário aparece na parte superior da página. Depois, já com o login realizado com sucesso, acesse diretamente o endereço da página

`produto.xhtml` (`http://localhost:8080/fj26-notas-fiscais/produto.xhtml`). O que aconteceu com o nome de login do usuário que deveria estar na parte superior da página?

- O problema é que os dados do usuário que estão no *Managed Bean* `LoginBean` são perdidos a partir das próximas requisições.

No entanto, nós queremos que esse dado seja mantido enquanto o usuário estiver logado no sistema (futuramente desenvolveremos o logout). Resolvemos esse problema de forma fácil colocando o nosso *Managed Bean* no escopo de sessão. Para isso, basta adicionar a anotação `@SessionScoped` na classe `LoginBean` :

```
@ManagedBean  
 @SessionScoped  
 public class LoginBean {  
     // ...  
 }
```

- Acesse novamente a página `login.xhtml`, entre novamente no sistema e adicione ou altere alguns produtos. Repare que, agora, está funcionando perfeitamente.

7.7 EXERCÍCIO OPCIONAL: APLICANDO LAYOUT NA TELA DE LOGIN

- Agora vamos importar os CSS e adicionar o cabeçalho e o rodapé para melhorarmos o layout da página de login. Para isso faça o seguinte:

- Adicione o CSS de layout no `head` da nossa página de login:

```
<h:head>  
    <title>Sistema de Notas Fiscais - UberDist</title>  
    <h:outputStylesheet library="css" name="style.css" />  
</h:head>
```

- O próximo passo é adicionarmos o cabeçalho. Coloque imediatamente abaixo do `h:body`:

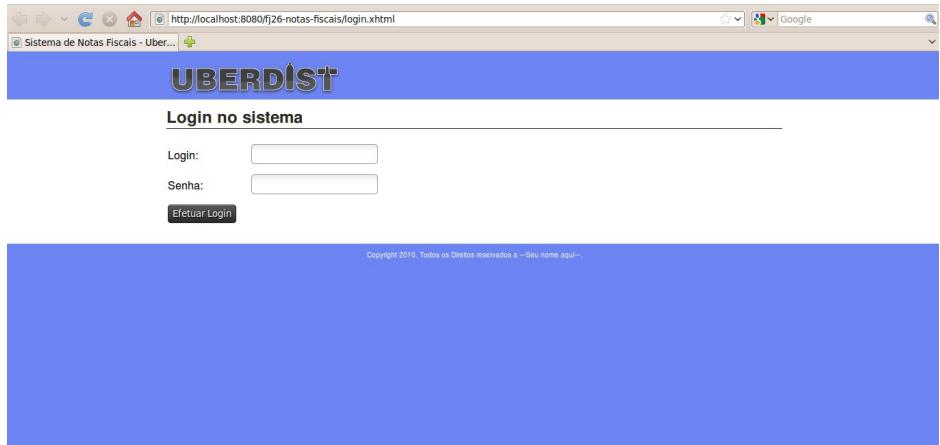
```
<div id="cabecalho">  
    <h:graphicImage library="imagens" name="logo-uber.png"  
        id="logoCompany" />  
</div>
```

- E por fim, imediatamente antes de fechar o `h:body`, adicione o rodapé:

```
<div id="rodape">  
    Copyright 2013. Todos os Direitos reservados a --Seu nome aqui--.  
</div>
```

- Será que teremos que fazer essa cópia do cabeçalho e rodapé em todas as páginas do sistema?

- Acesse novamente a página pelo endereço: `http://localhost:8080/fj26-notas-fiscais/login.xhtml`, o resultado deve ser similar à imagem abaixo:



Agora é a melhor hora de respirar mais tecnologia!



Se você está gostando dessa apostila, certamente vai aproveitar os **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum. Programação, Mobile, Design, Infra, Front-End e Business! Ex-aluno da Caelum tem 15% de desconto, siga o link!

[Conheça a Alura Cursos Online.](#)

7.8 REDIRECIONAMENTO DURANTE A NAVEGAÇÃO

Um problema bastante conhecido ao criarmos uma aplicação web é que logo após submetermos um formulário via método POST, que o JSF sempre faz quando utilizamos `commandButton` ou `commandLink`, essa submissão dos dados pode ser refeita ao apertarmos F5 no nosso navegador. Com isso, dependendo da situação, podemos gravar duas vezes a mesma informação no banco de dados e acabarmos tendo informações repetidas.

Isso acontece, pois o navegador sempre poderá submeter novamente a última requisição, mesmo quando essa última requisição utilizou o método POST. Dependendo do navegador utilizado, uma pergunta ao usuário é feita para saber se a requisição deve realmente ser submetida novamente ou não.

Uma forma de resolver esse problema é forçar o navegador, logo após a execução da lógica, a gerar uma nova requisição, somente para renderizar a página. Com isso, teremos duas requisições (a primeira com um POST para a ação e a segunda apenas um GET para a página a ser renderizada). Como a última requisição serve apenas para a renderização da página, ao apertarmos F5 no navegador, o usuário não

submeterá novamente os dados, apenas buscará a página novamente.

Mas, como fazermos isso de forma transparente para o usuário, sem que ele saiba que duas requisições estão sendo feitas?

Uma opção é fazer com que a aplicação, ao devolver a resposta, devolva para o navegador uma indicação de que a página deve ser buscada novamente. Isso pode ser feito através dos cabeçalhos HTTP de resposta, devolvendo um código 302 (*Moved Temporarily*) e o endereço que deve ser buscado em seguida. Dessa forma o navegador faz todo o trabalho, sem que o usuário da aplicação saiba.

Mas trabalhar com o protocolo HTTP não é algo tão amigável e justamente para isso temos os Frameworks MVC, no nosso caso o JSF. Podemos fazer com que o JSF nos ajude com tudo isso de forma muito simples. Basta indicarmos ao JSF que queremos um redirecionamento para uma página após a execução da ação. Para isso, basta indicarmos no retorno o parâmetro `faces-redirect=true`.

Nesse caso, poderíamos fazer com que as requisições para gravar novos produtos fossem feitas dessa maneira com o seguinte código:

```
public String grava() {  
    //código para a gravação do produto  
  
    return "produto?faces-redirect=true";  
}
```

O que o JSF faz internamente é executar o método `sendRedirect` de `HttpServletResponse` para que todo esse redirecionamento aconteça.

Nas versões anteriores do JSF, isso também era possível através do uso da Tag `redirect` com o valor `true` dentro de um `navigation-case`.

Vale lembrar também que o mesmo não ocorre com requisições AJAX, pois ao pressionarmos F5 nos navegadores as mesmas não são submetidas novamente.

7.9 EXERCÍCIO: EVITANDO SUBMISSÕES DUPLICADAS

1. Faça com que ao efetuar o login, o usuário seja redirecionado para a página de produtos sem que seja possível submeter novamente o login após o mesmo ser feito.

INJEÇÃO DE DEPENDÊNCIAS COM CDI

"A grandeza não consiste em receber honras, mas em merecê-las." -- Aristóteles

Ao término desse capítulo, você será capaz de:

- Entender os problemas do alto acoplamento;
- Aplicar os conceitos de Injeção de Dependências e Inversão de Controle;
- Utilizar a API de CDI para fazer injeção de dependências.

8.1 PROBLEMAS DO ALTO ACOPLAMENTO

Um dos principais assuntos discutidos quando começamos a trabalhar com Orientação a Objetos é a questão de manter o baixo acoplamento e a alta coesão.

Crescemos no mundo O.O. ouvindo isso de todos os lados, mas de fato, qual o real problema que o alto acoplamento nos traz? Primeiro, precisamos entender o que é um código acoplado.

Um código com alto acoplamento é quando uma classe conhece muitos detalhes de suas dependências, de modo que qualquer alteração em uma das dependências faz com que a classe que possui a dependência tenha que ser alterada.

Em nosso código, temos um exemplo claro do quão o alto acoplamento pode ser maléfico. A classe `UsuarioDao` requisita dentro do método `existe` um objeto do tipo `EntityManager` através da classe `JPAUtil`, onde centralizamos a criação dos nossos `EntityManager`s.

Suponha que a classe `JPAUtil` precise agora instanciar `EntityManager`s de algum outro banco de dados e não somente do único banco presente no arquivo `persistence.xml`. Precisaríamos, neste caso, que a classe `JPAUtil` fosse alterada para ter dois tipos de `EntityManagerFactory`, uma para cada unidade de persistência que trabalhamos.

A classe `JPAUtil` precisará saber a partir de qual `EntityManagerFactory` ela criará as instâncias de `EntityManager`. Uma possível estratégia, seria passar na chamada do método `getEntityManager` uma `String` ou algo similar, que diga a classe `JPAUtil` qual `EntityManagerFactory` utilizar para criar a `EntityManager`. Teríamos um código parecido com este:

```
public class JPAUtil {  
    private static EntityManagerFactory emf =
```

```

        Persistence.createEntityManagerFactory("notas");

    private static EntityManagerFactory emf2 =
        Persistence.createEntityManagerFactory("notas2");

    public EntityManager getEntityManager(String factory) {
        if ("notas".equals(factory)) {
            return emf.createEntityManager();
        } else if("notas2".equals(factory)) {
            return emf2.createEntityManager();
        }
    }
}

```

Automaticamente teremos um erro de compilação dentro da classe `UsuarioDao` e não somente nela, mas em todas as classes que fazem referência a classe `JPAUtil`, e que na atual abordagem que estamos usando, podem vir a tornar-se muitas. Um claro indício que nosso código está acoplado demais.

Uma prática muito comum e importante atualmente é, como vemos no curso Laboratório Java com Testes, JSF e Design Patterns (FJ-22), a criação de testes de unidade com ferramentas como JUnit para testar os métodos na nossa aplicação. E um problema muito comum é quando temos testes de classes que dependem de banco de dados, em nosso caso de um `EntityManager`.

Imagine que quiséssemos testar a nossa classe `UsuarioDao` para saber se nossas *queries* estão corretas ou não. Na maneira que estamos tratando nosso *design* atualmente, não teríamos outra alternativa a não ser testar com o banco de dados real, o que pode ser ruim ao longo do tempo, pois sabemos que eventualmente nosso banco pode estar fora do ar ou até mesmo não estar consistente/de acordo com o que esperamos para testar nossas pesquisas.

Isso acontece porque a "busca" por um `EntityManager` na classe `UsuarioDao` está fixa, está acoplada demais, passando uma `String` como parâmetro.

Editora Casa do Código com livros de uma forma diferente



Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.

Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.](#)

8.2 ESTRATÉGIAS PARA DIMINUIR O ACOPLAMENTO

Vimos que a maneira que estamos trabalhando atualmente em nosso *design* está mais atrapalhando do que ajudando. Isso graças ao alto grau de acoplamento que obtivemos ao longo do tempo. Mas como resolver isso? Como desacoplar o código e não afetá-lo com qualquer mudança e poder testar usando testes de unidade?

A fábrica de `EntityManager` que criamos e chamamos de `JPAUtil` pode ser considerada uma forma de diminuir o acoplamento em nossas classes. Ela nos poupa de termos que criar o `EntityManagerFactory` todas vez que precisarmos de um. Porém, como vimos anteriormente, ela nos ajuda mas ainda assim o acoplamento é grande.

O grande problema é que o código da classe `UsuarioDao` busca a dependência que precisa para realizar suas tarefas. E se ao invés de buscarmos esta dependência, recebêssemos ela? A classe `UsuarioDao` passaria agora a receber qualquer `EntityManager` não importando de onde ele veio. Em nossos testes, poderíamos criar a `EntityManager` com base em um banco de dados em memória, por exemplo, o HyperSql Database (HSQLDB), ou ainda nem usar um banco de dados, simulando o objeto.

8.3 INJEÇÃO DE DEPENDÊNCIAS (DI)

Quando definimos que a classe `UsuarioDao` não vai mais atrás de suas dependências e sim recebê-las de alguém, dizemos que estamos invertendo o controle. Inversão de Controle é apenas uma questão de postura com relação ao desenvolvimento. Ao invés de fazer os nossos objetos irem atrás daquilo que precisam para cumprirem seus papéis, fazemos com que recebam tudo já pronto para uso.

Isto porque a responsabilidade de abrir e fechar um `EntityManager` que era da classe `UsuarioDao` é agora de responsabilidade de quem o invoca. Umas das formas de obtermos inversão de controle é através de uma técnica conhecida como **injeção de dependências**, que consiste na premissa de que um objeto recebe suas dependências ao invés de buscá-las manualmente.

Existem diversas formas de fazermos injeção de dependências. Uma delas, e a mais elegante, é injetarmos a dependência via construtor. Nos referimos a um objeto que recebe suas dependências pelo construtor, como um objeto **Bom Cidadão**. Porque todas as suas dependências estão ajustadas e não precisamos nos preocupar com a consistência das mesmas.

A outra forma de injetar uma dependência, seria via *setter*. Porém, desta maneira não conseguimos garantir a consistência do objeto, porque suas dependências podem não estar presentes no momento em que o objeto necessita delas. Teríamos que garantir de algum jeito, que os *setters* que injetam as dependências foram chamados, e não temos como fazer isto.

Quando trabalhamos com Injeção de Dependências sempre temos um objeto dependente (`UsuarioDao`), uma dependência ou várias (`EntityManager`) e um provedor que é o responsável por fazer essa ligação entre dependente e dependência. Esse *provider* pode ser um *Service Locator*, uma *Factory* (`JPAUtil`) ou um framework, que é o mais comum hoje em dia.

Os mais conhecidos frameworks de injeção de dependências utilizam diferentes técnicas para fazê-la. *Pico Container* injeta dependência via construtor, assim como o *Google Guice*. O *Spring*, bem conhecido e muito utilizado, faz a injeção tanto via *setter*, quanto via construtor. O EJB a partir da versão 3.0 oferece um pouco de injeção de dependências por si só.

8.4 IMPLEMENTANDO DI COM CONTEXTS E DEPENDENCY INJECTION API - CDI

O "Context and Dependency Injection for the Java EE platform" (CDI) é uma especificação que diz como devemos trabalhar com injeção de dependências em qualquer aplicação Java EE. Especificada na JSR 299, passou a ser incluída no Java EE 6 no fim de 2009. Atualmente está na versão 1.1 (JSR 346), já incluída no Java EE 7 recentemente lançado. Ela foi inspirada em muitas ideias do *JBoss Seam 2*, ambos possuindo o mesmo criador, Gavin King.

A especificação já foi chamada de *WebBeans* e depois renomeada para CDI. Outro ponto que ainda causa um pouco de confusão foi a criação da JSR 330 "Dependency Injection for Java", que nada mais é do que a extração das anotações da CDI para uma especificação menor, tornando assim mais fácil que frameworks como *Spring* e *Guice* utilizem anotações padrão, mesmo não sendo implementações completas da CDI.

O CDI sofreu várias influências do *JBoss Seam 2* e *Google Guice*, a vantagem é que ele faz injeção de dependências de modo *type safe*, diferente do *Seam 2* que fazia as injeções baseadas em Strings.

Como toda especificação, para utilizá-la dentro da nossa aplicação precisamos de uma implementação. Utilizaremos a implementação da JBoss conhecida como **Weld**. O Weld é a implementação de referência da especificação e é usada internamente tanto no JBoss e quanto no Glassfish.

LIMITAÇÕES EM UM SERVLET CONTAINER

Na especificação JSR-299, é definido que não existe suporte para *Servlets Containers*, porém o JBoss Weld abriu uma exceção e oferece suporte no Tomcat e no Jetty.

Na documentação do Weld, eles dizem que uma aplicação rodando em um *Servlet Container*, não é suportado a criação de *Session Beans*, injeção de `@EJB` ou `@PersistenceContext`.

Quando começamos a trabalhar com o CDI é importante entender o que é um **bean package**. Um bean package nada mais é do que um `.jar` ou uma aplicação que possua um arquivo chamado **beans.xml** dentro de uma pasta `META-INF` (ou `WEB-INF` num projeto web). O módulo (JAR ou aplicação) que contiver o arquivo `beans.xml` será considerado pelo CDI e será um candidato a ser

injetado como dependência.

Vamos criar um arquivo em branco, chamado `beans.xml` dentro da pasta `WEB-INF`, junto com os outros arquivos de configuração. Fazendo isso, tornamos todas as classes do projeto `fj26-notas-fiscais` como `beans` elegíveis para serem injetados.

Estamos prontos para resolver as pendências do *design* da nossa aplicação. Vamos primeiro fazer com que a classe `UsuarioDao` receba uma `EntityManager` ao invés de buscá-la dentro da `JPAUtil`.

Se estivéssemos usando um `EntityManager` administrado pelo servidor de aplicação, poderíamos simplesmente colocar um atributo de instância do tipo `EntityManager`, na classe `UsuarioDao`, e anotá-lo com `@PersistenceContext`.

```
public class UsuarioDao {  
  
    @PersistenceContext  
    private EntityManager manager;  
  
    public boolean existe(Usuario usuario) {  
        // implementação usando o EntityManager  
    }  
}
```

Porém, o código acima não funciona pois não configuramos a JPA dessa maneira. No nosso caso a aplicação que é responsável por inicializar a `EntityManager` no `JPAUtil`. Neste caso, utilizamos uma estratégia um pouco diferente que é fornecida pelo CDI. Utilizamos um conceito de **método produtor**. Quando criamos um método produtor, o CDI passa a considerá-lo no momento em que precisa injetar e/ou gerenciar as dependências.

Vamos configurar o método `getEntityManager`, da classe `JPAUtil`, para que ele torne-se um método produtor. Para isto, basta anotar o método com a anotação `@Produces`.

```
public class JPAUtil {  
    private static EntityManagerFactory emf = Persistence  
        .createEntityManagerFactory("notas");  
  
    @Produces  
    public EntityManager getEntityManager() {  
        return emf.createEntityManager();  
    }  
}
```

Pronto! Quando existir uma dependência do tipo `EntityManager` o CDI injetará a dependência utilizando o método produtor de `EntityManager`s que configuramos na classe `JPAUtil`.

Vamos **inverter o controle e injetar a dependência** dentro da classe `UsuarioDao`. Uma das maneiras de injetarmos dependências, utilizando o CDI, é criando um atributo de instância e anotá-lo com `@Inject`:

```
public class UsuarioDao {  
  
    @Inject
```

```

private EntityManager manager;

public boolean existe(Usuario usuario) {
    // implementação usando o EntityManager
}
}

```

A classe `UsuarioDAO` atualmente é utilizada dentro do *ManagedBean* `LoginBean`. Dentro do método `efetuaLogin` é criada uma instância de `UsuarioDAO`, onde é efetuada uma chamada ao método `existe` para verificar se o usuário passado, existe no banco de dados.

Porém, quando "manualmente" criamos uma instância da classe `UsuarioDAO`, nós mesmo assumimos o controle dela. Obviamente o CDI não tem controle sobre estas ações e não tem como efetuar a injeção de dependências.

Portanto, para que a instância `UsuarioDAO` receba a injeção de uma `EntityManager` não a instanciaremos. Faremos com que a dependência `UsuarioDAO` da classe `LoginBean` também seja injetada pelo CDI, de modo que o CDI passará a controlar a criação da mesma.

Mas para que a dependência possa ser injetada dentro do `LoginBean`, precisamos torná-la uma classe gerenciável pelo CDI. Sendo assim, vamos substituir a anotação `@ManagedBean` do JSF e usar a anotação `@Named` do CDI em seu lugar para que nosso *ManagedBean* seja gerenciável pelo CDI e acessível via *Expression Language*. Além disso, como nosso *bean* será gerenciado pelo CDI, devemos usar seus escopos próprios. Para isso, trocamos a anotação `@SessionScoped` do JSF (`javax.faces.bean.SessionScoped`) por outra, com mesmo nome, mas no pacote do CDI (`javax.enterprise.context.SessionScoped`).

```

@Named
@SessionScoped
public class LoginBean {

    private Usuario usuario = new Usuario();

    @Inject
    private UsuarioDAO dao;

    // outros métodos
}

```

ANOTAÇÕES DE ESCOPO DO CDI

Além da anotação `@SessionScoped` temos outras que auxiliam no gerenciamento de classes injetáveis pelo CDI: `@RequestScoped`, `@ApplicationScoped` e `@ConversationScoped`.

O CDI, porém, não suportava o `@ViewScoped` do JSF que veremos mais para frente. Por isso entrou na versão 2.2 do JSF uma extensão do CDI para fornecer um `@ViewScope`. Nas versões anteriores do JSF era preciso usar alguma biblioteca externa como o *JBoss Seam 3* ou *Apache CODI*.

ANOTAÇÃO @NAMED

A anotação `@Named` serve para expor um *bean* para o CDI a partir de um determinado nome. No caso da classe `LoginBean` fizemos este procedimento para que possamos utilizar a classe dentro da EL do JSF. Faríamos isso usando a seguinte sintaxe:

```
html#{loginBean}
```

Repare que, neste caso, o CDI assumiu que o nome do *bean* será de acordo com o padrão *Java Bean*. Poderíamos alterar este comportamento, colocando um argumento com o nome que desejamos que o *bean* seja exposto: `@Named("login")`.

Repare que o método `efetuaLogin` não precisará mais buscar a dependência, instanciando um objeto `UsuarioDAO`. O melhor é que a instância de `UsuarioDAO` que será injetada, também receberá automaticamente um `EntityManager`.

Já conhece os cursos online Alura?



A Alura oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Programação, Front-end, Mobile, Design & UX, Infra e Business, com um plano que dá acesso a todos os cursos. Ex aluno da Caelum tem 15% de desconto neste link!

[Conheça os cursos online Alura.](#)

8.5 LIBERAÇÃO DE DEPENDÊNCIAS COM @DISPOSES

Já utilizamos o CDI para fazer a abertura da `EntityManager` mas não nos preocupamos naquele momento com o seu fechamento. Quando anotamos o método `getEntityManager` da classe `JPAUtil` com `@Produces` nós dissemos quem será o responsável pela produção do objeto.

O fechamento, até o momento, está no próprio método `efetuaLogin` no *bean*. Mas imagine que tivéssemos outro método no mesmo *bean* que também usa o banco de dados. Recebemos o `EntityManager` no construtor, mas onde fechá-lo? Se um método fechar, o outro método não funcionará. Ou ainda, se quisermos compartilhar o mesmo `EntityManager` entre vários

ManagedBeans diferentes: qual deles deverá fechá-lo? Em que momento?

Podemos deixar o trabalho de fechar o `EntityManager` para o CDI. Basta ter um método dentro de `JPAUtil` que será responsável por receber uma `EntityManager` e fazer o seu **fechamento**:

```
// recebe uma EntityManager e fecha
public void close(EntityManager manager) {
    manager.close();
}
```

Com isso, quando o CDI precisar criar o objeto ela utilizará o método com `@Produces` e para o fechamento, ele deve utilizar o novo método `close` que acabamos de criar. Para isso, basta indicarmos que a `EntityManager` que estamos recebendo como parâmetro será **descartada** através da anotação `@Disposes`:

```
public void close(@Disposes EntityManager manager) {
    manager.close();
}
```

No entanto, ainda temos um problema. Em todo o ponto da nossa aplicação que precisarmos de uma `EntityManager` injetada através do `@Inject` o método `getEntityManager` será invocado, criando novas `EntityManagers`. Durante a requisição, só queremos **uma** única `EntityManager`. Para conseguirmos isso, basta dizer que o objeto produzido será utilizado no escopo de `request` anotando o método com `@RequestScoped`:

```
@Produces @RequestScoped
public EntityManager getEntityManager() {
    return emf.createEntityManager();
}
```

Com isso, resolvemos o problema de deixar conexões abertas indiscriminadamente. Um novo `EntityManager` será criado a cada requisição e usado por todos que pedirem para injetá-lo. Ao fim do escopo, no término da requisição, nosso método de liberação será invocado, fechando o `EntityManager`.

8.6 PARA SABER MAIS: CONFIGURAÇÃO DO WELD NO SERVLET CONTAINER

Como estamos usando o Glassfish que já vem pré-configurado com Weld basta usar as anotações. Dentro de um *Servlet Container*, como o Apache Tomcat, existem mais duas restrições para tudo isto funcionar: i) É preciso configurar um **listener** em nosso `web.xml` para que o Tomcat passe a funcionar com o CDI/Weld; e ii) Colocar um JAR encontrado no pacote que baixamos do Weld, chamado `weld-servlet`, na pasta `WEB-INF/lib` da nossa aplicação.

Sendo assim, dentro do `web.xml` basta colocar o seguinte código:

```
<listener>
    <listener-class>
        org.jboss.weld.environment.servlet.Listener
    </listener-class>

```

```
</listener-class>
</listener>
```

E depois copiar o `weld-servlet.jar` para a pasta `WEB-INF/lib`. A distribuição do Weld encontra-se ainda na página do Seam Framework:

<http://www.seamframework.org/Weld/Downloads>

8.7 EXERCÍCIOS: UTILIZANDO CDI PARA INJETAR DEPENDÊNCIAS

1. **Copie** o arquivo chamado `beans.xml`, da pasta `caelum/26/configs`, para dentro da pasta `WebContent/WEB-INF`. Isso inicializa o CDI no nosso projeto web.

O arquivo tem nada mais do uma declaração da tag `bean`:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
                           http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
       bean-discovery-mode="all">

</beans>
```

2. **Altere** a classe `JPAUtil` para tornar o método `getEntityManager` um método produtor do CDI. **Adicione** as anotações `@Produces` e `@RequestScoped` na assinatura do método `getEntityManager`.

Cuidado: A anotação `@Produces` é do pacote `javax.enterprise.inject.Produces` e `@RequestScoped` do pacote `javax.enterprise.context.RequestScoped`.

3. Ainda na classe `JPAUtil`, **crie** o método para tratar do fechamento dos `EntityManagers` ao fim do *request*, usando a anotação `@Disposes`:

```
public void close(@Disposes EntityManager manager) {
    manager.close();
}
```

4. Altere também sua classe `UsuarioDao` para receber o `EntityManager` injetado a partir de nossa classe produtora:

- **Crie** um atributo do tipo `EntityManager` e anote-o com `@Inject` (do pacote `javax.inject`).

```
@Inject
private EntityManager manager;
```

- Por fim, **altere** o método do `UsuarioDao` que usa o `EntityManager`, removendo a linha que invoca a classe `JPAUtil` manualmente, de forma que o método passe a usar o atributo novo que injetamos. Caso exista, **retire** também o fechamento do `EntityManager` já que agora o CDI cuidará disso.

5. Agora, **altere** sua classe `LoginBean`, fazendo-a receber o `UsuarioDao` via injeção. Para isso, **crie** um atributo do tipo `UsuarioDao` e anote-o com `@Inject`.

```
@Inject
private UsuarioDao dao;
```

6. Ainda na classe `LoginBean`, troque a anotação `@ManagedBean` por `@Named` e use a anotação `@SessionScoped` do pacote `javax.enterprise.context.SessionScoped`:

```
@Named
@SessionScoped
public class LoginBean {
    // código omitido
}
```

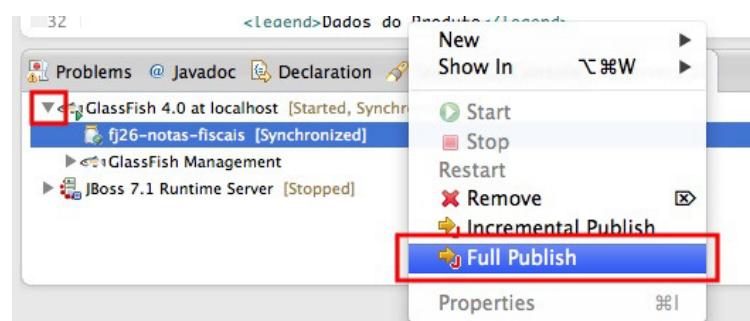
Cuidado para importar as anotações do CDI e não do JSF.

O Weld exige ainda que nossas classes implementem `Serializable` para estar no escopo de sessão. Dessa forma, ele saberá como persistir o objeto internamente. Além do `LoginBean`, faça com que as classes `Usuario` e `UsuarioDAO` também implementem `Serializable`.

7. **Altere** também a lógica do método `efetuaLogin` para usar o `dao` injetado e "setar" o usuário logado na sessão em caso de sucesso:

```
public String efetuaLogin() {
    boolean loginValido = dao.existe(this.usuario);
    if (loginValido) {
        return "produto?faces-redirect=true";
    } else {
        this.usuario = new Usuario();
        return "login";
    }
}
```

8. Para garantir que todas as alterações feitas no projeto serão enviadas ao servidor Glassfish 4.0 (*deploy*), na aba *Servers*, clique na seta que está em frente ao nome do servidor para exibir as aplicações que estão implantadas no servidor. Clique com o botão direito do mouse sobre o nome da aplicação que estamos trabalhando (`fj26-notas-fiscais`), escolha **Full Publish** e aguarde até a aplicação estar *Synchronized*.



9. Acesse sua página de login e faça os testes com usuários válidos e inválidos.

Tudo deve continuar funcionando como antes, sem alterações para o usuário. O que está melhor?

Pense bem: Faz sentido injetar o `UsuarioDao` no `LoginBean` que possui o escopo `SessionScoped` ?

O DILEMA DO CONSTRUTOR PADRÃO VS. INJEÇÃO EM ATRIBUTOS OU SETTERS

Falamos que injetar via construtor é uma prática melhor que via atributos privados ou *setters*, para manter a consistência do objeto e ter um melhor *design*.

Mas há uma restrição importante com o Weld que é a exigência do construtor padrão para uso na geração de proxies, caso o escopo usado seja diferente do `@Dependant` padrão. Podemos colocar o construtor como *private-package* mas sua presença pode incomodar algumas pessoas.

Uma forma de aliviar esse incômodo seria marcar esse construtor como `@Deprecated` para indicar aos programadores que não se deve chamá-lo no código. Documentá-lo adequadamente também seria uma boa ideia.

O mais comum em projetos JSF com CDI é usar anotações nos atributos, como vimos nesse capítulo.

Saber inglês é muito importante em TI

galandra O **Galandra** auxilia a prática de inglês através de *flash cards* e *spaced repetition learning*. Conheça e aproveite os preços especiais.

[Pratique seu inglês no Galandra.](#)

8.8 MELHORANDO A AUTENTICAÇÃO COM CDI

Ainda temos um problema em nosso código. A classe `LoginBean` está no escopo de sessão porque precisa guardar o usuário que se logou na aplicação. Isto implica que ela persistirá junto ao CDI, pelo tempo da sessão do usuário.

O problema é que, fazendo isto, a instância da classe `UsuarioDao`, que foi injetada em `LoginBean`,

também permanecerá na sessão, pois a classe `UsuarioDao` está com o escopo default do CDI(`@Dependent`).

Podemos resolver isto colocando a classe `LoginBean` no escopo de requisição, mas com isso teríamos o problema de perder as informações do usuário logado. Visando resolver mais este problema, vamos criar uma nova classe, em escopo de sessão, que será gerenciada pelo CDI, e que guardará as informações do usuário logado de maneira separada da classe `LoginBean`, cuja responsabilidade agora será apenas efetuar o login/logout do usuário na aplicação.

Para isso usaremos a anotação `@Named` junto a definição do escopo `@SessionScoped`:

```
@Named  
@SessionScoped  
public class UsuarioLogadoBean implements Serializable {  
  
    private Usuario usuario;  
  
    public void logar(Usuario usuario) {  
        this.usuario = usuario;  
    }  
  
    public void deslogar() {  
        this.usuario = null;  
    }  
}
```

Devemos agora alterar o código da classe `LoginBean` para que guarde o usuário logado dentro da classe `UsuarioLogadoBean`.

Agora a classe `LoginBean` receberá a dependência `UsuarioLogadoBean`, o método `efetuaLogin` será alterado para guardar o usuário logado dentro da instância recebida da classe `UsuarioLogadoBean` e por fim retiraremos a anotação `@SessionScoped` para colocar o *bean* em escopo de requisição com `@RequestScoped`:

```
@Named  
@RequestScoped  
public class LoginBean {  
  
    private Usuario usuario = new Usuario();  
  
    @Inject  
    private UsuarioLogadoBean usuarioLogado;  
  
    @Inject  
    private UsuarioDao dao;  
  
    // outros métodos  
  
    public String efetuaLogin() {  
  
        boolean loginValido = dao.existe(this.usuario);  
  
        if (loginValido) {  
            usuarioLogado.logar(usuario);  
            return "produto?faces-redirect=true";  
        }  
    }  
}
```

```

        } else {
            usuarioLogado.deslogar();
            return "login";
        }
    }
}

```

8.9 EXERCÍCIOS: MELHOR GERENCIAMENTO DOS ESCOPOS DAS DEPENDÊNCIAS

- Precisamos isolar o usuário logado na sessão em seu próprio *ManagedBean*. Ele deverá ser independente de *LoginBean*, que então tratará só do acesso à banco de dados e da tela de login.

Podemos fazer isso criando um novo *ManagedBean* bem simples chamado *UsuarioLogadoBean*, gerenciado pelo CDI e que apenas guarde em um atributo o usuário logado na sessão. Não podemos deixar de marcá-lo como *@SessionScoped*:

```

@Named
@SessionScoped
public class UsuarioLogadoBean implements Serializable {

    private Usuario usuario;

    public void logar(Usuario usuario) {
        this.usuario = usuario;
    }

    public void deslogar() {
        this.usuario = null;
    }

    public Usuario getUsuario() {
        return usuario;
    }
}

```

Mais uma vez **cuidado** para importar as anotações do CDI (`javax.enterprise.context.SessionScoped`) e não do JSF.

Não esqueça que o Weld exige que nossa classe implemente *Serializable* para estar no escopo de sessão.

- Altere sua classe *LoginBean*. Tire a anotação *@SessionScoped*, coloque *@RequestScoped* no lugar e faça-o receber um *UsuarioLogadoBean* via injeção:

```

import javax.enterprise.context.RequestScoped
// outros imports

@Named
@RequestScoped
public class LoginBean {

    @Inject
    private UsuarioLogadoBean usuarioLogado;
}

```

```
// outros atributos e métodos  
}
```

Altere também a lógica do método `efetuaLogin` para invocar o método que atribui o usuário logado à sessão em caso de sucesso:

```
public String efetuaLogin() {  
    boolean loginValido = dao.existe(this.usuario);  
    if (loginValido) {  
        usuarioLogado.logar(usuario);  
        return "produto?faces-redirect=true";  
    } else {  
        usuarioLogado.deslogar();  
        this.usuario = new Usuario();  
        return "login";  
    }  
}
```

3. Realize um **Full Publish** da aplicação no servidor, acesse sua página de login e faça alguns testes com usuários válidos e inválidos.
4. Na página `produto.xhtml`, na `div` com a id `usuarioLogado` altere a expressão para acessar o novo *bean* `usuarioLogadoBean`.

```
<div id="usuarioLogado">  
    Logado como: #{usuarioLogadoBean.usuario.login}  
</div>
```

8.10 PARA SABER MAIS: INJEÇÃO DE DEPENDÊNCIAS E TESTES

É sempre bom reforçar que apesar da possibilidade de usar o CDI para injetar nossas dependências diretamente nos atributos, é considerada uma boa prática fazer a injeção por construtor, assim garantimos que quando instanciarmos um objeto da classe, todas as suas dependências serão atendidas, mesmo que estejamos instanciando o objeto na mão. (Good Citizen).

A vantagem dessa abordagem é clara quando fazemos o uso de testes de unidade: podemos facilmente passar todas dependências ao instanciar a classe testada, sem criar *setters* adicionais ou lançar mão de uso da API de *Reflection* para acessarmos diretamente o atributo (o que indica que nosso teste conhece detalhes da implementação da classe testada que estão encapsulados).

Seus livros de tecnologia parecem do século passado?



Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**. Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

[Casa do Código, Livros de Tecnologia.](#)

8.11 INTERCEPTANDO AS FASES COM PHASELISTENERS

Relembrando as fases do JSF

Como vimos no capítulo 6, o JSF possuí um ciclo de vida com seis fases. Para usar JSF extensivamente, é necessário conhecê-las bem. Vamos brevemente relembrá-las.

A primeira fase é conhecida como *Restore View*, onde o JSF guarda o estado da árvore de componentes na sessão HTTP ou em um campo hidden no cliente.

A segunda fase é conhecida como *Apply Request Values*, onde o JSF recupera os valores da requisição e atribuiu os mesmos para a árvore de componentes.

A terceira fase é conhecida como *Process Validations*, onde o JSF converte os valores que foram recuperados como String na fase anterior, os converte para o tipo apropriado do nosso modelo e após isso, aplica as validações necessárias.

A quarta fase é conhecida como *Update Model Values*, onde o JSF atribui os valores que foram convertidos na fase anterior para nossos objetos de negócio que foram ligados (binding) com a tela.

A quinta fase é conhecida como *Invoke Application*. Agora que os dados foram devidamente atribuídos dentro do *ManagedBean*, o mesmo encontra-se em um estado consistente e o JSF faz a chamada para os métodos de negócio.

A sexta e última fase é conhecida como *Render Response*, onde o JSF devolve uma resposta para o cliente que fez a requisição.

Como verificar se o usuário está logado ou não no sistema?

Nos capítulos anteriores fizemos a funcionalidade para autenticar usuários em nosso sistema. Porém,

o usuário consegue acessar qualquer uma das telas do sistema da UberDist, porque em nenhum momento verificamos se o usuário está mesmo logado antes que ele acesse a tela desejada.

Precisamos portanto validar se o usuário está logado no momento em que tenta acessar alguma tela. Poderíamos utilizar um *Filter* da API de Servlets para resolver este problema, mas como atrelamos a funcionalidade de autenticação com o JSF, essa informação está guardada dentro de `UsuarioLogadoBean` que fica dentro do *Session Context*, que por sua vez está dentro da *Session*.

```
FacesContext context = ...  
UsuarioLogadoBean usuarioLogado = context.pegaUsuarioLogado();
```

Como usamos o CDI basta receber o `UsuarioLogadoBean`, ou seja injetá-lo:

```
@Inject  
private UsuarioLogadoBean usuarioLogado;
```

8.12 AUTORIZAÇÃO COM PHASELISTENERS

Como atrelamos a funcionalidade de autenticação com o JSF, o melhor é utilizar algo mais próximo do que estamos trabalhando. No JSF existe uma funcionalidade chamada *PhaseListener* que é capaz de fazer o mesmo papel que um filtro faz, porém com uma diferença: utilizando *PhaseListener* podemos interceptar cada uma das diferentes fases do JSF e executar algo antes, ou depois de cada uma delas. Para isto, basta criar uma classe que implemente a interface `javax.faces.event.PhaseListener`. Esta interface, exige que implementemos 3 métodos: `afterPhase`, `beforePhase` e `getPhaseId`.

```
public class Autorizador implements PhaseListener {  
    private static final long serialVersionUID = 1;  
  
    public void beforePhase(PhaseEvent event) {  
    }  
  
    public void afterPhase(PhaseEvent event) {  
    }  
  
    public PhaseId getPhaseId() {  
        return null;  
    }  
}
```

Os métodos `beforePhase` e `afterPhase` são executados antes e depois de alguma fase, respectivamente. Mas antes de qual fase? É exatamente isto que definimos no método `getPhaseId`.

Este método espera que seja retornada uma *enumeration* chamada `PhaseId` que possui as fases do JSF: `RESTORE_VIEW`, `APPLY_REQUEST_VALUES`, `INVOKE_APPLICATION`, `PROCESS_VALIDATIONS`, `RENDER_RESPONSE` e `UPDATE_MODEL_VALUES`. Além da `ANY_FASE`, que, caso utilizada, faz com que o *PhaseListener* seja executado em qualquer uma das fases.

Vamos definir primeiro em qual das fases do JSF verificaremos se o usuário está logado. Para isto, no

método `getPhaseId` retornaremos que a verificação deverá executar o *PhaseListener* na parte de recuperar a view (`RESTORE_VIEW`):

```
public PhaseId getPhaseId() {  
    return PhaseId.RESTORE_VIEW;  
}
```

Como estamos validando se o usuário está logado, podemos escolher em que momento queremos fazer essa validação, neste caso, vamos fazê-la depois da fase que foi definida no método `getPhaseId`.

```
public void afterPhase(PhaseEvent event) {  
    FacesContext context = event.getFacesContext();  
  
    if ("/login.xhtml".equals(context.getViewRoot().getViewId())) {  
        return;  
    }  
}
```

No código acima, estamos validando se a requisição veio a partir da tela de login, afinal se o usuário está efetuando login, não precisamos validar se o mesmo está logado ou não.

Caso o usuário esteja tentando o acesso a partir de uma tela diferente da tela de login, precisamos efetuar a validação de usuário logado.

Para verificar se o usuário está logado, precisamos primeiro obter a instância da classe `UsuarioLogadoBean` que contém o possível usuário logado. Novamente o CDI ajuda e podemos facilmente injetar o `UsuarioLogadoBean`:

```
public class Autorizador implements PhaseListener {  
  
    @Inject  
    private UsuarioLogadoBean usuarioLogado;  
  
    //outros métodos aqui  
  
}
```

Após obter a instância que está na memória, precisamos verificar se o atributo `usuario`, dentro do `UsuarioLogadoBean`, está nulo ou não, pois é isto o que define se o usuário está ou não logado. Mantendo sempre as boas práticas de programação orientada a objetos, vamos incluir um método que verifique esta condição dentro do `UsuarioLogadoBean`. O chamaremos de `isLogado`.

```
@Named @SessionScoped  
public class UsuarioLogadoBean implements Serializable {  
  
    private Usuario usuario;  
  
    public void logar(Usuario usuario) {  
        this.usuario = usuario;  
    }  
  
    public void deslogar() {  
        this.usuario = null;  
    }
```

```

    public Usuario getUsuario() {
        return usuario;
    }

    public boolean isLogado() {
        return usuario != null;
    }
}

```

Dentro do *PhaseListener*, efetuamos a verificação e, caso o método `isLogado` retorne `false`, faremos um redirecionamento do usuário para a tela de login, utilizando a navegação implícita que foi explicada no capítulo de navegação.

```

public void afterPhase(PhaseEvent event) {

    FacesContext context = event.getFacesContext();

    if ("/login.xhtml".equals(context.getViewRoot().getViewId())) {
        return;
    }

    // Usando o usuarioLogado que foi injetado
    if (!usuarioLogado.isLogado()) {

        NavigationHandler handler = context.getApplication()
            .getNavigationHandler();
        handler.handleNavigation(context, null, "login?faces-redirect=true");

        // Efetua renderização da tela
        context.renderResponse();
    }
}

```

Lembrando que não precisamos configurar nenhuma regra de navegação dentro do `faces-config.xml` como era feito anteriormente na versão 1.2 do JSF. O JSF 2.2 automaticamente vai procurar por um arquivo `.xhtml` de acordo com a String que foi especificada dentro do método `handleNavigation`.

Com toda a lógica feita, basta dizermos ao JSF que o *PhaseListener* que acabamos de fazer deve ser aplicado às requisições. Este é um dos pontos que tínhamos a esperança de ser alterado na versão mais nova da JSF, mas não foi feito.

Para registrarmos um *PhaseListener*, temos que configurar ele no arquivo `faces-config.xml`:

```

<lifecycle>
    <phase-listener>
        br.com.caelum.notasfiscais.listener.Autorizador
    </phase-listener>
</lifecycle>

```

8.13 EXERCÍCIOS: AUTORIZAÇÃO COM PHASELISTENER

1. Para sabermos se o usuário está, ou não, logado na aplicação, vamos criar um novo método na classe `UsuarioLogadoBean` que faça essa verificação:

```

@Named
@SessionScoped
public class UsuarioLogadoBean implements Serializable {

    // Outros atributos e métodos...

    public boolean isLoggedIn() {
        return usuario != null;
    }
}

```

2. Crie uma classe que implemente a interface `PhaseListener` e faça a verificação se o usuário está logado. Caso ele não esteja, será redirecionado para a tela de login.

Chame a sua classe de `Autorizador` e coloque-a no pacote `br.com.caelum.notasfiscais.listener`. Ela deve implementar a interface `PhaseListener` e seus três métodos (`beforePhase` , `afterPhase` e `getPhaseId`).

Dentro do método `getPhaseId` defina que nosso autorizador deve estar associado à fase **`RESTORE_VIEW`**:

```

public PhaseId getPhaseId() {
    return PhaseId.RESTORE_VIEW;
}

```

A autorização em si deve ser feita **após a fase RESTORE_VIEW**.

No método `afterPhase` vamos precisar do `UsuarioLogadoBean` . Injete-o na classe `Autorizador` :

```

public class Autorizador implements PhaseListener {

    @Inject
    private UsuarioLogadoBean usuarioLogado;

    // demais métodos
}

```

Implemente o método `afterPhase` . Lembre-se de deixar a página `login.xhtml` liberada e de redirecionar para o login caso o usuário não esteja logado:

```

public void afterPhase(PhaseEvent event) {
    FacesContext context = event.getFacesContext();

    if ("/login.xhtml".equals(context.getViewRoot().getViewId())) {
        return;
    }

    // Verificação
    if (!usuarioLogado.isLoggedIn()) {
        NavigationHandler handler = context.getApplication()
            .getNavigationHandler();

        handler.handleNavigation(context, null,
            "login?faces-redirect=true");

    // Efetua renderização da tela
}

```

```
        context.renderResponse();
    }
}
```

3. Configure o `PhaseListener` de autorização no arquivo `faces-config.xml`:

```
``` xml
<lifecycle>
 <phase-listener>
 br.com.caelum.notasfiscais.listener.Autorizador
 </phase-listener>

 <!-- outros phase listeners podem estar aqui -->
</lifecycle>
```

```

1. Reinicie o servidor e acesse a página de cadastro de produtos sem logar. Verifique que estamos sendo redirecionados automaticamente para a tela de login. Uma comprovação de que o nosso `PhaseListener` está funcionando corretamente.

Efetue login com um nome de usuário e senha válidos, criados no capítulo de navegação, e tente acessar novamente a tela de produtos.

2. DESAFIO: No método `afterPhase` tem mais duas linhas que fazem um *lookup* ao invés de injetar a dependência:

```
FacesContext context = event.getFacesContext();
// ...
NavigationHandler handler = context.getApplication()
    .getNavigationHandler();
```

Podemos simplificar isso com CDI. Como?

Agora é a melhor hora de respirar mais tecnologia!



Se você está gostando dessa apostila, certamente vai aproveitar os **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum. Programação, Mobile, Design, Infra, Front-End e Business! Ex-aluno da Caelum tem 15% de desconto, siga o link!

[Conheça a Alura Cursos Online.](#)

8.14 EXERCÍCIOS: LOGOUT E TELA DE CADASTRO DE USUÁRIOS

1. Desenvolva o `logout` da aplicação. Basta fazer uma lógica em `LoginBean` que limpe o usuário da

sessão. Não esqueça de reiniciar o servidor ao final.

2. (opcional) Baseado no cadastro do produto, faça toda a funcionalidade para cadastrar usuários.

Dica: Lembre-se que você pode mexer nos DAOs, na entidade e criar páginas e *ManagedBeans* novos.

8.15 EXERCÍCIOS: MIGRAR TUDO PARA CDI

Com tudo o que já aprendemos, podemos migrar o restante do sistema para CDI.

1. Na classe `ProdutoBean`, troque a anotação `@ManagedBean` por `@Named`. Não esqueça de configurar o `@RequestScoped` (do pacote `javax.enterprise.context`).

Ainda na classe `ProdutoBean`, injete o `ProdutoDao` e apague todas as linhas que instanciem o `ProdutoDao`.

2. Na classe `ProdutoDao`, injete o `EntityManager`. Depois apague todas as linhas que abram e fechem o `EntityManager`.

Editora Casa do Código com livros de uma forma diferente



Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.

Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.](#)

TEMPLATES COM FACELETS

"O caminho do inferno está pavimentado de boas intenções." -- Marx

9.1 REPETIÇÃO DE CÓDIGO E SEUS PROBLEMAS

Um dos problemas mais comuns encontrados em aplicações que crescem muito em quantidade de código com o passar do tempo, é a duplicidade do mesmo. Essa duplicidade faz com que aumente a dificuldade em dar manutenção, pois ao querermos alterar alguma funcionalidade, precisaremos rastrear os locais da aplicação em que o código está espalhado.

Muitas vezes duplicar código pode ser a solução mais simples e rápida para resolver determinados problemas, mas é importante estar ciente do impacto negativo que ela causará futuramente na manutenção do sistema.

A aplicação da UberDist está começando a ter problemas com a duplicidade de código. Tanto na página `login.xhtml`, quanto na `produto.xhtml`, duplicamos o código do cabeçalho e do rodapé do sistema. Da maneira que a aplicação está desenvolvida atualmente, se precisarmos alterar algo no cabeçalho ou no rodapé, precisaremos fazer a alteração em ambas as páginas. Imagine isso acontecendo em uma aplicação contendo 100 ou mais páginas!

Já conhece os cursos online Alura?



A Alura oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Programação, Front-end, Mobile, Design & UX, Infra e Business, com um plano que dá acesso a todos os cursos. Ex aluno da Caelum tem 15% de desconto neste link!

[Conheça os cursos online Alura.](#)

9.2 RESOLVENDO A DUPLICIDADE DAS VIEWS

Uma das formas mais comuns de resolvemos a duplicidade nos JSPs é separando as telas em arquivos (fragmentos), dessa forma, para termos uma tela completa de login, teríamos que inclui em seu cabeçalho e rodapé os respectivos fragmentos de páginas criados em outros arquivos.

No entanto, ao utilizarmos **Facelets** a abordagem torna-se um pouco diferente da maneira seguida nos JSPs.

9.3 TEMPLATING COM O FACELETS

Ao utilizarmos *Facelets*, a forma que temos de remover o problema de duplicidade dos códigos de nossas *Views* é através da criação de **templates**.

Os templates são divididos em duas partes: o *arquivo de template* e o *cliente do template*.

Para entendermos melhor, podemos fazer uma analogia simples com uma folha de papel. Essa folha de papel tem desenhado nela partes que são comuns para todas as páginas, no nosso caso específico o cabeçalho e o rodapé. Essa folha de papel será o equivalente ao *arquivo de template*.

A folha que representa o *arquivo de template* possui alguns buracos dentro dela, nos quais utilizaremos outras folhas contendo somente as partes que deverão ser exibidas dentro do *arquivo de template*. Essas outras folhas são por sua vez os *clientes do template*, que no nosso exemplo são os formulários de login e a tela de cadastro de produtos.

O que precisamos fazer primeiramente para implementarmos essa estratégia é criar o *arquivo de template*. No caso, o chamaremos de `_template.xhtml`. Nessa página, teremos o código do cabeçalho e do rodapé:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">

    <h:head>
        <title>Sistema de Notas Fiscais - UberDist</title>
        <h:outputStylesheet library="css" name="style.css" />
    </h:head>
    <h:body>
        <div id="cabecalho">
            <h:graphicImage library="imagens" name="logo-uber.png"
                           id="logoCompany" />
        <div id="usuarioLogado">
            <h:form rendered="#{usuarioLogadoBean.logado}">
                Logado como: #{usuarioLogadoBean.usuario.login}
                <h:commandLink value="[Sair]"
                               action="#{loginBean.logout}" />
            </h:form>
        </div>
    </div>

    <!-- AQUI PRECISAREMOS DO CONTEÚDO -->

    <div id="rodape">
```

```

Copyright 2013.
Todos os Direitos reservados a --Seu nome aqui--.
</div>
</h:body>
</html>

```

O NOME DO ARQUIVO DO TEMPLATE

É comum encontrar o nome do arquivo de template com o prefixo `_`. Não existe uma regra definida sobre isso, e tampouco uma convenção. Na prática, é comum encontrarmos diversos estilos diferentes para nomear os templates, inclusive organizando-os em diretórios separados.

Também é comum colocar o arquivo de template na pasta `WEB-INF` para proibir o acesso direto pelo navegador.

Repare que no código acima não informamos qual será o código que teremos como "corpo" da nossa página. Esse é justamente o papel do *arquivo de template*. Ele é um modelo que será seguido por outras páginas. O próximo passo é indicarmos em qual lugar será inserido código.

No nosso caso, precisamos que no nosso template códigos sejam inseridos entre o cabeçalhos e o rodapé. Para isso, precisaremos utilizar a Taglib do Facelets que importamos em nossa página com a instrução seguinte:

```

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">

    ...

</html>

```

Agora que importamos a Taglib do Facelets com o prefixo `ui`, basta indicarmos o local em que algum conteúdo deverá ser incluído. Para isso, basta indicarmos na página com a Tag `ui:insert` e darmos um nome para essa inserção através do atributo `name` como no código abaixo:

```
<ui:insert name="corpo" />
```

Dessa forma, a página `_template.xhtml` ficará com o seguinte código:

```

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">

    <h:head>
        <title>Sistema de Notas Fiscais - UberDist</title>
        <h:outputStylesheet library="css" name="style.css" />
    </h:head>
    <h:body>

```

```

<div id="cabecalho">
    <h:graphicImage library="imagens" name="logo-uber.png"
        id="logoCompany" />
    <div id="usuarioLogado">
        <h:form rendered="#{usuarioLogadoBean.logado}">
            Logado como: #{usuarioLogadoBean.usuario.login}
            <h:commandLink value="[Sair]"
                action="#{loginBean.logout}" />
        </h:form>
    </div>
</div>

<div id="conteudo">
    <ui:insert name="corpo" />
</div>

<div id="rodape">
    Copyright 2013.
    Todos os Direitos reservados a --Seu nome aqui--.
</div>
</h:body>
</html>

```

OUTRAS TECNOLOGIAS PARA TEMPLATING

Um dos problemas que temos ao trabalhar com JSP é a falta de suporte à criação de templates. Para resolver esse problema surgiram alternativas ao JSP, as mais famosas hoje em dia são o Freemarker e o Velocity que fazem inclusive otimizações na performance para a renderização das páginas, adicionando caches aos templates. O Tiles também auxilia o trabalho com templates.

Agora que já definimos o nosso *arquivo de template*, precisamos criar os seus clientes que irão inserir o conteúdo no `ui:insert` chamado `corpo` que acabamos de colocar no `_template.xhtml`. Para isso, podemos remover o código do cabeçalho e do rodapé do arquivo `login.xhtml`.

O próximo passo é, dentro da tag `h:body`, informarmos que a página `login.xhtml` vai compor o `_template.xhtml`, completando-o. Para isso, basta utilizarmos a tag `ui:composition` indicando no atributo `template` qual o template que será utilizado, como abaixo:

```

<ui:composition template="/_template.xhtml">

</ui:composition>

```

Em seguida, devemos indicar qual trecho do `_template.xhtml` estaremos definindo, que no nosso caso é o trecho que chamamos de `corpo`. Indicamos isso através da Tag `ui:define` com o atributo `name` indicando qual o trecho que vamos definir, como abaixo:

```

<ui:define name="corpo">

</ui:define>

```

Por fim, dentro da Tag `ui:define` basta adicionarmos o conteúdo que desejamos inserir no `corpo`

da nossa página. O resultado final será algo parecido com:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">

    <ui:composition template="/_template.xhtml">
        <ui:define name="corpo">

            <h2>Login no sistema</h2>
            <h:form id="formlogin">
                <h:panelGrid columns="2" styleClass="campos">

                    <h:outputLabel value="Login:" />
                    <h:inputText value="#{loginBean.usuario.login}" />

                    <h:outputLabel value="Senha:" />
                    <h:inputSecret value="#{loginBean.usuario.senha}" />

                    <h:commandButton value="Efetuar Login"
                                      action="#{loginBean.efetuaLogin}" />
                </h:panelGrid>
            </h:form>

        </ui:define>
    </ui:composition>
</html>
```

9.4 EXERCÍCIOS: TEMPLATES COM FACELETS

- Crie o template em um novo arquivo chamado `_template.xhtml` e salve-o na pasta `WebContent`. Use como base do HTML que já fizemos para o cadastro de produtos. A diferença estará dentro da `div conteudo` onde você deverá usar a tag `<ui:insert>` para incluir dinamicamente o `corpo` do template. Também será preciso incluir o *xml namespace* dos Facelets na tag `html`:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">

    <h:head>
        <title>Sistema de Notas Fiscais - UberDist</title>
        <h:outputStylesheet library="css" name="style.css" />
    </h:head>
    <h:body>
        <div id="cabecalho">
            <h:graphicImage library="imagens" name="logo-uber.png"
                           id="logoCompany" />
            <div id="usuarioLogado">
                <h:form>
                    Logado como: #{usuarioLogadoBean.usuario.login}
                    <h:commandLink value="[Sair]"
                                   action="#{loginBean.logout}" />
                </h:form>
            </div>
        </div>
    </h:body>
</html>
```

```

        </div>
    </div>

    <div id="conteudo">
        <ui:insert name="corpo" />
    </div>

    <div id="rodape">
        Copyright 2013.
        Todos os Direitos reservados a --Seu nome aqui--.
    </div>
</h:body>
</html>

```

2. O próximo passo é alterar a página `login.xhtml` para utilizar o template que acabamos de criar.

- Altere a definição da tag `html`, incluindo o *xml namespace* dos Facelets:

```

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html">

    ...

</html>

```

- Remova a Tag `h:head` com seu respectivo conteúdo.
- Na tag `body`, remova os `div` de `cabecalho` e `rodape`, se houverem.
- Remova a abertura e o fechamento das tags `h:body` e do `div` de `conteúdo`.
- Envolva o conteúdo da página nas tags de composição do Facelets. Use a `ui:composition` para fazer a composição do `_template.xhtml` e a `<ui:define name="corpo">` para indicar o corpo da página:

```

<ui:composition template="/_template.xhtml">
    <ui:define name="corpo">

        <!-- conteúdo da antiga div "conteúdo" vai
            aqui dentro com o formulário de login -->

    </ui:define>
</ui:composition>

```

3. Acesse a página de login pelo navegador e verifique se tudo continua funcionando como anteriormente.
4. Faça as mesmas alterações para o `produto.xhtml` também usar o template.
5. O que acontece se precisarmos mudar o ano do Copyright que está no rodapé? Em quantos lugares precisaremos mudar agora? Faça um teste e altere o seu rodapé.
6. Faça com que a mensagem indicando que o usuário está logado fique no template. Com isso você precisará tomar cuidado para que, na tela de login, a mensagem não apareça caso o usuário não esteja logado. Dica: use o atributo `rendered` visto antes.

Saber inglês é muito importante em TI

galandra

O **Galandra** auxilia a prática de inglês através de *flash cards* e *spaced repetition learning*. Conheça e aproveite os preços especiais.

[Pratique seu inglês no Galandra.](#)

9.5 PARA SABER MAIS: CAMPOS PADRÕES NO TEMPLATE

Quando criamos um template, é possível definirmos partes do mesmo que possuirão algum conteúdo caso as páginas que deverão redefini-lo não o façam. Para isso, basta adicionar esse conteúdo dentro da Tag `ui:insert` e, automaticamente, o Facelets reaproveitará esse conteúdo nas páginas caso nenhum outro seja passado.

9.6 EXERCÍCIOS OPCIONAIS: VALORES DEFAULT NO TEMPLATE

1. Utilize o `_template.xhtml` para definir também o título das páginas que é exibido na barra de título dos navegadores. Faça com que a tela de Login possua um título padrão, e com que a página de produto tenha no título uma mensagem específica dela.

9.7 COMPONENTES CUSTOMIZADOS COM FACELETS

Uma das grandes queixas dos desenvolvedores na versão 1.x do JSF era a grande complexidade para criar componentes próprios. Era preciso conhecer todos os detalhes sobre o ciclo da vida e da API JSF. Essa complexidade em si continua, mas com a entrada de Facelets podemos utilizar um recurso para criar novos componentes baseados em outros já existentes, tudo declarativamente, sem mexer na API. Essa maneira de definir componentes é chamada **composite-componentes**.

Os composite-componentes são parecidos com os tag-files do JSP. A ideia básica é criar um arquivo que use componentes JSF existentes e receba alguns parâmetros (atributos). Os atributos são usados para customizar o componente, por exemplo a `ID` ou `value` do mesmo. O arquivo pronto podemos usar como se fosse um componente novo.

Por exemplo, é muito comum usarmos os componentes `h:message` , `h:outputLabel` e `h:inputText` juntos, dentro de um formulário:

```
<h:message for="preco" />
<h:outputLabel for="preco" value="Preço:" />
<h:inputText id="preco" value="#{produtoBean.preco}" />
```

Para não repetir os três componentes podemos aproveitar os composite-componentes . Ou seja, definir um novo componente composto pelos três já existentes. Para isso é preciso criar uma pasta dentro da pasta padrão resources , por exemplo caelum . A pasta caelum seria então o nosso tagdir, onde todos os componentes customizados ficariam. Nessa pasta, cada composite-componentes será representado por um arquivo xhtml , por exemplo: campoTexto.xhtml . Nele faremos a definição do componente customizado.

Para a definição do componente é preciso usar um novo XML namespace na definição da tag html :

```
<html xmlns="http://www.w3.org/1999/xhtml"
      ...
      xmlns:cc="http://java.sun.com/jsf/composite">
</html>
```

Esse namespace cc serve para declarar a interface e a implementação do componente customizado. Na interface é preciso declarar tudo o que o componente pode receber como parâmetros (atributos) e na implementação, define-se o componente propriamente dito usando os atributos.

Veja como ficar a declaração inteira, com o namespace, a interface e a implementation:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:cc="http://java.sun.com/jsf/composite">

<cc:interface>
    <cc:attribute name="id" required="true" />
    <cc:attribute name="label" required="true" />
    <cc:attribute name="value" />
</cc:interface>

<cc:implementation>
    <h:message for="#{cc.attrs.id}" />
    <h:outputLabel value="#{cc.attrs.label}" for="#{cc.attrs.id}" />
    <h:inputText id="#{cc.attrs.id}" value="#{cc.attrs.value}" />
</cc:implementation>
</html>
```

O componente declara três atributos: id , label e value , onde id e label são atributos obrigatórios. Por sua vez, a implementação refere-se aos atributos através da expression language, por exemplo #{cc.attrs.id} ou #{cc.attrs.value} . Isso significa que o componente customizado será criado baseado-se nos atributos atuais.

Como resultado podemos usar o componente campoTexto , bastando importar o namespace caelum :

```
xmlns:caelum="http://java.sun.com/jsf/composite/caelum"
```

e aproveitar o novo componente:

```
<caelum:campoTexto id="nome" label="Nome:"
                     value="#{produtoBean.produto.preco}" />
```

Seus livros de tecnologia parecem do século passado?



Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**. Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

[Casa do Código, Livros de Tecnologia.](#)

9.8 EXERCÍCIOS OPCIONAIS: COMPOSITE-COMPONENTES

1. Crie uma nova pasta chamada **caelum** dentro da pasta **resources**. Dentro da pasta **caelum** crie um novo arquivo **campoTexto.xhtml**
2. Abra o arquivo **campoTexto.xhtml** e nele coloque o namespace padrão dos componentes JSF (`xmlns:h="http://java.sun.com/jsf/html"` e `xmlns:f="http://java.sun.com/jsf/core"`).

Também adicione o namespace **cc**:

```
xmlns:cc="http://java.sun.com/jsf/composite"
```

3. Adicione no arquivo a interface e a implementação do composite-componente. Dentro do tag **html** crie:

```
<cc:interface>
    <cc:attribute name="id" required="true" />
    <cc:attribute name="label" required="true" />
    <cc:attribute name="value" />
</cc:interface>

<cc:implementation>
    <h:message for="#{cc.attrs.id}" />
    <h:outputLabel value="#{cc.attrs.label}" for="#{cc.attrs.id}" />
    <h:inputText id="#{cc.attrs.id}" value="#{cc.attrs.value}" />
</cc:implementation>
```

4. Na página **index.xhtml** aplique o seu componente customizado. Para isso importe o namespace **caelum**:

```
xmlns:caelum="http://java.sun.com/jsf/composite/caelum"
```

e use dentro do formulário o componente:

```
<caelum:campoTexto id="nome" label="Nome:" 
    value="#{produtoBean.produto.nome}" />
```

5. Não esqueça de liberar sua página `index.xhtml` no **Autorizador.java** para garantir o êxito nos testes.

```
public void afterPhase(PhaseEvent event) {  
    FacesContext context = event.getFacesContext();  
  
    // código omitido  
  
    if ("/index.xhtml".equals(context.getViewRoot().getViewId())) {  
        return;  
    }  
}
```

6. Acesse a página `index.xhtml` e verifique o resultado.

7. **Desafio:** Crie um componente próprio que representa um botão (`h:commandButton`) que já utiliza o ajax por padrão. Transforme o botão seguinte em um composite-componente :

```
<h:commandButton action="#{bean.action}" value="Grava">  
    <f:ajax execute="@form" render="@form" />  
</h:commandButton>
```

O componente customizado deve se chamar **ajaxButton**. Veja um exemplo:

```
<caelum:ajaxButton action="#{testBean.grava}" value="clique-me"/>
```

FORMULÁRIO MASTER/DETAIL STATEFUL

"Ser líder é como ser uma dama: se você precisa provar que é, então você não é" -- Margareth Thatcher

Nesse capítulo, vamos ampliar nosso sistema, para que ele fique mais complexo e próximo do que enxergamos no dia a dia, inserindo mais componentes visuais.

10.1 CADASTRO DE NOTAS FISCAIS

Umas das exigências do projeto da UberDist é a emissão de notas fiscais para controlar as vendas feitas. Um cliente pode efetuar a compra de uma quantidade N de um determinado **produto**, e obrigatoriamente todos esses **itens** devem compor uma **nota fiscal** que foi emitida a um determinado cliente.

Criamos as classes que atendem essas necessidades no momento da criação do projeto. Precisamos criar agora a funcionalidade de efetivação de venda em nosso sistema para que os funcionários possam cadastrar as vendas e emitir as notas fiscais.

Antes de começarmos a definir a tela da funcionalidade de efetivar compra, precisamos definir o **Managed Bean** que fará a ligação com os componentes da tela. O *Managed Bean* será chamado de **NotaFiscalBean** e terá um objeto **NotaFiscal** que é a própria nota fiscal que estará sendo criada e um objeto **Item** para representar cada um dos itens da nota fiscal que serão adicionados um por vez. Como estamos usando o CDI para gerenciar nossos *Managed Beans*, vamos usar as anotações `@Named` e

`@RequestScoped` dos pacotes do CDI.

```
@Named  
@RequestScoped  
public class NotaFiscalBean {  
  
    private Item item = new Item();  
    private NotaFiscal notaFiscal = new NotaFiscal();  
  
    public Item getItem() {  
        return item;  
    }  
  
    public NotaFiscal getNotaFiscal() {  
        return notaFiscal;  
    }  
}
```

Uma vez criado o *Managed Bean*, vamos criar a tela para efetivação de vendas em nosso sistema que será chamada de `notafiscal.xhtml` e dividiremos a mesma em 3 partes.

Na primeira parte, ficará o cabeçalho da nota fiscal, que contém a data e o CNPJ do cliente.

```
<fieldset>  
    <legend>Dados da Nota</legend>  
  
    <h:outputLabel value="CNPJ do Cliente:" />  
    <h:inputText value="#{notaFiscalBean.notaFiscal.cnpj}" />  
  
    <h:outputLabel value="Data:" />  
    <h:inputText value="#{notaFiscalBean.notaFiscal.data.time}">  
        <f:convertDateTime pattern="dd/MM/yyyy"  
            timeZone="America/Sao_Paulo" />  
    </h:inputText>  
</fieldset>
```

O `<h:inputText>` que corresponde ao CNPJ da nota fiscal não contém nenhuma novidade, ele tem a mesma estrutura dos campos que criamos na adição dos produtos. O que está diferente é o `<h:inputText>` que corresponde ao campo **data** da nota fiscal.

O atributo `data` da classe `NotaFiscal` é do tipo `java.util.Calendar` e sabemos que o protocolo HTTP trafega somente texto, por isso é necessário que seja feita uma conversão. Como foi visto no capítulo das fases do JSF, ele possui uma fase de conversão bem definida e além disso alguns conversores já prontos.

No `<h:inputText>` correspondente a data da nota fiscal, utilizaremos o conversor `<f:convertDateTime>`, e nele especificaremos o padrão de data desejado através do atributo `pattern="dd/MM/yyyy"`. Precisamos também ajustar o *time zone* em que estamos trabalhando, senão o JSF sempre utiliza um time zone padrão, e que acaba causando uma conversão inconsistente. Resolvemos este problema ajustando o `time zone` através do atributo `timeZone="America/Sao_Paulo"`.

Outro detalhe, não menos importante, é que a conversão é feita para um objeto do tipo

`java.util.Date`, por isso fazemos o *binding* com o `time` da propriedade `data` presente na nota fiscal.

Para efetivar uma venda, precisamos salvar a nota fiscal no banco de dados. O primeiro passo é injetar o `NotaFiscalDao` dentro do *Managed Bean*:

```
@Named  
@RequestScoped  
public class NotaFiscalBean {  
  
    @Inject  
    private NotaFiscalDao notaFiscalDao;  
  
    ...  
  
}
```

O segundo passo é criar um método dentro do `NotaFiscalBean` que usa o DAO para gravar a nota fiscal.

Esse método fará uma chamada ao método `adiciona` do DAO, que é responsável pela adição no banco de dados. Além disso, ao final, precisamos limpar os atributos `notaFiscal` e `item` para podermos adicionar outros itens e notas fiscais.

```
public void gravar() {  
    notaFiscalDao.adiciona(notaFiscal);  
  
    this.notaFiscal = new NotaFiscal();  
    this.item = new Item();  
}
```

Agora, basta fazermos o *binding* do método `gravar` com o `<h:commandButton>` através do atributo `action`:

```
<h:commandButton value="Gravar" action="#{notaFiscalBean.gravar}" />
```

Agora é a melhor hora de respirar mais tecnologia!



Se você está gostando dessa apostila, certamente vai aproveitar os **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum. Programação, Mobile, Design, Infra, Front-End e Business! Ex-aluno da Caelum tem 15% de desconto, siga o link!

[Conheça a Alura Cursos Online.](#)

10.2 EXERCÍCIOS: MASTER (NOTAFISCAL)

- Vamos criar a funcionalidade de cadastro de notas fiscais. Crie uma nova classe `NotaFiscalBean` no pacote `br.com.caelum.notasfiscais.mb`. Por enquanto, ela terá apenas as propriedades `notaFiscal` e `notaFiscalDao`, além do método que salva as notas no banco de dados. Já vamos usar CDI desde inicio, por isso tenha cuidado com as importações.

```

@Named
@RequestScoped
public class NotaFiscalBean {

    private NotaFiscal notaFiscal = new NotaFiscal();

    @Inject
    private NotaFiscalDao notaFiscalDao;

    public void gravar() {
        this.notaFiscalDao.adiciona(notaFiscal);

        this.notaFiscal = new NotaFiscal();
    }

    public NotaFiscal getNotaFiscal() {
        return notaFiscal;
    }
}

```

- Crie o arquivo `notafiscal.xhtml` no diretório `WebContent`, que será o arquivo base para a tela. Vamos utilizar o conceito de templates que aprendemos no capítulo anterior para reaproveitarmos o cabeçalho e o rodapé.

Declare a tag `html` importando as tags do JSF (`core`, `html` e `facelets`). Aproveite e importe o template que fizemos e defina a tag corpo:

```

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">

    <ui:composition template="/_template.xhtml">
        <ui:define name="corpo">

            <!-- O conteúdo vai aqui -->

            </ui:define>
        </ui:composition>
    </html>

```

- Adicione no corpo da página `notafiscal.xhtml` o formulário com os campos de cadastro de notas fiscais:

```

<h:form>
    <fieldset>
        <legend>Dados da nota</legend>

        <h:outputLabel value="CNPJ do Cliente:" />
        <h:inputText value="#{notaFiscalBean.notaFiscal.cnpj}" />

        <h:outputLabel value="Data:" />

```

```

<h:inputText value="#{notaFiscalBean.notaFiscal.data.time}">
    <f:convertDateTime pattern="dd/MM/yyyy"
        timeZone="America/Sao_Paulo" />
</h:inputText>
</fieldset>

<h:commandButton value="Gravar" action="#{notaFiscalBean.gravar}" />
</h:form>

```

Colocamos as tags html `<fieldset>` e `<legend>` para manter nosso layout mais amigável.

4. Acesse o cadastro de notas no navegador. Repare que ainda está incompleto, sem a funcionalidade de guardar os itens da nota. Mas já deve ser possível cadastrar notas fiscais sem itens (verifique no banco de dados).

10.3 DETALHES DA NOTA

Na segunda parte da nossa tela de notas fiscais, deveremos adicionar itens na nota. Um item da nota fiscal é composto por um determinado **produto** e uma **quantidade**. A tela terá um componente que permita selecionarmos o produto desejado e digitar a quantidade que estamos vendendo dele.

O componente utilizado para selecionar um determinado produto será um `<h:selectOneMenu>`, equivalente a qualquer *Combo Box* presente em aplicações Desktop.

```

<fieldset>
    <legend>Dados do item</legend>

    <h:outputLabel value="Produto:" />
    <h:selectOneMenu value="#{notaFiscalBean.idProduto}">
        <f:selectItems value="#{produtoBean.produtos}" var="produto"
            itemValue="#{produto.id}" itemLabel="#{produto.nome}" />
    </h:selectOneMenu>

    <h:outputLabel value="Quantidade:" />
    <h:inputText value="#{notaFiscalBean.item.quantidade}" />

    <h:commandButton action="#{notaFiscalBean.guardaItem}"
        value="Guardar Item" />
</fieldset>

```

Umas das grandes vantagens do JSF 2.2 com relação a outros frameworks Web é a possibilidade de adicionarmos diretamente os objetos que desejamos popular em nosso *Combo Box*. No código acima, na tag `<f:selectItems>`, fizemos o *binding* diretamente com os objetos do tipo `Produto` retornados pelo *Managed Bean* `ProdutoBean`.

Além do *binding* feito entre o campo quantidade da tela e a propriedade quantidade, necessitamos também do `id` do produto que foi selecionado, para efetuarmos a busca dele no banco de dados e adicioná-lo dentro do item que está sendo adicionado na nota fiscal. Para isto, criamos uma propriedade `idProduto` dentro do *Managed Bean* `NotaFiscalBean`.

Existe a necessidade de criarmos um método que faça essa lógica de buscar o produto no banco de

dados, depois incluí-lo dentro do item e por último adicionar este item na nota fiscal. Além disso, precisamos limpar o atributo `item` para podermos incluir diferentes itens dentro da nota fiscal:

```
@Named  
@RequestScoped  
public class NotaFiscalBean {  
  
    private Item item = new Item();  
    private NotaFiscal notaFiscal = new NotaFiscal();  
    private Long idProduto;  
  
    @Inject  
    private NotaFiscalDao notaFiscalDao;  
  
    @Inject  
    private ProdutoDao produtoDao;  
  
    public void guardaItem() {  
  
        Produto produto = produtoDao.buscaPorId(idProduto);  
        item.setProduto(produto);  
        item.setValorUnitario(produto.getPreco());  
  
        notaFiscal.getItens().add(item);  
        item.setNotaFiscal(notaFiscal);  
  
        item = new Item();  
        idProduto = null;  
    }  
  
    // getters e setters  
}
```

Fizemos a adição dos itens na nota fiscal, porém ainda não conseguimos visualizá-los. Uma técnica muito utilizada quando estamos desenvolvendo aplicações desktop é adicionar uma tabela logo abaixo do formulário de adição de itens, exibindo-os conforme adicionados.

Faremos uma tabela similar as outras que já criamos até aqui, porém agora fazendo o *binding* com os itens da nota fiscal do *Managed Bean* `NotaFiscalBean`.

```
<h:dataTable value="#{notaFiscalBean.notaFiscal.itens}" var="item"  
    styleClass="dados" rowClasses="impar,par">  
    <h:column>  
        <f:facet name="header">Produto</f:facet>  
        #{item.produto.nome}  
    </h:column>  
    <h:column>  
        <f:facet name="header">Preço</f:facet>  
        <h:outputText value="#{item.valorUnitario}">  
            <f:convertNumber pattern="R$ #0.00"/>  
        </h:outputText>  
    </h:column>  
    <h:column>  
        <f:facet name="header">Quantidade</f:facet>  
        #{item.quantidade}  
    </h:column>  
    <h:column>  
        <f:facet name="header">Valor</f:facet>  
        <h:outputText value="#{item.total}" >
```

```

        <f:convertNumber pattern="R$ #0.00"/>
    </h:outputText>
</h:column>
</h:dataTable>

```

10.4 EXERCÍCIOS: DETAIL (ITEM)

- No `NotaFiscalBean`, vamos precisar de mais três propriedades para nos ajudar no cadastro dos itens. Primeiro, uma propriedade `Item` que manterá os dados do item atualmente sendo cadastrado.

Mas como haverá um relacionamento de item com `Produto`, precisamos de uma outra propriedade que mantenha o `idProduto` para o relacionamento:

A terceira é o `ProdutoDao` que será preciso para carregar um produto pela `id`.

Crie as propriedades. Para os atributos `item` e `idProduto` gere também os *getters/setters*:

```

private Item item = new Item();
private Long idProduto;

@.Inject
private ProdutoDao produtoDao;

// getters e setters

```

- A última modificação no *Managed Bean* `NotaFiscalBean` é a criação de um método responsável por fazer a lógica de adição do item na nota fiscal.

Ele deve usar o `idProduto` para buscar o respectivo `Produto` no banco de dados e associá-lo ao `item` atual. Por fim, esse método deve associar a nota fiscal ao item sendo cadastrado, além de limpar o formulário de cadastro de itens:

```

public void guardaItem() {
    Produto produto = produtoDao.buscaPorId(idProduto);
    item.setProduto(produto);
    item.setValorUnitario(produto.getPreco());

    notaFiscal.getItens().add(item);
    item.setNotaFiscal(notaFiscal);

    item = new Item();
    idProduto = null;
}

```

- Na tela `notafiscal.xhtml` precisamos adicionar os campos para cadastro dos itens. Faça isso usando um novo `<fieldset>` logo abaixo do *fieldset* do cadastro de notas (e acima do botão de gravar as notas):

```

<fieldset>
    <legend>Dados do item</legend>

    <h:outputLabel value="Produto:> />

```

```

<h:selectOneMenu value="#{notaFiscalBean.idProduto}">
    <f:selectItems value="#{produtoBean.produtos}" var="produto"
        itemValue="#{produto.id}" itemLabel="#{produto.nome}" />
</h:selectOneMenu>

<h:outputLabel value="Quantidade:" />
<h:inputText value="#{notaFiscalBean.item.quantidade}" />

<h:commandButton action="#{notaFiscalBean.guardaItem}"
    value="Guardar Item" />
</fieldset>

```

4. Crie uma tabela para exibir os itens que já foram adicionados na nota fiscal. Adicione um `<h:dataTable>` no final do formulário de adição de notas e itens.

```

<h2>Itens da nota fiscal</h2>
<h:dataTable value="#{notaFiscalBean.notaFiscal.itens}" var="item"
    styleClass="dados" rowClasses="impar,par">
    <h:column>
        <f:facet name="header">Produto</f:facet>
        #{item.produto.nome}
    </h:column>
    <h:column>
        <f:facet name="header">Valor unitário</f:facet>
        <h:outputText value="#{item.valorUnitario}">
            <f:convertNumber type="currency" currencySymbol="R$ "
                locale="pt_BR" />
        </h:outputText>
    </h:column>
    <h:column>
        <f:facet name="header">Quantidade</f:facet>
        #{item.quantidade}
    </h:column>
    <h:column>
        <f:facet name="header">Total</f:facet>
        <h:outputText value="#{item.total}">
            <f:convertNumber type="currency" currencySymbol="R$ "
                locale="pt_BR" />
        </h:outputText>
    </h:column>
</h:dataTable>

```

5. Acesse a página pelo endereço: <http://localhost:8080/fj26-notas-fiscais/notafiscal.xhtml>, o resultado deve ser similar à imagem a seguir:

The screenshot shows a web application interface for managing invoices. At the top, there's a header bar with the URL <http://localhost:8080/fj26-notas-fiscais/notafiscal.xhtml>. Below the header, the page title is "Sistema de Notas Fiscais - Uber...". The main content area has a blue header bar with the text "UBERDIST".

The form is divided into several sections:

- Dados da nota**: Contains fields for "CNPJ Cliente" (with an input field) and "Data" (with an input field).
- Dados do produto**: Contains a "Produto" dropdown menu set to "Cerveja" and a "Quantidade" input field.
- Itens da nota fiscal**: A table with four columns: "Produto", "Preco", "Quantidade", and "Valor". There is also a "Gravar" (Save) button at the bottom of the table.

At the very bottom of the page, there is a footer bar with the text "Copyright 2010. Todos os Direitos reservados a --Seu nome aqui--".

Tente adicionar mais do que um item na nota. Tudo está funcionando como esperado?

Editora Casa do Código com livros de uma forma diferente



Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.

Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.](#)

10.5 O ESTADO DA TELA E O @VIEWSCOPED

A funcionalidade de adicionar um item na nota fiscal não está funcionando corretamente. Ela tem um problema grave que acontece quando tentarmos adicionar mais do que um item. O nosso formulário mostra apenas o último item.

Quando usamos o botão **Guardar item** enviamos um *request* com os dado do item e a `id` do produto. No servidor, dentro do método `guardaItem`, guardamos o item na lista da nota fiscal. A nota por sua vez é um atributo do *Managed Bean*:

```
@Named  
@RequestScoped  
public class NotaFiscalBean {  
  
    private NotaFiscal notaFiscal = new NotaFiscal();  
  
    ...  
}
```

Repare que o `NotaFiscalBean` está no escopo `@RequestScoped`. Isso significa que o CDI criará em cada *request* um novo *Managed Bean* e consequentemente uma nova nota fiscal. Por isso vimos apenas um item na nota.

Podemos deixar o `NotaFiscalBean` no escopo da sessão, assim o CDI criará o *bean* apenas no início da sessão HTTP. A problema aqui é que várias abas dentro do mesmo navegador vão compartilhar o mesmo *bean* no lado do servidor. Com isso, um formulário vai alterar os dados do outro.

Queremos uma solução elegante, que dure mais do que um *request*, mas isole corretamente cada formulário do usuário. Para resolver este problema o JSF 2 introduziu um escopo de tela, ou **ViewScoped**.

Guardar estado entre requisições costuma ser trabalhoso na Web tradicional. Mas a característica *stateful* do JSF traz esse recurso pronto para trabalharmos. A anotação `@ViewScoped` do JSF 2 permite que certo *Managed Bean* seja preservado enquanto a tela (*view*) estiver ativa. Isso é feito guardando temporariamente o objeto na sessão do usuário e automaticamente removendo-o quando o usuário sair da tela atual. Exatamente o que estamos procurando.

Cuidado: caso os *Managed Beans* sejam gerenciados pelo JSF e não pelo CDI (ou seja, usamos `@ManagedBean` invés de `@Named`) devemos usar `@ViewScoped` do pacote `javax.faces.bean`:

```
import javax.faces.bean.ManagedBean;
import javax.faces.bean.ViewScoped;

@ManagedBean
@ViewScoped
public class NotaFiscalBean {

    ...

}
```

10.6 NOVIDADE DO JAVA EE 7: VIEWSCOPED COM CDI

A nossa aplicação já usa o CDI e todos os *beans* estão sendo gerenciados por ele. É por isso que não devemos usar `@ViewScoped` do pacote `javax.faces.bean`. No Java EE 7, através do JSF 2.2, foi introduzido um outro pacote, `javax.faces.view`, que devemos utilizar no caso de estarmos trabalhando com CDI. Só através dessa anotação habilitamos o *ViewScope* para CDI:

```
import javax.inject.Named;
import javax.faces.view.ViewScoped;

@Named
@ViewScoped
public class NotaFiscalBean {

    private NotaFiscal notaFiscal = new NotaFiscal();

    ...

}
```

PROBLEMAS DO @VIEWSCOPED

Embora seja a solução mais adequada para o JSF, usar **ViewScope** não é inteiramente positivo. O primeiro problema está na sobrecarga da sessão do usuário. Apesar de guardar os dados só no escopo da tela, isso acarretará em todos os usuários atuais guardando a lista de objetos produto na memória do servidor. É o preço do *stateful*.

PRECISO PASSAR O ID HIDDEN?

Quem está acostumado a frameworks *action based* sempre lembra que, para suportar a edição, o formulário, de alguma forma, deve informar o **id** do elemento que está sendo editado. O jeito mais comum de fazer isso é incluir um campo *hidden* no formulário. Poderíamos ter feito isso no JSF:

```
<h:inputHidden value="#{produtoBean.produto.id}" />
```

Mas pela característica *stateful* do JSF e pelo nosso *Managed Bean* estar no *ViewScope*, isso não é necessário. O produto atualmente sendo editado está salvo na memória dentro do atributo do *ProdutoBean*, incluindo seu **id** carregado do banco.

Como nosso formulário edita apenas as outras propriedades, quando for submetido, o **id** continuará disponível e nosso método *grava* funcionará corretamente.

Note, porém, que isso só funciona porque estamos usando *ViewScope* e guardando estado! Em outros cenários o uso do `<h:inputHidden>` pode ser necessário.

10.7 EXERCÍCIOS: @VIEWSCOPED

1. Teste o problema exposto na seção anterior.

Para resolver, abra sua classe `NotaFiscalBean` e anote-a com `@ViewScoped` substituindo `@RequestScoped`. O pacote é `javax.faces.view`.

Reita o teste e veja a diferença.

Já conhece os cursos online Alura?



A **Alura** oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Programação, Front-end, Mobile, Design & UX, Infra e Business, com um plano que dá acesso a todos os cursos. Ex aluno da Caelum tem 15% de desconto neste link!

[Conheça os cursos online Alura.](#)

10.8 EXERCÍCIOS OPCIONAIS: AJAX

1. Implemente a funcionalidade de adição de itens e gravação de notas fiscais usando Ajax.

10.9 NOVIDADE DO JAVA EE 7: STATELESS VIEW NO JSF

Já vimos como configurar os escopos para os *Managed Beans*. Usamos as anotações `@RequestScoped` ou `ViewScoped` entre outras disponíveis. Enquanto tivermos um controle fino sobre o escopo do modelo, a árvore de componentes sempre fica na sessão HTTP.

O JSF 2.2 introduziu o conceito de *stateless view*. *Stateless* significa que a árvore de componentes é recriada em cada *request*, ou seja, não será guardada na sessão.

Para habilitar o *stateless view* basta adicionar a tag `f:view` com o atributo `transient="true"` na página `xhtml`:

```
<f:view transient="true" />
```

Ao acessar a página, o *input* escondido `javax.faces.ViewState`, que guarda normalmente a `id` da árvore, terá o valor `stateless`:

```
/><input type="hidden" name="javax.faces.ViewState" id="j_id1:javax.faces.ViewState:0"
value="stateless" autocomplete="off" />
```

Uma vantagem disso é que não recebemos mais o `ViewExpiredException`, mas estamos limitados ao `@RequestScoped`.

RECURSOS AVANÇADOS DE CDI

"A grandeza não consiste em receber honras, mas em merecê-las." -- Aristóteles

A integração do JSF 2 com CDI abre novas possibilidades para nosso projeto. Muita coisa pode ser simplificada com o novo modelo de desenvolvimento do Java EE 7.

11.1 INTERCEPTADORES DO CDI

Em diversos momentos surge a necessidade de interceptar a execução de métodos de negócio com códigos que não fazem parte da lógica principal. O melhor seria conseguir isolar esses aspectos e reaproveitá-los dinamicamente ao invés de ter que fazer essas chamadas em todo lugar.

Um bom exemplo é o código do gerenciamento de transações (início e *commit*). Outros exemplos poderiam ser auditoria, log, controle de exceções, entre outros.

Usando CDI, podemos usar **Interceptors** para implementar esse recurso.

Onde devo abrir e fechar minhas conexões e transações com o banco de dados?

Já utilizamos o CDI para realizar a injeção de do `EntityManager` e tratar do fechamento do mesmo no momento adequado. Mas podemos ir um pouco adiante e tratar o gerenciamento das transações também de maneira centralizada.

Geralmente, quando trabalhamos em uma aplicação Web, o ciclo de vida de uma conexão e uma transação é o tempo necessário para o processamento da requisição, que também é conhecido como **unidade de trabalho** (*unit of work*) onde faremos todo o processo para trabalhar com o banco de dados.

Queremos portanto, que seja aberta uma nova transação por requisição e que esta seja "comitada" ao final. Isso nos permitirá ainda resolver outros problemas que ainda temos. Conseguiremos, por exemplo, chamar diversas operações de uma mesma tela atomicamente, com a garantia do *rollback* caso alguma delas falhe.

Uma forma de resolvemos esse problema é através da criação de Filtros da API de *Servlets*. Ele poderia abrir o `EntityManager`, iniciar uma transação e repassar o processamento para o JSF. Por fim, após o fim da execução da lógica e da renderização da *View* para o usuário, a conexão e a transação são finalizadas.

TRATANDO AS ENTITYMANAGERS AO TRABALHAR COM A JPA

É sempre uma boa prática ao trabalhar em uma aplicação web com a JPA abrir um *EntityManager* e fechá-lo no começo ou no fim do processo de requisição. Como a JPA em vários pontos faz *Lazy Loading* dos objetos, é necessário que no momento em que a tela é renderizada a *EntityManager* esteja aberta. Uma das formas para resolver esse problema é o pattern *Open Session in View* (no caso da JPA chamado de *Open Entity Manager in View*) que é visto em detalhes no curso **Persistência com JPA, Hibernate e EJB light** (FJ-25).

Uma solução com CDI

A abertura e fechamento dos *EntityManagers* já foram resolvidas com o CDI, usando o `@Produces` e o `@Disposes`. Além deles, podemos usar um **interceptador do CDI** para executar o controle transacional.

O plano é criar uma forma de indicar ao CDI quais os momentos que precisamos de uma transação nova, e um interceptador cuidará de executar o código necessário para nós. Criaremos uma anotação customizada para ser usada nos métodos dos beans que precisam de controle transacional:

```
@Named  
@ViewScoped  
public class ProdutoBean {  
  
    @Transactional  
    public void grava() {  
        // ...  
    }  
}
```

Essa anotação `@Transactional` não existe. Será uma anotação nossa (customizada) para indicar esse tratamento com o CDI. É uma *annotation Java* comum, apenas com uma configuração a mais do CDI, a `@InterceptorBinding`:

```
@InterceptorBinding  
@Target({ElementType.TYPE, ElementType.METHOD})  
@Retention(RetentionPolicy.RUNTIME)  
public @interface Transactional {  
  
}
```

O próximo passo é criarmos o interceptador para gerenciar a transação nas classes que forem anotadas com `@Transactional`. Criamos uma classe que será anotada com `@Interceptor` do CDI. Além disso, ela deverá ser marcada com a mesma anotação que usaremos em nossos *beans*, a `@Transactional`. Sua implementação deverá receber o *EntityManager* injetado e gerenciar a transação:

```

@Interceptor
@Transactional
public class TransactionInterceptor {

    @Inject
    private EntityManager manager;

    // aqui vai o método que trata da transação
}

```

Ao implementarmos o nosso interceptador, precisamos criar o método que iniciará a transação e indicar que esse método é o que interceptará as requisições. Como nossa classe não implementa nenhuma interface, podemos atribuir qualquer nome para o método, que chamaremos de `intercept`. Esse método deve receber como parâmetro um objeto do tipo `InvocationContext` e retornar um `Object`. Para indicar que esse método é quem interceptará as invocações, precisamos anotá-lo com `@AroundInvoke`:

```

@Interceptor
@Transactional
public class TransactionInterceptor {

    @AroundInvoke
    public Object intercept(InvocationContext ctx) {
        // vamos implementar a lógica aqui
    }
}

```

O próximo passo é implementarmos a abertura da transação. Assim que ela for aberta, o processamento deve prosseguir para que o JSF processe a requisição. Para isso, utilizaremos o método `invoke` do `InvocationContext` que recebemos como parâmetro. Esse método retorna um objeto que é o retorno do nosso `ManagedBean`. Ou seja, se nosso `ManagedBean` retorna uma `String` "ok", o nosso interceptador vai recuperar essa `String` e para que o usuário seja redirecionado para o lugar correto, nosso `Interceptor` precisa retornar essa mesma `String`.

No entanto, precisamos também fazer o `commit` da transação antes do método `intercept` retornar algo. Dessa forma, a implementação será parecida com:

```

@AroundInvoke
public Object intercept(InvocationContext ctx) throws Exception {
    manager.getTransaction().begin();

    // Passando para o JSF tratar a requisição e pegando o
    // retorno da lógica
    Object resultado = ctx.proceed();

    manager.getTransaction().commit();
    return resultado;
}

```

Por fim, precisamos declarar o nosso interceptador no arquivo `beans.xml`:

```

<beans>
    <interceptors>
        <class>
            br.com.caelum.notasfiscais.interceptor.TransactionInterceptor
        </class>
    </interceptors>
</beans>

```

```
</class>
</interceptors>
</beans>
```

ROLLBACK

Este nosso exemplo não trata o *rollback* da transação, já que nosso foco está em aprender os recursos do CDI. Mas seria bastante simples adicionar esse comportamento ao interceptador.

Saber inglês é muito importante em TI

galandra O Galandra auxilia a prática de inglês através de *flash cards* e *spaced repetition learning*. Conheça e aproveite os preços especiais.

[Pratique seu inglês no Galandra.](#)

11.2 EXERCÍCIOS: TRABALHANDO COM CONEXÕES E TRANSAÇÕES

1. Crie uma nova anotação Java para marcar as classes que deverão ter controle transacional. Use a anotação `@InterceptorBinding` para configurá-la.

Como sugestão use o pacote `br.com.caelum.notasfiscais.tx`:

```
@InterceptorBinding
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
public @interface Transactional {
```

```
}
```

2. No mesmo package crie uma nova classe para conter nosso interceptador do CDI. Ela deve ser marcada com `@Interceptor` e `@Transactional`. Além disso, ela deverá receber um `EntityManager` injetado e ter um método anotado com `@AroundInvoke` que faz o controle transacional propriamente dito:

```
@Interceptor
@Transactional
public class TransactionInterceptor implements Serializable {

    @Inject
    private EntityManager manager;

    @AroundInvoke
```

```

public Object intercept(InvocationContext ctx) throws Exception{
    System.out.println("Abrindo a transação");
    manager.getTransaction().begin();

    // Passando para o JSF tratar a requisição e pegando o
    // retorno da lógica
    Object resultado = ctx.proceed();

    manager.getTransaction().commit();
    System.out.println("Comitando a transação");

    return resultado;
}
}

```

3. Configure o interceptador no XML do CDI. Abra o arquivo `beans.xml` e adicione:

```

<beans>
    <interceptors>
        <class>br.com.caelum.notasfiscais.tx.TransactionInterceptor</class>
    </interceptors>
</beans>

```

Tenha cuidado com o **nome do pacote do interceptador**.

4. Anote seu `ManagedBean` para que ele passe pelo nosso *interceptor*. Para isso, adicione a anotação `@Transactional` nos métodos do `ProdutoBean` e do `NotaFiscalBean` que precisam de controle transacional como o método `grava`.
5. Nos DAOs há ainda métodos que fazem um controle transacional dentro de cada método. **Retire** todo esse código de gerenciamento de transação dos DAOs.

Você precisa alterar as classes `ProdutoDao` e o `NotaFiscalDao`. Por exemplo, o método adiciona do `ProdutoDao` fica simples assim:

```

public class ProdutoDao implements Serializable {

    // atributos

    public void adiciona(Produto produto) {
        manager.persist(produto); // sem nenhum código de transação
    }
}

```

Faça a alteração para os demais métodos que possuem código relacionado ao gerenciamento de transações. **Apague estes códigos**.

6. Acesse novamente o sistema e adicione um novo Produto. Veja que as mensagens que deixamos dentro do método `intercept` estão aparecendo no console, ou seja, o nosso novo *interceptor* está sendo invocado corretamente.

11.3 DISCUSSÃO: TRANSAÇÕES DEMARCADAS OU TRANSAÇÃO POR REQUEST

Além de nos ajudar a aprender o recurso de interceptadores do CDI, o exemplo anterior levanta a questão de quando devemos ou não usar demarcação de transações. Nossa exemplo exige que marquemos os métodos que serão transacionais com uma anotação específica.

Apesar das vantagens, essa abordagem tem certas limitações. Não conseguimos, por exemplo, incluir a execução de dois métodos ou dois managed beans diferentes na mesma transação.

Uma abordagem comum na Web é trabalhar com uma transação por request, iniciando uma nova junto com a criação do `EntityManager` e comitando-a antes de fechar o EM. Pensando no nosso código CDI, conseguimos implementar facilmente esse comportamento fazendo o início da transação no método produtor de `EntityManager` e comitando a transação no método de fechamento (do `disposes`).

Essa abordagem funcionará muito bem a menos que tenhamos que tratar rollbacks. Geralmente queremos que a transação atual seja cancelada caso haja um erro ou exception na lógica de negócios. Porém, a abordagem usando os métodos de `@Produces` e `@Disposes` não permite que identifiquemos quando houve problemas na lógica de negócios. A única forma de tratar erros ocorridos na lógica de negócios é com os interceptadores.

Seria possível uma solução híbrida, com um interceptor apenas responsável por dar rollback nas transações em caso de erro na lógica de negócios. O início e commit da transação normal seriam colocados nos métodos produtores e de `dispose`, e o interceptor ficaria apenas com o rollback em caso de problemas. É uma solução menos elegante mas soluciona nosso problema de transação por request usando apenas CDI.

Outra forma de fazer uma transação por request seria com Filtros Web da API de Servlets. É a solução mais usada no mercado, e teria todas as vantagens de definir o escopo como request, não precisar de demarcação manual, permitir identificar erros da aplicação e fazer rollback. O problema é que os Filtros não são nem JSF nem CDI. É possível integrá-los mas nunca de forma tão elegante quanto uma solução CDI pura.

No curso FJ-25, abordamos o problema do gerenciamento de transações através do EJB. Dentro de um servidor de aplicação como JBoss ou Glassfish os _Enterprise Java Bean_s assumem a responsabilidade de cuidar parte da infraestrutura como o gerenciamento da transação. É uma solução portável, inclusive fora do JSF, que resolve os problemas do desenvolvedor de maneira fácil e transparente.

11.4 EVENTOS E OBSERVERS DO CDI

Uma das funcionalidades do CDI que ajudam em diminuir o acoplamento entre duas partes do sistema é sua API de eventos. É possível separar dois componentes do sistema de forma a se comunicarem via o disparo de eventos de negócio, seguindo o padrão observer.

A ideia é ter um componente que gera eventos para o CDI. E o CDI entrega esses eventos para componentes observadores que podem ter se registrado com a anotação `@Observes`.

Um exemplo prático seria propagar automaticamente o usuário quando ele faz o login. Outros sistemas poderiam estar interessados de receber uma notificação quando o usuário se loga na aplicação.

Para tal devemos injetar um `Event` na classe que dispara. Esse evento é parametrizado com o tipo da mensagem que queremos passar, por exemplo:

```
@Inject  
Event<Usuario> eventoLogin;
```

Agora só faltar gerar o evento pelo método `fire`. O `fire` recebe o objeto que queremos associar com o evento, aqui é o *usuario*:

```
eventoLogin.fire(usuario);
```

Quem gostaria de ser notificado e receber o *usuario*, declara o interesse através da anotação `@Observes`:

```
public class LoginListener {  
  
    public void onLogin(@Observes Usuario usuario) {  
        System.out.printf("Usuario %s se logou no sistema. " + usuario.getLogin());  
        //enviando email para outro sistema.  
    }  
}
```

Seus livros de tecnologia parecem do século passado?



Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**. Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

[Casa do Código, Livros de Tecnologia.](#)

11.5 EXERCÍCIOS: DISPARANDO EVENTOS COM CDI

1. Faça que o `LoginBean` dispara um evento com CDI.

Primeiro injete na classe o objeto `Event`:

```
@Inject  
Event<Usuario> eventoLogin;
```

No método `efetuaLogin`, dentro do if, gera o evento:

```
eventoLogin.fire(usuario);
```

2. No pacote `br.com.caelum.notasfiscais.listener` cria uma nova classe `LoginListener` que receberá a notificação:

```
public void onLogin(@Observes Usuario usuario) {
    System.out.printf("Usuario %s se logou no sistema. ", usuario.getLogin());
    //enviando email para outro sistema.
}
```

Faça um login no sistema é verifique se a mensagem aparece no console.

11.6 ATRIBUTOS PRODUTORES

Imagine que temos que implementar uma nova funcionalidade no sistema: toda vez que cadastrarmos um novo produto, um e-mail de notificação deve ser enviado para `comercial@uberdist.com.br`. Mas não queremos escrever esse endereço de e-mail no meio do código do `ProdutoBean` porque ele pode mudar muito no futuro ou podemos querer obtê-lo a partir de alguma configuração.

Queremos injetar o endereço de e-mail correto dentro do `ProdutoBean` para que não tenhamos acoplamento com o endereço real.

O e-mail é uma simples `String` que podemos produzir com `@Produces` e depois injetar com `@Inject`:

```
public class EmailFactory {

    @Produces
    private String emailComercial = "comercial@uberdist.com.br";
}
```

Repare que usamos um *atributo produtor*. É bem parecido com métodos produtores que tínhamos antes mas ele serve bem para casos mais simples como esse de um valor estático.

E usar esse e-mail no `ProdutoBean`:

```
public class ProdutoBean {

    @Inject
    private String email;

    // ... outros atributos e métodos

    public void grava() {
        // ... lógica de gravação

        System.out.println("Notifica " + email + ": novo Produto cadastrado.");
    }
}
```

Apenas injetamos a `String` e fingimos o envio do e-mail na saída. A ideia aqui é observar a injeção da `String` pelo CDI e o uso de um atributo produtor.

11.7 QUALIFIERS DO CDI

E se tivermos mais de um e-mail possível no sistema? Imagine que novas notas fiscais devam ser notificadas ao e-mail `financeiro@uberdist.com.br`. Como diferenciar os e-mails?

Na produção, até podemos ter dois atributos independentes:

```
public class EmailFactory {  
    @Produces  
    private String emailComercial = "comercial@uberdist.com.br";  
  
    @Produces  
    private String emailFinanceiro = "financeiro@uberdist.com.br";  
}
```

Mas na hora da injeção, qual `String` o CDI injetará? Nenhuma, o contexto nem sobe, há uma *ambiguidade* de tipos na nossa aplicação.

Em diversas situações nos encontramos nesse cenário: temos mais de um objeto do mesmo tipo disponível para injeção e queremos injetar cada um em um momento diferente. O exemplo do e-mail é um deles. Outro, mais complexo, poderia ser injetar `EntityManagers` diferentes para usuários de diferentes clientes da aplicação.

Como diferenciar os objetos na hora da produção e da injeção? Seguindo a linha sempre *typesafe* do CDI que evita o uso de nomes/ids pra diferenciar as injeções, usaremos **qualifiers**.

Qualifiers são anotações específicas de negócio que vamos criar para diferenciar os objetos. Por serem código Java, nos garantem type safety, auto completar, compilação etc. Imagine duas anotações, `@EmailComercial` e `@EmailFinanceiro`. Usaremos-nas para diferenciar a produção:

```
public class EmailFactory {  
    @Produces @EmailComercial  
    private String emailComercial = "comercial@uberdist.com.br";  
  
    @Produces @EmailFinanceiro  
    private String emailFinanceiro = "financeiro@uberdist.com.br";  
}
```

E usaremos também no momento da injeção:

```
public class ProdutoBean {  
    @Inject @EmailComercial  
    private String email;  
  
    // ... outros atributos e métodos  
}  
  
public class NotaFiscalBean {  
    @Inject @EmailFinanceiro
```

```
private String email;  
// ... outros atributos e métodos  
}
```

Para isso funcionar, só precisamos criar as anotações e configurá-las como qualifiers. Isso é feito anotando a própria anotação com `@Qualifier`:

```
@Qualifier  
@Retention(RetentionPolicy.RUNTIME)  
@Target(ElementType.FIELD)  
public @interface EmailFinanceiro {  
}
```

Agora é a melhor hora de respirar mais tecnologia!



Se você está gostando dessa apostila, certamente vai aproveitar os **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum. Programação, Mobile, Design, Infra, Front-End e Business! Ex-aluno da Caelum tem 15% de desconto, siga o link!

[Conheça a Alura Cursos Online.](#)

11.8 INJEÇÃO COM @ANY

Há alguns qualifiers já prontos no CDI que servem propósitos especiais. O `@Any`, por exemplo, permite que accessemos, de uma única vez, todos os valores possíveis:

```
@Inject @Any  
private Instance<String> todosEmails;
```

Esse tipo `Instance` do CDI nos permite depois iterar nos valores:

```
for (String email: todosEmails) {  
    System.out.println("Notifica " + email);  
}
```

11.9 EXERCÍCIOS: QUALIFIERS

1. Crie duas **anotações** Java para serem nossos qualificadores: `@EmailComercial` e `@EmailFinanceiro` (use o package `util`).

Ambas devem funcionar em *RUNTIME*, podendo ser usadas em *FIELD_s e são _qualifiers*:

```
@Qualifier  
@Retention(RetentionPolicy.RUNTIME)
```

```
@Target(ElementType.FIELD)
public @interface EmailComercial {
}
```

2. Crie uma classe produtora para os valores dos e-mails. Use *atributos produtores*:

```
public class EmailFactory {
    @Produces @EmailComercial
    private String emailComercial = "comercial@uberdist.com.br";

    @Produces @EmailFinanceiro
    private String emailFinanceiro = "financeiro@uberdist.com.br";
}
```

3. Injete o e-mail comercial na classe `LoginListener`. Use os qualificadores criados antes para diferenciar as injeções.

Modifique o método `onLogin` do `LoginListener` para imprimir uma mensagem no console simulando o envio de e-mail de notificação de cada caso específico.

Teste sua aplicação e note as injeções qualificadas acontecendo. O CDI saberá qual valor injetar em cada momento.

11.10 MENOS ANOTAÇÕES NAS CLASSES COM STEREOTYPES

A base de todo funcionamento do CDI consiste no conceito de meta programação, que neste caso foram várias anotações que colocamos para que o CDI consiga funcionar.

Muitas vezes acabamos com várias anotações, e estas anotações se repetem várias vezes em nosso código, muitas vezes de maneira idêntica, e acabam poluindo nosso modelo. Ou ainda pior, as anotações não possuem significado algum para o design das classes e geram um acoplamento maior com o framework.

Os Stereotypes do CDI, permitem que os desenvolvedores identifiquem estas "regras" e isolem estas meta informações em um lugar comum, em anotações próprias de negócio.

Em nosso código existem vários pontos onde as anotações se repetem, por exemplo, na classe `LoginBean` colocamos duas anotações:

```
@Named @RequestScoped
public class LoginBean {
```

Muitos managed beans de JSF seguem esse comportamento, de terem o `@Named` e serem de escopo de request. Tanto que o CDI já possui uma anotação que simplifica esse design, a `@Model`, que substitui ambas:

```
@Model
public class LoginBean {
```

```
}
```

Essa anotação é chamada de **estereótipo**. Ela apenas indica as outras anotações que devem ser consideradas no lugar. Sua implementação é bastante simples:

```
@Named  
@RequestScoped  
@Stereotype  
@Target({ElementType.TYPE, ElementType.METHOD})  
@Retention(RetentionPolicy.RUNTIME)  
public @interface Model {}
```

A novidade aqui é a anotação `@Stereotype` do CDI que indica que ela é apenas um estereótipo. Mas o melhor é que podemos criar nossas próprias anotações de estereótipos. Poderíamos criar um estereótipo para usarmos em nossos managed beans de escopo de View:

```
@Named  
@ViewScoped  
@Stereotype  
@Retention(RetentionPolicy.RUNTIME)  
@Target(ElementType.TYPE)  
public @interface ViewModel {}
```

Agora podemos simplificar nossos managed beans de produtos e notas fiscais usando esse estereótipo:

```
@ViewModel  
public class NotaFiscalBean implements Serializable {  
    // ...  
}
```

Os estereótipos vão ficando cada vez mais interessantes conforme o número de anotações cresce, em particular quando trabalhamos com interceptadores de classe. Imagine que quiséssemos transformar todos os métodos de certos beans em transacionais usando o interceptador que fizemos antes e a anotação `@Transactional`. Poderíamos definir um novo estereótipo:

```
@Stereotype  
@RequestScoped  
@Named  
@Transactional  
@Retention(RetentionPolicy.RUNTIME)  
@Target(ElementType.TYPE)  
public @interface TransactionModel {}
```

```
}
```

Agora podemos substituir as anotações na classe `LoginBean` pela anotação `@TransactionModel`.

```
@TransactionModel  
public class LoginBean {  
  
    //métodos e atributos  
  
}
```

Editora Casa do Código com livros de uma forma diferente



Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.

Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.](#)

11.11 EXERCÍCIOS: STEREOTYPES

1. Crie um novo estereótipo **@ViewModel** para os managed beans do JSF em view scope. Esse estereótipo deve incluir as anotações **@Named** e **@ViewScoped**.

Use-a na classe **NotaFiscalBean**.

2. Use o estereótipo **@Model** nos managed beans que usam **@RequestScoped** e **@Named**. Repare que trocaremos duas anotações por uma.

11.12 CONVERSATION SCOPE

Imagine que nossa **NotaFiscal** necessite de muitos dados além de CNPJ e data. Seriam dezenas de campos diferentes. A usabilidade de deixar todo o cadastro em uma única tela como temos hoje seria péssima. Uma solução comum é quebrar o grande cadastro em diversas telas formando um **wizard**. Os dados seriam coletados em etapas e apenas inseridos no banco de dados ao final da última etapa.

Como implementar essa funcionalidade?

A Web é naturalmente stateless e, se nada for feito, os dados se perderiam entre os vários requests. Normalmente, a solução é trabalhar com a **sessão** do usuário, armazenando os dados intermediários para que possam ser recuperados ao final e serem inseridos no banco.

Seria simples fazer isso em JSF: bastaria um managed bean anotado com **@SessionScoped**. Mas usar o escopo de sessão para essa atividade tem inúmeros problemas. O primeiro e mais evidente é que a sessão é "global" para o usuário. Se ele abrir várias abas no navegador, estará na mesma sessão. Isolar as diferentes abas para que os dados de uma não interfiram na outra é possível mas muito trabalhoso de fazer manualmente.

Outro problema de usar o escopo de sessão para isso é que ele é um escopo muito grande e custoso. Se não o limparmos explicitamente, os dados ficarão em memória durante todo o tempo de vida da sessão do usuário, o que pode ser bastante tempo.

E tudo isso quando, na verdade, para o nosso wizard, bastaria guardar os dados durante uns poucos requests enquanto o usuário interage com nosso cadastro e logo descartá-los quando o wizard fosse completado.

Um dos grandes recursos do CDI, inspirado no antigo Seam 2, é o chamado **escopo de conversação**. É um escopo intermediário entre *request* e *session*, indicado quando precisamos que os dados sejam mantidos durante um processo do usuário que leve vários passos mas tenha um fim previsível.

O JSF 2 trouxe também um escopo intermediário entre *request* e *session*, o *view scope* que já vimos. Ele é usado quando queremos guardar estado das interações do usuário na mesma tela. O *view scope* não nos ajudaria a implementar nosso wizard de várias telas.

O *conversation scope* permite interações maiores que o *view*. O usuário poderá navegar entre telas, seguir um wizard, executar vários passos dentro da mesma conversação. Usando CDI podemos usar todos esses escopos simultaneamente.

Para usarmos o escopo de conversação, podemos anotar nosso bean com `@ConversationScoped`. Mas, diferentemente dos outros escopos que vimos (*request*, *view*, *session*, *application*), o *conversation scope* não tem as etapas de criação e finalização do escopo bem definidos. Um *request* acaba quando a resposta é enviada ao cliente; a *view* acaba quando o usuário navega para outra tela; a sessão acaba quando é invalidada; e a aplicação acaba quando o contexto é desligado. E uma conversação, quando acaba?

Usar apenas a anotação `@ConversationScoped` não nos dará automaticamente o recurso de conversação entre múltiplos requests, algo chamado de *long running conversation*. Para isso, precisamos **demarcar** o início e o fim da conversação. Isso é feito, usando o objeto `conversation` que possui métodos `begin()` e `end()` que devemos chamar explicitamente em nosso managed bean nos momentos adequados.

Wizard para notas fiscais

Vamos quebrar nosso cadastro de notas em duas telas. O arquivo `notafiscal.xhtml` ficará apenas com os dados de uma `NotaFiscal`:

```
<h:form>
    <h:outputLabel value="CNPJ Cliente:" />
    <h:inputText value="#{notaFiscalBean.notaFiscal.cnpj}" />

    <h:outputLabel value="Data:" />
    <h:inputText value="#{notaFiscalBean.notaFiscal.data.time}" />

    <h:commandButton value="Avançar"
```

```

        action="#{notaFiscalBean.avancar}"  />
</h:form>
```

A única diferença é que o botão nessa tela não mais insere no banco de dados mas avança para o próximo passo do wizard. Criaremos o método `avancar` no managed bean que deve fazer duas coisas: iniciar uma nova conversação e redirecionar para a próxima tela, que chamaremos de `item.xhtml`. Para iniciar a conversação, precisamos chamar o método `begin()` da `Conversation`, que podemos obter via injecção de dependências. Além disso, nosso bean deverá ser anotado com `@ConversationScoped`:

```

@Named @ConversationScoped
public class NotaFiscalBean {

    @Inject
    private Conversation conversation;

    // ... outros atributos

    public String avancar() {
        conversation.begin();
        return "item?faces-redirect=true";
    }

    // ... outros métodos
}
```

A tela de cadastro de itens terá o restante da tela, com os dados dos itens, a funcionalidade de adicioná-los à nota atual e a tabela que exibe os itens já adicionados. Para enxergarmos que estamos na mesma conversação que a tela anterior, vamos exibir no topo dessa tela o CNPJ que foi coletado na primeira tela:

```

<h:form>
    <h:outputText
        value="CNPJ Cliente: #{notaFiscalBean.notaFiscal.cnpj}" />

    <fieldset>
        <legend>Novo Item</legend>
        <!-- Campos do item: Produto e Quantidade -->
        <h:commandButton action="#{notaFiscalBean.guardaItem}"
            value="Guardar Item" />
    </fieldset>

    <h2>Itens adicionados na nota fiscal</h2>
    <!-- DataTable com os itens atuais -->

    <h:commandButton value="Finalizar"
        action="#{notaFiscalBean.gravar}" />
</h:form>
```

O código anterior é um pedaço do exato mesmo código que estamos usando até agora, o que fizemos foi apenas quebrar o formulário em duas páginas para aprendermos sobre conversações.

O método `gravar` do managed bean é quase igual ao que tínhamos antes. Temos apenas uma novidade: após inserir os dados no banco de dados, finalizamos a conversação com `conversation.end()`:

```

public String gravar() {
    dao.adiciona(notaFiscal);
    conversation.end();
    return "notafiscal?faces-redirect=true";
}

```

O último toque é redirecionar de volta para a tela de notas fiscais para que o usuário possa cadastrar mais uma nota.

Voltando a conversação

E se quando o usuário estiver inserindo os novos itens na nota atual ele lembrar que precisa mexer em algum dado da primeira tela? Como voltar a navegação sem perder os dados, tanto da nota quanto possíveis itens já adicionados?

Um botão de voltar na tela `item.xhtml` seria bastante simples:

```
<h:commandButton value="Voltar" action="notafiscal?faces-redirect=true"/>
```

Ele apenas causa uma navegação de volta para a primeira tela. Mas como estamos com a conversação ativa, os dados serão mantidos e conseguiremos editar sem problemas os dados da nota.

Há apenas uma questão a ser resolvida: após editar as informações na nota, o usuário clicará novamente em "Avançar" para prosseguir novamente o wizard. Mas nosso método `avancar` chama `conversation.begin()`, o que nos dará um erro nesse cenário já que a conversação já foi criada e ainda está ativa.

A solução é fazer o método `avancar` verificar se já não existe uma conversação ativa antes de chamar o `begin()`:

```

public String avancar() {
    if (conversation.isTransient()) {
        conversation.begin();
    }
    return "item?faces-redirect=true";
}

```

O PARÂMETRO CID

Repare que, quando fazemos a navegação entre as telas, é feito um request GET para a URL nova de cada respectiva tela. E junto com a URL, é passado um parâmetro *cid*, o **conversation id**. Esse parâmetro é fundamental para que o JSF saiba em qual conversação estávamos no momento e permitir ainda o uso de conversações diferentes em abas diferentes.

É possível acessar o ID da conversação programaticamente pelo método `getId()` de `Conversation`.

11.13 EXERCÍCIOS OPCIONAL: ESCOPO DE CONVERSAÇÃO

1. Implemente o wizard para cadastro de notas fiscais. Quebre nossa tela atual em duas partes, deixando os campos de `NotaFiscal` em `notafiscal.xhtml` e os campos de Item e seu data table em `item.xhtml`.

Transforme o `NotaFiscalBean` em `@ConversationScoped` e use o objeto `Conversation` para delimitar o início e término da conversação.

Já conhece os cursos online Alura?



A Alura oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Programação, Front-end, Mobile, Design & UX, Infra e Business, com um plano que dá acesso a todos os cursos. Ex aluno da Caelum tem 15% de desconto neste link!

[Conheça os cursos online Alura.](#)

11.14 NOVIDADE DO JAVA EE 7: FACES FLOW

Muitas vezes um cadastro ou processo na aplicação web se estende por várias páginas. Um exemplo clássico é a compra de ticket de viagem no qual seguimos várias páginas para escolher origem, destino e forma de pagamento. Esse processo ou *workflow* tem um ponto de entrada e saída bem definida e não queremos que um usuário entre no meio do processo.

Para ajudar a representar o fluxo de páginas foram criadas extensões do JSF, como por exemplo Spring Web Flow ou ADF Task Flow. Com a atualização do JSF para 2.2 entraram na especificação as melhores ideias desses frameworks. O novo **Faces Flow** introduz uma forma simples de organizar um fluxo e associar um *Managed Bean* a este fluxo.

Para facilitar foi introduzida uma nova anotação **FlowScoped**. Ela define o modelo que suportará o fluxo. Repare que isso é bem parecido com o escopo de conversão, mas sem a necessidade de inicializar e terminar o escopo programaticamente. As páginas que fazem parte desse *flow* podem aproveitar o `NotaFlowBean` para guardar valores:

```
@Named  
@FlowScoped("cadastroNota")  
public class NotaFlowBean {
```

11.15 PARA SABER MAIS: INTERCEPTADOR DE AUDITORIA

O que podemos fazer se o proprietário da UberDist quiser saber qual funcionário que cadastrou determinado produto? E se precisarmos adicionar em mais métodos a mesma funcionalidade de auditoria que adicionamos no método `adiciona`?

Podemos utilizar a técnica no Ctrl+C e Ctrl+V, porém sabemos que esta não é a melhor maneira de resolver os problemas. Estamos numa situação onde precisamos de algo que seja executado antes dos métodos, afinal é esta ação que estamos tomando em todos os métodos que desejamos colocar o log de auditoria.

Como vimos anteriormente o CDI provê uma maneira muito simples e elegante de resolver este problema, onde conseguimos executar algo antes e/ou depois de algum método. Os conhecidos Interceptors.

Sendo assim, vamos criar a classe do interceptor e a anotação personalizada para ser utilizada nas classes que desejamos interceptar.

```
@InterceptorBinding
@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface Auditavel {

}

@Interceptor @Auditavel
public class AuditoriaInterceptor {

    @AroundInvoke
    public Object audita(InvocationContext context) throws Exception {
        return null;
    }
}
```

Dentro do método `audita` podemos, por exemplo, gravar as mensagens de quem está executando determinado método e guardá-las em um arquivo de log utilizando o log4j.

Porém precisamos saber qual usuário está logado na aplicação antes de gravarmos as mensagens no arquivo. Resolvemos facilmente este problema utilizando CDI, que injeta a dependência que precisamos, neste caso, o objeto do tipo `UsuarioLogado` que está na sessão do usuário:

```
@Interceptor @Auditavel
public class AuditoriaInterceptor {

    private @Inject UsuarioLogado usuarioLogado;

    private static final Logger logger =
        Logger.getLogger(AuditoriaInterceptor.class);

    @AroundInvoke
    public Object aroundInvoke(InvocationContext ic) throws Exception {
        Object result = ic.proceed();
        logger.info("O método: " + ic.getMethod().getName())
    }
}
```

```

        + " foi executado pelo usuario: "
        + usuarioLogado.getUsuario().getLogin()
        + " na data: " + new SimpleDateFormat("dd/MM/yyyy 'às' hh:mm:ss"));

    return result;
}

}

```

Agora vamos anotar a classe `ProdutoBean` para que ela seja interceptada. Porém precisamos dizer que a classe é gerenciada pelo CDI. Para isto, basta anotarmos a classe com `@RequestScoped` e com `@Named` para que seja acessível por EL nas páginas xhtml.

Além destas configurações, anotaremos o método `grava` da classe `ProdutoBean` com a anotação `@Auditavel`.

11.16 INJECTION POINTS

Já migramos a aplicação para usar injeção de dependências. A ideia básica é transformar tudo em beans do CDI e injetar nos pontos certos com `@Inject`, evitando dar `new` nos componentes.

Fizemos isso com o `UsuarioDao`, `ProdutoDao` e `NotaFiscalDao`. No entanto, há alguns métodos comuns entre os DAOs os quais poderíamos extrair para um classe `Dao`. Essa classe agrupa os métodos mais genéricos, também chamados os métodos CRUDs. O *Dao genérico* será parametrizado parecido com as classes do `java.util` (por exemplo `List<Produto>`). Ou seja, ao utilizar a classe definimos o nome da entidade que `Dao` atende, por exemplo:

```

Dao<Produto> daoProduto = new Dao<Produto>(Produto.class);

Dao<NotaFiscal> daoNotaFiscal = new Dao<NotaFiscal>(NotaFiscal.class);

```

Repare também que passamos a classe da entidade no construtor, útil para alguns métodos de pesquisa com JPA. A implementação da classe é nada complicada. O segredo é definir um parâmetro na declaração da classe, um tipo `T`, que representa a entidade, no exemplo acima, a entidade é `Produto` ou `NotaFiscal` (aquilo que está dentro `<>`):

```

public class Dao<T> {

    private final Class<T> classe;

    @Inject
    private EntityManager manager;

    public Dao(Class<T> classe) {
        this.classe = classe;
    }

    public void adiciona(T entidade) {
        manager.persist(entidade);
    }

    //outros método CRUD
}

```

```
}
```

Dentro do Dao já fazemos o uso do CDI, estamos injetando o `EntityManager`. No entanto, fazer o mesmo com o Dao genérico não é tão simples. Por ser genérico, ele precisa ter acesso ao objeto `Class` que estamos querendo usar no momento. Vimos que podemos usar *método produtores* quando queremos construir algum objeto mais complexo. Vamos usar esse recurso para produzir o Dao:

```
public class DaoFactory {  
  
    @Produces  
    public <T> Dao<T> createDao() {  
        return new Dao(???); // Dao de qual entidade?  
    }  
}
```

Repare que retiramos as informações de generics dessa classe, já que ela deve ser capaz de produzir todo tipo de Dao. Mas ainda precisamos passar a classe como argumento para ele. Se quisermos um `Dao<Produto>`, devemos passar o `Produto.class` para o `Dao`. Mas essa classe depende do *ponto de injeção* do Dao:

```
public class ProdutoBean {  
  
    @Inject  
    private Dao<Produto> dao;  
}
```

O CDI nos permite acessar informações detalhadas sobre o ponto de injeção através da classe `InjectionPoint`, que podemos receber direto na fábrica:

```
public class DaoFactory {  
  
    @Produces  
    public <T> Dao<T> createDao(InjectionPoint injectionPoint) {  
        Class classe = // ... recupera a classe  
        return new Dao(classe);  
    }  
}
```

Através do `InjectionPoint` conseguimos acessar, via `Reflection`, o a classe do tipo genérico passado no ponto de injeção. O código é um pouco estranho mas é relativamente curto para resolver o problema que temos:

```
public class DaoFactory {  
    @Produces  
    public <T> Dao<T> createDao(InjectionPoint injectionPoint) {  
        ParameterizedType type = (ParameterizedType) injectionPoint.getType();  
        Class classe = (Class) type.getActualTypeArguments()[0];  
        return new Dao(classe);  
    }  
}
```

Saber inglês é muito importante em TI

galandra O Galandra auxilia a prática de inglês através de *flash cards* e *spaced repetition learning*. Conheça e aproveite os preços especiais.

[Pratique seu inglês no Galandra.](#)

11.17 EXERCÍCIOS OPCIONAIS: INJECTIONPOINT

1. Crie a classe `DaoFactory` que produz o Dao genérico baseado no tipo do ponto de injeção:

```
public class DaoFactory {  
    @Produces  
    public <T> Dao<T> createDao(InjectionPoint injectionPoint) {  
        ParameterizedType type = (ParameterizedType) injectionPoint.getType();  
        Class classe = (Class) type.getActualTypeArguments()[0];  
        return new Dao(classe);  
    }  
}
```

Como vamos produzir o `Dao`, ele precisa implementar `Serializable`.

2. Injete um `Dao<Produto>` na classe `ProdutoBean` e delega nos métodos CRUD ao *Dao genérico*.

Teste novamente o cadastro de produtos e tudo deve estar funcionando.

INTERFACES WEB 2.0 COM COMPONENTES RICOS

"A grandeza não consiste em receber honras, mas em merecê-las." -- Aristóteles

Ao término desse capítulo, você será capaz de:

- Utilizar componentes ricos utilizando componentes externos

12.1 BIBLIOTECAS DE COMPONENTES

Atualmente, as aplicações que trabalham na plataforma Web cada vez mais precisam ser amigáveis e de fácil utilização por parte dos usuários. Para atender estas necessidades, é muito comum precisarmos de componentes que facilitam esse processo. Criar componentes JSF ricos, que possuem uma excelente interação, não é uma tarefa trivial.

Além disso, é muito comum quando precisamos de um determinado componente, que ele já tenha sido criado de maneira semelhante por algum desenvolvedor ou empresa. Isto nos faz lembrarmos daquela premissa de que não devemos "reinventar a roda", ou seja, vamos utilizar esses componentes prontos ao invés de refazê-los.

Existem diversas bibliotecas de componentes criadas para o JSF, das quais se destacam o **JBoss Richfaces** e o **PrimeFaces**.

ENCONTRANDO O JBOSS RICHFACES E O PRIMEFACES

O JBoss Richfaces pode ser encontrado em <http://www.jboss.org/richfaces> e o PrimeFaces pode ser encontrado em <http://www.primefaces.org/>

Seus livros de tecnologia parecem do século passado?



Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**. Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

[Casa do Código, Livros de Tecnologia.](#)

12.2 MELHORANDO A TELA DE ADIÇÃO DE NOTAS FISCAIS

A tela de adição de notas fiscais que fizemos no capítulo anterior está funcionando e resolve os problemas da UberDist. Porém, sabemos que a tela não está tão amigável quanto poderíamos deixá-la. Existem algumas deficiências nela.

Por exemplo, no campo da data da nota fiscal tivemos que contornar o problema do time zone e ajustá-lo. Imagine ter que ajustar o time zone em todos os lugares que precisarmos de um campo de data: seria totalmente inviável, uma vez que, se precisássemos alterar o time zone, teríamos que alterar em todos estes lugares. Outro ponto ruim no atual campo de data é a possibilidade do usuário digitar datas inválidas, o que ocasionaria um erro e uma inconsistência em nosso sistema.

Uma prática comum para evitar estes dois problemas citados, seria utilizar um componente de data que exibe um calendário e permite ao usuário selecioná-la de maneira simples e direta. Componentes como esses são conhecidos como *date pickers*, e são, na maioria das vezes, implementados utilizando JavaScript e CSS.



Além do problema da data, poderíamos melhorar a naveabilidade do sistema se colocássemos o cadastro de produtos e o cadastro de notas fiscais na mesma página, porém utilizando um conceito de abas, tradicionalmente utilizada em aplicações Desktop.

The screenshot shows a web application interface. At the top, there's a blue header bar with the word 'UBERDIST' in large, bold, white letters. Below the header is a navigation bar with two tabs: 'Produtos' (selected) and 'Notas Fiscais'. The main content area has a title 'Novo Produto' (New Product). Underneath it, there's a section titled 'Dados do Produto' (Product Data) with three input fields: 'Nome' (Name), 'Descrição' (Description), and 'Preço' (Price), each with a corresponding text input box. Below these fields is a 'Gravar' (Save) button. To the right of this form is a table titled 'Listagem de Produtos' (Product List) with columns 'Nome' (Name), 'Descrição' (Description), and 'Preço' (Price). A single row is shown with the data: 'Refrigerante' (Refrigerant), an empty description field, and '3.0' in the price field. To the right of this row are two buttons: 'Remover' (Remove) and 'Alterar' (Edit). At the bottom of the page, there's a small copyright notice: 'Copyright 2010. Todos os Direitos reservados a --Seu nome aqui--.'

12.3 PRIMEFACES

Durante o curso, usaremos o **PrimeFaces**, uma das bibliotecas de componentes mais famosas para JSF2. Na época do JSF 1.x, o Richfaces era a biblioteca mais famosa. Mas sua demora (quase 2 anos) para lançar a versão compatível com JSF2 fez com que o Primefaces dominasse o mercado das aplicações em JSF2. Além disso, o Primefaces possui centenas de componentes ricos, muitos mais que o Richfaces.

Mas o importante é aprender os conceitos e a integração com alguma biblioteca. Usaremos tags e JARs específicos, mas trocar para outra biblioteca é tão simples quanto trocar os JARs e usar as novas tags.

O PrimeFaces também é uma biblioteca que conta com mais de 100 componentes ricos e que podem ser utilizados gratuitamente. Colocá-lo em nossa aplicação é muito simples, basta baixar JAR e copiá-lo para dentro da pasta *WEB-INF/lib* da sua aplicação.

Configuração antes das Servlets 3

A única configuração necessária é declarar uma `servlet` do PrimeFaces responsável por enviar os recursos necessários para o funcionamento para o cliente. Em servidores que suportam Servlet 3.0/Java EE 6, como o Glassfish 3/4 ou o Tomcat 7, esse procedimento não é necessário e pode ser ignorado.

```
<servlet>
    <servlet-name>Resource Servlet</servlet-name>
    <servlet-class>
        org.primefaces.resource.ResourceServlet
    </servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>Resource Servlet</servlet-name>
    <url-pattern>/primefaces_resource/*</url-pattern>
</servlet-mapping>
```

12.4 ADICIONANDO DATEPICKER

O próximo passo é adicionar o namespace do PrimeFaces na tag `<html>` : `xmlns:p="http://primefaces.org/ui"`.

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:p="http://primefaces.org/ui">
```

Isso já é suficiente! Estamos prontos para utilizar os componentes prontos e nos livrar de alguns problemas discutidos anteriormente. Na tela de adição de notas fiscais, podemos substituir o `<h:inputText>` pela tag `<p:calendar>` do prime faces:

```
<p:calendar value="#{notaFiscalBean.notaFiscal.data.time}"
             pattern="dd/MM/yyyy"/>
```

O campo referente a data da nota fiscal teria a seguinte aparência:

The screenshot displays a web-based application interface. At the top, a blue header bar features the text "UBERDIST". Below the header, there are two main sections: "Dados da nota" and "Dados do produto". The "Dados da nota" section contains fields for "CNPJ Cliente" (with an empty input field) and "Data" (containing the value "25/05/2011"). The "Dados do produto" section contains fields for "Produto" (with a dropdown menu showing "Produto #1") and "Quantidade" (with an empty input field). Below these sections is a button labeled "Adicionar Item". Underneath these sections is a heading "Itens da nota fiscal" followed by a table with four columns: "Produto", "Preco", "Quantidade", and "Valor". A "Gravar nota fiscal" button is located at the bottom left of the main form area. At the very bottom of the page, a blue footer bar contains the text "Copyright 2010. Todos os Direitos reservados a ---Seu nome aqui---".

Agora é a melhor hora de respirar mais tecnologia!



Se você está gostando dessa apostila, certamente vai aproveitar os **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum. Programação, Mobile, Design, Infra, Front-End e Business! Ex-aluno da Caelum tem 15% de desconto, siga o link!

[Conheça a Alura Cursos Online.](#)

12.5 EXERCÍCIOS: COMPONENTE DE CALENDÁRIO

1. Precisamos habilitar o Primefaces. Para isso, primeiramente vamos pegar o JAR e adicionar à nossa aplicação.

Entre no diretório `Caelum/26/` e copie os jars das pastas `jars-primefaces` e `jars-commons` para a pasta `WebContent/WEB-INF/lib` do seu projeto. Como estamos no Glassfish 4, não é necessário mais nenhuma configuração.

2. Vamos **alterar** o arquivo `notafiscal.xhtml` para que agora ele utilize um componente de data.

- **Adicione** na tag de cabeçalho o import das tags de componentes do PrimeFaces:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:p="http://primefaces.org/ui">
```

- **Remova** o componente `h:inputText` referente a **data** da nota fiscal.

- Em seu lugar, use o componente `<p:calendar>` :

```
<p:calendar pattern="dd/MM/yyyy"
             value="#{notaFiscalBean.notaFiscal.data.time}" />
```

3. Acesse: <http://localhost:8080/fj26-notas-fiscais/notafiscal.xhtml>

A página de notas fiscais ficará com a seguinte aparência:

UBERDIST

Dados da nota

CNPJ Cliente:

Data:

25/05/2011

Dados do produto

Produto:

Produto #1

Quantidade:

Adicionar Item

Itens da nota fiscal

Produto	Preço	Quantidade	Valor
---------	-------	------------	-------

Gravar nota fiscal

Copyright 2010. Todos os Direitos reservados a ---Seu nome aqui---.

12.6 MÁSCARA COM P:INPUTMASK

O PrimeFaces possui diversos componentes que podemos utilizar em nossa aplicação. Por exemplo, podemos fazer com que o campo CNPJ só possa ser informado no formato adequado. Uma das formas de conseguirmos isso é através de caixas de texto que contenham uma máscara pré-definida para formatar o valor de forma adequada.

No PrimeFaces, existe o componente `inputMask` onde podemos informar um atributo chamado `mask` com a máscara adequada. Nesse caso, para usarmos para a entrada de um CNPJ, teríamos:

```
<p:inputMask value="#{managedBean.cnpj}" mask="99.999.999/9999-99" />
```

12.7 EXERCÍCIOS: MÁSCARA NO CAMPO CNPJ

1. Use o componente `<p:inputMask>` na tela de notas fiscais para receber o CNPJ com a máscara adequada. Substitua o antigo campo `<h:inputText>`.

```
<p:inputMask value="#{notaFiscalBean.notaFiscal.cnpj}"  
mask="99.999.999/9999-99" />
```

Editora Casa do Código com livros de uma forma diferente



Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.

Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.](#)

12.8 REALIZANDO PAGINAÇÃO DE DADOS

Para fazermos a listagem das informações, até o momento utilizamos o componente `<h:dataTable>` que recebe uma lista para determinar as informações que serão exibidas. No entanto, quando temos muitas informações para listar, essa tabela começará a crescer e atrapalhará a usabilidade da aplicação. Para resolver esse problema, é muito comum realizarmos a paginação dos dados.

Na paginação, os itens são separados em páginas onde os dados são exibidos em um intervalo pré-definido, por exemplo cada conjunto de dez linhas ficam em uma página e com isso é possível navegar para as páginas seguintes ou inclusive voltar atrás.

Mas, como podemos fazer tudo isso? Precisamos de algum jeito controlar qual é a página que está sendo visualizada no momento, e além disso, fazermos pequenos cálculos para determinar qual intervalo de dados deverá ser exibido a cada página. Essa não é uma tarefa trivial.

Justamente para resolver esse problema, as bibliotecas de componente geralmente possuem seus próprios componentes para tabulação de dados, que no caso do PrimeFaces também é chamado de `dataTable`, onde podemos indicar que desejamos fazer a paginação, qual a quantidade de itens que deve ser exibido por página e assim por diante.

Com isso, poderíamos fazer uma nova tela para exibir todas as notas fiscais da nossa aplicação e paginá-las com a Tag do PrimeFaces.

A Tag `dataTable` do PrimeFaces recebe um atributo chamado `paginator` onde indicamos se queremos ou não que a paginação seja realizada. Além disso, precisamos indicar a quantidade de linhas que cada página terá com o atributo `rows`:

```
<p:DataTable value="#{listaNotasFiscaisBean.notasFiscais}"  
var="notaFiscal" paginator="true" rows="5">
```

Dentro da `<p:dataTable>` é obrigatório o uso da `<p:column>` do PrimeFaces. Ela possui inclusive algumas facilidades a mais, como um atributo para passar o texto do header mais facilmente:

```
<p:column headerText="CNPJ">
    #{notaFiscal.cnpj}
</p:column>

<p:column headerText="DATA">
    <h:outputText value="#{notaFiscal.data.time}">
        <f:convertDateTime pattern="dd/MM/yyyy"
            timeZone="America/Sao_Paulo" />
    </h:outputText>
</p:column>
```

Listagem de Notas Fiscais

CNPJ	Data
123	08/06/2011
	08/06/2011
teste	08/06/2011

12.9 EXERCÍCIO: REALIZANDO A PAGINAÇÃO DOS DADOS

1. Crie uma nova tela para a listagem das notas fiscais em um novo arquivo `listanotas.xhtml`. Faça com que a lista seja exibida de forma paginada. Para isso, utilize a Tag `dataTable` do PrimeFaces.

Para fazer a listagem, crie um ManagedBean novo chamado `ListaNotasFiscaisBean` e faça toda a lógica dentro desse Bean. Lembre-se de utilizar o DAO para fazer a consulta ao banco de dados. O bean deve estar no `ViewScoped`. **Use as anotações do CDI.**

12.10 PAGINAÇÃO CUSTOMIZADA COM BANCO DE DADOS

Agora que temos a listagem das notas paginadas, precisamos resolver um grave problema que deixamos em nossa aplicação. Apesar de a exibição dos dados ser paginada, ou seja, é feita em pequenas partes, a consulta ao banco de dados traz **todos** os dados que estão na tabela de notas fiscais. Isso faz com que carreguemos diversos objetos de forma desnecessária para a memória.

O ideal nesses casos é que além de realizarmos a paginação visual, possamos também realizar a paginação no banco de dados, buscando só os dados que deverão ser exibidos, com isso, diminuindo a quantidade de objetos e são carregados para a memória. Todos os bancos de dados atualmente possuem funções que suportam buscar apenas uma faixa de dados, que no caso do MySQL é através da função `limit`.

O PrimeFaces permite que nós o estendamos de forma bastante fácil através de alguns pontos de extensão. Para isso, o componente `dataTable` permite que seja trabalhado além de listas, um objeto do tipo `LazyDataModel`. A ideia é que esse objeto deverá ser estendido para realizar a lógica de buscar os

dados no banco de dados a cada troca de página. Para isso, precisamos estendê-lo e implementar o seu método `fetchLazyData` que recebe como parâmetros o índice do primeiro registro para a página requisitada e a quantidade de registros que deve ser buscada no banco de dados.

A classe `LazyDataModel` deve ser tipada com o tipo da informação que vamos devolver, portanto, nesse caso adicionaremos `<NotaFiscal>` na declaração da herança da classe. Com isso, teremos um código similar ao abaixo:

```
public class DataModelNotasFiscais extends LazyDataModel<NotaFiscal> {

    public List<NotaFiscal> load (int inicio, int quantidade,
        String campoOrdenacao, SortOrder sentidoOrdenacao,
        Map<String, Object> filtros) {

        // vamos implementar esse método para realizar a busca com paginação
    }

}
```

O `dataTable` que anteriormente estava referenciando uma lista de notas fiscais, agora deverá referenciar uma instância de `DataModelNotasFiscais`. Vamos receber essa instância através do CDI:

```
public class ListaNotasFiscaisBean {

    @Inject
    private DataModelNotasFiscais dataModel;

    public LazyDataModel<NotaFiscal> getDataModel() {
        return dataModel;
    }
}
```

Com isso, precisamos alterar a Tag `dataTable` para referenciar a esse `dataModel`:

```
<p:dataTable value="#{listaNotasFiscaisBean.dataModel}"
    var="notaFiscal" paginator="true" rows="5">
```

No entanto, agora precisamos dizer que o `dataTable` fará a pesquisa de forma *Lazy*, ou seja, carregará um pouco de informação por vez. Para isso, basta adicionarmos o atributo `lazy` com o valor `true`:

```
<p:dataTable lazy="true" value="#{listaNotasFiscaisBean.dataModel}"
    var="notaFiscal" paginator="true" rows="5">
```

Quando uma nova página é solicitado, ou seja, o método `load` agora será invocado, e receberá como argumentos a quantidade de registros que devem ser buscados, e o índice inicial para realizarmos essa busca. Podemos, nessa implementação, usar a classe `NotaFiscalDao` (*injetado*) que faça essa busca com a paginação e devolve uma lista de notas fiscais devidamente paginada:

```
@Inject
private NotaFiscalDao dao;

public List<NotaFiscal> load(int inicio, int quantidade,
    String campoOrdenacao, boolean sentidoOrdenacao, Map<String, Object> filtros) {
```

```
        return dao.listaTodosPaginada(inicio, quantidade);
    }
```

Aparentemente, terminamos. No entanto, ao executar a página para ver o resultado, nada aparece na tabela. É como se não existisse dado algum. Isso acontece pois o `DataModelNotasFiscais` precisa saber a quantidade de registros que existem no banco, para calcular a quantidade de páginas. Isso pode ser conseguido com uma consulta no banco que realize um `count`. Esse valor, deve ser passado para o método de `LazyDataModel` chamado `setRowCount`. Assim, ele poderá calcular a quantidade de páginas necessárias.

Agora, no momento da instanciação, devemos passar como atributo essa quantidade. Vamos usar um *callback* de criação para automaticamente inicializar a quantidade de registros. Um *callback* é nada mais do que um método que é chamado pelo CDI quando acontece algum evento na ciclo da vida do objeto. Dessa maneira usaremos a anotação `@PostConstruct` onde teríamos que fazer algo como:

```
public class DataModelNotasFiscais extends LazyDataModel<NotaFiscal>{

    @Inject
    private NotaFiscalDao dao;

    @PostConstruct
    void inicializaTotal() {
        this.setRowCount(dao.contaTodos());
    }

    @Override
    public List<NotaFiscal> load(int first, int pageSize,
        String sortField, SortOrder sortOrder,
        Map<String, Object> filters) {

        return dao.listaTodosPaginada(first, pageSize);
    }
}
```

O método `inicializaTotal` é chamado após criação do objeto e injeção das dependências. Com isso, nossa paginação agora é realizada tanto na tela quanto no banco de dados. Sua grande vantagem com relação ao que tínhamos antes é a melhor utilização da memória, sem carregar dados desnecessários, o que pode ser um problema em sistemas que precisam fazer listagem de muitos dados.

Já conhece os cursos online Alura?



A Alura oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Programação, Front-end, Mobile, Design & UX, Infra e Business, com um plano que dá acesso a todos os cursos. Ex aluno da Caelum tem 15% de desconto neste link!

[Conheça os cursos online Alura.](#)

12.11 EXERCÍCIOS: ESTENDENDO COMPONENTES PARA REALIZAR PAGINAÇÃO NO BANCO DE DADOS

- Precisamos fazer com que a paginação dos dados seja realizada também no banco de dados, ou seja, sempre que o usuário passar para outra página na paginação uma nova busca deve ser realizada no banco apenas retornando os dados que serão exibidos. Para isso precisaremos estender a tag `dataTable`.

Crie uma nova classe chamada `DataModelNotasFiscais` no pacote `br.com.caelum.notasfiscais.datamodel` e faça com que ela estenda de `LazyDataModel<NotaFiscal>`.

Injete o `NotaFiscalDao`:

```
@Inject  
private NotaFiscalDao dao;
```

Adicione um método `posConstrucao` que inicializa o `DataModel`, preenchendo a quantidade de registros que estão no banco através do método `setRowCount`.

```
@PostConstruct  
void inicializaTotal() {  
    super.setRowCount(dao.contaTodos());  
}
```

Implemente o método `load` para buscar os dados paginados no banco de dados.

```
public List<NotaFiscal> load(int inicio, int quantidade,  
String campoOrdenacao, SortOrder sentidoOrdenacao,  
Map<String, Object> filtros) {  
  
    return dao.listaTodosPaginada(inicio, quantidade);  
}
```

2. Faça com que o Managed Bean `ListaNotasFiscaisBean` injete o novo `DataModelNotasFiscais`, ao invés de usar a simples lista de notas fiscais.

```

@Named
@ViewScoped
public class ListaNotasFiscaisBean implements Serializable {

    @Inject
    private DataModelNotasFiscais dataModel;

    public DataModelNotasFiscais getDataModel() {
        return dataModel;
    }
}

```

3. Altere na tela o `<p:dataTable>` para fazer a listagem de forma `lazy`. Não se esqueça que para buscar os novos dados uma nova requisição AJAX deve ser disparada, portanto envolva o `<p:dataTable>` em uma tag `<h:form>`. Além disso, troque seu `value` para apontar para nosso novo data model:

```

<h:form>
    <p:dataTable value="#{listaNotasFiscaisBean.dataModel}" var="notaFiscal"
        paginator="true" rows="5" lazy="true">
        <p:column headerText="CNPJ">
            #{notaFiscal.cnpj}
        </p:column>

        <p:column headerText="DATA">
            <h:outputText value="#{notaFiscal.data.time}">
                <f:convertDateTime pattern="dd/MM/yyyy"
                    timeZone="America/Sao_Paulo" />
            </h:outputText>
        </p:column>
    </p:dataTable>
</h:form>

```

12.12 MENUS E NAVEGAÇÃO

Um recurso recorrente nas bibliotecas de componentes é o uso de menus ricos. Com o PrimeFaces não é diferente. Há diversos tipos de menus disponíveis, tanto dropdown, laterais, barras de menu etc. Você pode usar submenus, navegação, ações Ajax, personalizar ícones e muito mais.

Vamos usar o `<p:menubar>` no topo da nossa div conteúdo para prover navegação entre as diversas páginas do sistema. Podemos ter inclusive um submenu para as duas páginas relacionadas à notas fiscais:

```

<h:form>
    <p:menubar>
        <p:menuitem value="Produtos" url="produto.xhtml" />
        <p:submenu label="Notas Fiscais">
            <p:menuitem value="Cadastro" url="notafiscal.xhtml"/>
            <p:menuitem value="Listagem" url="listanotas.xhtml"/>
        </p:submenu>
        <p:menuitem value="Sair" action="#{loginBean.logout}"/>
    </p:menubar>
</h:form>

```

Repare como usamos navegações simples apontando diretamente as URLs e também o disparo de métodos de `action` que fazem sua própria navegação.

12.13 EXERCÍCIOS: MENUS

1. Adicione o menu de navegação ao template do nosso sistema. Um bom lugar para posicioná-lo é **dentro** da `<div id="conteudo">`, logo **acima** do insert do corpo:

```
<div id="conteudo">
    <h:form>
        <p:menubar>
            <p:menuitem value="Produtos" url="produto.xhtml" />
            <p:submenu label="Notas Fiscais">
                <p:menuitem value="Cadastro" url="notafiscal.xhtml"/>
                <p:menuitem value="Listagem" url="listanotas.xhtml"/>
            </p:submenu>
            <p:menuitem value="Sair" action="#{loginBean.logout}"/>
        </p:menubar>
    </h:form>

    <ui:insert name="corpo" />
</div>
```

Não esqueça importar a biblioteca Primefaces no cabeçalho da página.

2. Você pode melhorar ainda mais adicionando a funcionalidade de mostrar o menu apenas quando o usuário estiver logado. Dica: use o `rendered` para isso.
3. É possível customizar o layout atribuindo ícones aos itens do menu ou a submenus. O próprio PrimeFaces vem com alguns ícones prontos para uso. Para usá-los, basta preencher o atributo `icon` do `<p:menuitem>` ou do `<p:submenu>`:

```
<p:menuitem value="Produtos" url="produto.xhtml"
            icon="ui-icon ui-icon-document"/>
```

Outras ideias de ícones disponíveis para uso: **ui-icon-pencil**, **ui-icon-close**, **ui-icon-suitcase**, **ui-icon-comment**, **ui-icon-person**, **ui-icon-home**, **ui-icon-flag**, **ui-icon-search** e dezenas de outros.

Saber inglês é muito importante em TI

galandra O Galandra auxilia a prática de inglês através de *flash cards* e *spaced repetition learning*. Conheça e aproveite os preços especiais.

[Pratique seu inglês no Galandra.](#)

12.14 ADICIONANDO ABAS

Outro ponto que discutimos e que fará nossa aplicação ter uma aparência mais simples é a colocação de abas para facilitar a naveabilidade dos usuários do sistema. Vamos utilizar o componente de abas do PrimeFaces. Para isto, basta usarmos a tag `<p:tabView>`. E, para cada aba que desejarmos adicionar, devemos colocar uma tag `<p:tab>` dentro da tag `<p:tabView>` adicionada anteriormente.

```
<p:tabView>
    <p:tab title="Aba 1">
        <h:outputText value="Conteúdo da aba 1" />
    </p:tab>
    <p:tab title="Aba 2">
        <h:outputText value="Conteúdo da aba 2" />
    </p:tab>
</p:tabView>
```

Com esse código teremos uma página com a seguinte aparência:



12.15 EXERCÍCIOS: COMPONENTE DE ABAS

1. Na tela de cadastro de notas fiscais, use abas para separar os campos de cadastro da nota em si e os campos dos itens.

Use um `<p:tabView>` para conter os dois fieldsets. Separe-os em `<p:tab>` diferentes.

12.16 EXERCÍCIOS OPCIONAIS

1. Coloque o `<p:dataTable>` também na tela de Produtos. Faça paginação como no exercício anterior.
2. Para manter uma certa consistência visual, você pode trocar os `inputText` para usar `<p:inputText>`. Use também o `<p:inputTextarea>` que, além do visual do PrimeFaces, também possui o recurso de crescer dinamicamente dependendo do tamanho do texto.

Outro componente que melhoraria apenas a parte visual é o `<p:fieldset legend="Texto">`. Cuidado que ele recebe o `legend` como atributo e não como tag.

3. Use o componente `<p:keyboard password="true"/>` para entrada de senhas. Faça o teste na página de login.
4. Na tela de produto, você pode usar um editor rico para a descrição com o `<p:editor/>`.
5. (trabalhoso) Implemente a remoção e edição de notas fiscais. Embora trabalhoso, não deve ser muito diferente do que já fizemos antes com produtos.

Seus livros de tecnologia parecem do século passado?



Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**. Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

[Casa do Código, Livros de Tecnologia.](#)

12.17 PARA SABER MAIS: CUSTOMIZANDO O VISUAL DOS COMPONENTES

O visual dos componentes do PrimeFaces é baseado no plugin de **ThemeRoller** do **jQuery**, podendo utilizar qualquer um dos temas disponíveis . Até a versão 2 do PrimeFaces, os temas eram baixados dinamicamente do site do **jQuery**, o que impedia a troca sem uma conexão com a internet.

A partir da versão 3, os temas devem ser baixados do site do PrimeFaces, em <http://primefaces.org/themes> .

Para escolher uma tema diferente do default, após copiar o jar para a pasta **WEB-INF/lib** é necessário configurá-lo no **web.xml**:

```
<context-param>
    <param-name>primefaces.THEME</param-name>
    <param-value>bluesky</param-value> <!-- aristo é default -->
</context-param>
```

Existe um componente que permite alterar o tema da aplicação dinamicamente, o **ThemeSwitcher**:

```
<p:themeSwitcher style="width:150px">
    <f:selectItem itemLabel="Escolha o Tema" itemValue="" />
    <f:selectItem itemLabel="Aristo" itemValue="aristo" />
    <f:selectItem itemLabel="Bluesky" itemValue="bluesky" />
</p:themeSwitcher>
```

Mas esse componente apenas muda o tema da página atual, ao mudar de página o tema antigo é exibido. Para customizar o tema na aplicação toda, é preciso criar um **ManagedBean** que gerencie o visual, e ligá-lo ao **<p:themeSwitcher>** :

```
<h:form>
    <p:themeSwitcher value="#{temaBean.tema}">
        <!-- Lista de temas disponíveis -->
        <f:selectItems value="#{temaBean.temas}" />
        <!-- a troca será realizada via ajax -->
        <f:ajax />
```

```

</p:themeSwitcher>
</h:form>

@Named
@SessionScoped
public class TemaBean implements Serializable {

    //Temas disponíveis
    private Map<String, String> temas;

    //Tema escolhido
    private String tema = "aristo";

    public TemaBean() {
        // cria lista de temas
        temas = new TreeMap<String, String>();

        temas.put("Aristo", "aristo");
        temas.put("Bluesky", "bluesky");
        temas.put("Cupertino", "cupertino");
        temas.put("Sam", "sam");
        temas.put("UI-Lightness", "ui-lightness");
        //...
    }
    //gets e sets...
}

```

Também temos que configurar o PrimeFaces para buscar a configuração do tema atual no TemaBean, dentro do web.xml. Assim, quando trocarmos de página, o tema selecionado será mantido:

```

<context-param>
    <param-name>primefaces.THEME</param-name>
    <param-value>#{temaBean.tema}</param-value>
</context-param>

```

12.18 EXERCÍCIOS OPCIONAIS

1. Copie os temas disponíveis na pasta Caelum/26/jars-temas/. Se desejar, baixe outros temas no site do PrimeFaces.
2. Altere o tema padrão da aplicação para um dos temas baixados.
3. (Desafio) Use o componente `<p:themeSwitcher/>` no cabeçalho para adicionar a possibilidade do usuário mudar o tema da aplicação. Crie um ManagedBean para gerenciar os temas da aplicação.

12.19 PARA SABER MAIS: GERAÇÃO DE GRÁFICOS COM PRIMEFACES

Uma outra funcionalidade interessante que é bastante facilitada através das bibliotecas de componentes é a geração de gráficos. O PrimeFaces possui uma biblioteca bastante rica nesse sentido, com a qual podemos gerar diferentes tipos de visualizações.

Para gerar os gráficos, podemos utilizar a tag `<p:chart type="pie">`, cujo intuito é gerar um gráfico de torta, também conhecido como gráfico de pizza. Essa Tag recebe um atributo chamado `model`, que é um objeto do tipo `ChartModel`, onde preenchemos os dados representando a descrição e

o valor de cada item do gráfico.

```
<p:chart type="pie" model="#{produtoBean.relatorioQuantidadePorProduto}" />
```

Nosso ManagedBean deverá devolver os dados para serem exibidos. Uma sugestão é criar apenas uma classe QuantidadePorProduto que recebe um Long quantidade e um Produto :

```
public class QuantidadePorProduto {  
  
    private final Produto produto;  
    private final Long quantidade;  
  
    public QuantidadePorProduto(Long quantidade, Produto produto) {  
        this.quantidade = quantidade;  
        this.produto = produto;  
    }  
  
    //getters  
  
}
```

Nosso DAO deverá fazer uma busca agrupada no Hibernate e devolver uma lista de QuantidadePorProduto . Além disso, nosso Managed bean deve devolver essa lista para a view:

```
public class GraficoDao{  
  
    @Inject  
    private EntityManager manager;  
  
    public List<QuantidadePorProduto> relatorioQuantidadePorProduto() {  
        return manager.createQuery(  
            "select new br.com.caelum.notasfiscais.modelo.QuantidadePorProduto(" +  
            "sum(i.quantidade), i.produto) " +  
            " from Item i group by i.produto").getResultList();  
    }  
}  
  
@Named  
@RequestScoped  
public class ProdutoBean {  
  
    @Inject  
    private GraficoDao graficoDao;  
    //...  
  
    public PieChartModel getRelatorioQuantidadePorProduto(){  
        PieChartModel model = new PieChartModel();  
        model.setTitle("Quantidade vendida por Produto");  
        model.setLegendPosition("w");  
        model.setShowDataLabels(true);  
  
        List<QuantidadePorProduto> lista = graficoDao  
            .relatorioQuantidadePorProduto();  
        for (QuantidadePorProduto qtde : lista) {  
            model.set(qtde.getProduto().getNome(), qtde.getQuantidade());  
        }  
  
        return model;  
    }  
    //...  
}
```



Agora é a melhor hora de respirar mais tecnologia!



Se você está gostando dessa apostila, certamente vai aproveitar os **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum. Programação, Mobile, Design, Infra, Front-End e Business! Ex-aluno da Caelum tem 15% de desconto, siga o link!

[Conheça a Alura Cursos Online.](#)

12.20 DESAFIO: GRÁFICOS

1. Seguindo a documentação do PrimeFaces, faça com que seja gerado gráficos para a aplicação onde possamos visualizar a quantidade vendida de cada produto. Crie uma nova aba chamada "Gráficos" onde esses resultados poderão ser visualizados.

12.21 PARA SABER MAIS: MELHORANDO A SELEÇÃO DE PRODUTOS NA TELA DE ITENS COM O AUTOCOMPLETE

Em nossa tela de cadastro de itens na nota fiscal estamos usando um menu para a seleção do produto. No caso da quantidade de produtos cadastrados no banco de dados ser muito grande a usabilidade da tela será prejudicada: O usuário terá que usar o scroll do menu que pode crescer até sair dos limites da tela.

Para resolver esse problema podemos usar o componente do Primefaces `<p:autoComplete>` . A ideia do componente é disponibilizar um input para o usuário digitar para que com base nos dados digitados, opções apareçam em um menu para serem selecionadas. O componente possui um atributo chamado `completeMethod` que será invocado à medida que o usuário digita no input. Podemos então

criar no ProdutoBean um método que busca produtos de acordo com os dados digitados pelo usuário. Para evitar um número excessivo de queries no banco é recomendável que usemos um atributo da tag chamado `minQueryLength` que determina a partir de quantos caracteres o `completeMethod` será executado.

De maneira análoga a tag `<f:selectItems>` precisamos determinar o `label` e o `value` dos itens do menu que trará o resultado da busca. Esse objetivo é atingido com o uso das tags `itemLabel` e `itemValue`.

```
<h:outputLabel value="Produto:" />
<p:autoComplete value="#{notaFiscalBean.idProduto}"
    completeMethod="#{notaFiscalBean.busca}" minQueryLength="3"
    var="produto" itemLabel="#{produto.nome}" itemValue="#{produto.id}"/>
```

Nosso DAO precisa de um método para realizar a consulta de um produto baseado em seu nome, que será usado pelo ProdutoBean em seu `completeMethod`.

```
public List<Produto> buscaPorNome(String nome) {
    String jpql = "select p from Produto p where "
        + " lower(p.nome) like :nome order by p.nome";
    return manager.createQuery(jpql, Produto.class)
        .setParameter("nome", nome+"%").getResultList();
}
```

12.22 EXERCÍCIO OPCIONAL

1. Substitua o `<h:selectOneMenu>` na tela de adição de itens na nota fiscal pelo componente de Autocomplete do primefaces.



Editora Casa do Código com livros de uma forma diferente

Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.

Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.](#)

12.23 NOVIDADE DO JAVA EE: FILE UPLOAD COM JSF

O upload de arquivos é uma tarefa comum para desenvolvedores web. É por isso as bibliotecas como

Primefaces ou Richfaces criaram componentes que facilitam o trabalho. A partir da versão JSF 2.2 entrou na especificação um componentes para tal tarefa, o `h:inputFile`. Para o `h:inputFile` funcionar devemos adicionar o atributo `enctype="multipart/form-data"` ao `h:form`:

```
<h:form enctype="multipart/form-data">
    <h:inputFile value="#{bean.file}" />
</h:form>
```

VALIDAÇÃO E CONVERSÃO DE DADOS

"A grandeza não consiste em receber honras, mas em merecê-las." -- Aristóteles

Ao término desse capítulo, você será capaz de:

- Utilizar o Hibernate Validator e o Bean Validation;
- Usar validadores do JSF;
- Criar seus próprios conversores e validadores.

13.1 VALIDAÇÃO

Quais dados devemos validar?

O sistema para controle de notas fiscais da UberDist está quase completo, porém os usuários gostariam que não pudesse ser cadastradas notas fiscais com dados inconsistentes. Por exemplo, notas fiscais que não possuam produtos, não fazem sentido serem cadastradas no sistema da UberDist. Assim como nota fiscal sem valor nem data de criação.

O importante é sempre validar o que faz sentido dentro do modelo de negócio da aplicação, mantendo os dados consistentes.

Escrevendo todo o código de validação

Como nossa aplicação é Web, poderíamos realizar a validação dos dados da nota fiscal na própria tela que está sendo controlada pelo browser, utilizando Java Script. Porém, a validação feita utilizando JavaScript pode ser desabilitada no browser, o que poderia causar inconsistência nos dados.

Precisamos fazer algum tipo de validação no lado do servidor, afim de garantir que os dados sejam validados num lugar onde o usuário não consiga interferir. Podemos utilizar as tags de validação do JSF, entretanto, se ele for removido em prol de outro framework, as regras de validação serão perdidas.

Se precisássemos validar nosso modelo na camada de dados, por exemplo, teríamos que repetir muitas validações que já haviam sido feitas na camada de visualização. Sabemos que copiar e colar código não é exatamente a melhor maneira de reaproveitar código. O que precisamos é de algum artifício que seja utilizado em todas as camadas do sistema, ou seja, que a validação seja válida na camada de dados, visualização ou qualquer outra camada que desejarmos.

Já conhece os cursos online Alura?



A **Alura** oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Programação, Front-end, Mobile, Design & UX, Infra e Business, com um plano que dá acesso a todos os cursos. Ex aluno da Caelum tem 15% de desconto neste link!

[Conheça os cursos online Alura.](#)

13.2 VALIDAÇÃO COM BEAN VALIDATION

O JCP idealizou uma especificação que resolvesse este problema. Liderada por Emmanuel Bernard, a JSR 303, conhecida como Bean Validation, define uma série de anotações e uma API para criação de validações para serem utilizadas em Java Beans, que podem ser validados agora em qualquer camada da aplicação.

Como todas especificações, precisamos de uma implementação para utilizarmos em nossa aplicação, a conhecida **Implementação de Referência (RI)**. O Hibernate Validator é a implementação de referência da spec Bean Validation, e será ele que utilizaremos no curso.

Como vimos anteriormente, o JSF possui fases muito bem definidas e estruturadas. Umas destas fases é a fase de validação, que possui compatibilidade com as anotações do Bean Validation. Se estivermos enviando uma requisição para um Managed Bean que possui referência para um Java Bean que por sua vez possui alguma anotação da Bean Validation, ela será aplicado e caso ocorra algum erro, como já sabemos, o JSF enviará o cliente para a página de onde a requisição foi feita.

Por exemplo, se, no Cadastro de Produto, quisermos que um produto tenha seu nome preenchido e que o valor seja pelo menos maior que 1, basta anotarmos na classe `Produto` o atributo `nome` com `@NotEmpty`:

```
@Entity
public class Produto {

    @Id @GeneratedValue
    private Long id;

    @NotEmpty
    private String nome;

    private String descricao;
```

```
private Double preco;  
  
//getters e setters  
}
```

Pronto! Com uma única anotação, qualquer objeto do tipo `Produto` será validado tanto na camada de visualização quanto na camada de persistência.

Como estamos falando de JSF, na fase de validação será verificado se o nome do produto não está vazio. Se estiver, as fases do JSF serão interrompidas e o cliente será redirecionado para a tela de onde a requisição foi feita.

@NotNull ou @NotEmpty?

Na especificação da Bean Validation existe uma anotação chamada `@NotNull` que, como o próprio nome diz, valida se um determinado atributo está nulo. Porém, no JSF, quando não preenchemos nenhum valor na tela, o atributo é setado como uma String vazia "", o que não reflete o que desejamos fazer na realidade, que é não permitir que um produto seja cadastrado sem nome.

No Hibernate Validator, existe uma anotação chamada `@NotEmpty`, que possui uma diferença sutil em relação a `@NotNull`. Ela valida se a String não é nula mas também se a String é diferente de "".

O importante é notar que poderíamos fazer essa validação de várias maneiras, como ainda com a anotação `@Length` passando `min=1`.

Se tentarmos cadastrar um produto com o nome vazio, não conseguiremos devido a validação que foi colocada. Percebemos isso porque nenhum produto é cadastrado no banco de dados, porém o usuário não sabe de fato o que aconteceu no processo.

Isso acontece porque apesar do JSF efetuar a validação, ele não imprime o erro que aconteceu de maneira automática, precisamos colocar na tela de onde foi efetuada a requisição uma tag `<h:messages />` que imprimirá todos os possíveis erros e validação e também de conversão que veremos mais a frente.

Se adicionarmos a tag `<h:messages />` no arquivo `produto.xhtml`, veríamos o erro da seguinte maneira:

The screenshot shows a JSF page with the title "Novo Produto". Below the title is a button labeled "Dados do Produto". To the right of the button is a text input field with the label "Nome:". Above the input field, there is a red rectangular box containing the text "may not be empty".

Essa mensagem em inglês na maioria das vezes não é o que queremos. O que queremos é na realidade imprimir alguma mensagem customizada, mais parecida com o que condiz com o nosso negócio.

A anotação `@NotEmpty` assim como as outras presentes no Bean Validation e no Hibernate Validator permite que seja especificada uma mensagem customizada. Basta setar o atributo `message` da anotação e digitar a mensagem que queremos que seja exibida. Por exemplo:

```
@Entity
public class Produto {

    @Id @GeneratedValue
    private Long id;

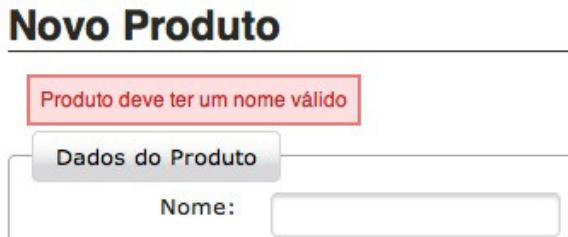
    @NotEmpty(message = "Produto deve ter um nome válido")
    private String nome;

    private String descricao;

    private Double preco;

    //getters e setters
}
```

Com isso, a mensagem exibida seria igual a imagem abaixo:



Existe uma outra anotação que abre um leque de possibilidades muito grande quando precisamos validar algum Java Bean. A anotação `@Pattern` facilita muito o trabalho e poupa um trabalho enorme, principalmente quando precisamos validar itens não tão genéricas, como email, CPF, CNPJ, etc. Ela evita a necessidade de termos que fazer anotações de validação própria como é visto no curso FJ-25.

A mesma validação que fizemos com o `@NotEmpty` podemos resolver com o `@Pattern` que permite que seja passada uma expressão regular. No caso da classe `Produto` podemos fazer a mesma validação da seguinte maneira:

```
@Entity
public class Produto {

    @Id @GeneratedValue
    private Long id;

    @Pattern(regexp = "[^ ]", message = "Produto deve ter um nome válido")
    private String nome;
```

```

private String descricao;
private Double preco;
//getters e setters
}

```

Teríamos o mesmo efeito! Como será que a anotação `@NotEmpty` do Hibernate Validator funciona? Utilizando a anotação `@Pattern` internamente.

13.3 BEAN VALIDATION SEM SERVIDOR DE APLICAÇÃO

Como usamos o Glassfish não precisamos nos preocupar com a configuração e inicialização do Bean Validation. No entanto pode ser útil ou necessário usar o Bean Validation fora do servidor de aplicação, por exemplo, dentro de um servlet container como Apache Tomcat.

Para tal, encontramos os JARs no site do Hibernate Validator:

<http://www.hibernate.org/subprojects/validator/download>

Depois ter adicionado todos os JARs no classpath podemos criar um `Validator`, o objeto principal do Bean Validation. O `Validator` é criado pela `ValidatorFactory`:

```

ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
Validator validator = factory.getValidator();

```

Com o `Validator` em mãos fica fácil aplicar as regras de validação, basta chamar o método `validate` que devolve um `Set` com as mensagens de erro:

```

Produto produto = new Produto(); //produto inválido
Set<ConstraintViolation<Produto>> validate = validator.validate(produto);
for (ConstraintViolation<Produto> constraintViolation : validate) {
    System.out.println(constraintViolation.getMessage());
}

```

13.4 EXERCÍCIOS: INTEGRANDO BEAN VALIDATION COM O JSF

1. Todos os JARs já vêm com o Glassfish e o nosso projeto está pronto para usar o Bean Validation. Podemos começar a definir as regras de validação.

Por isso altere a classe `Produto` para que ela passe a ser validada. **Adicione** a anotação seguinte na classe:

```

@Entity
public class Produto {

    @Id @GeneratedValue
    private Long id;

    @NotEmpty

```

```

    private String nome;
    private String descricao;
    private Double preco;
    //getters e setters
}

```

2. **Adicione** no arquivo `produto.xhtml` o componente `<h:messages/>` logo dentro do formulário de cadastro. Você pode usar a `styleClass="erros"` para ter um design mais interessante.
3. Acesse a página e faça o teste deixando o campo nome vazio. Veja que aparece a mensagem padrão.
4. **Altere** a anotação, passando uma mensagem customizada:

```

@NotEmpty(message = "Produto deve ter um nome válido")
private String nome;

```

5. Acesse novamente a página, refaça o teste e veja nossa mensagem customizada.

Novo Produto

6. Valide também o campo `cnpj` da `NotaFiscal`. Você pode usar a anotação `@Pattern` com a definição da regex para CNPJ:

```

@Pattern(regexp = "\\\d{2}\\.\\\\d{3}\\.\\\\d{3}/\\\\d{4}-\\\\d{2}",
         message = "CNPJ inválido")
private String cnpj;

```

Para mostrar as mensagens de erro das notas fiscais, **adicione** o componente `<h:messages/>` dentro do formulário do arquivo `notafiscal.xhtml`:

```

<h:messages styleClass="erros" />

```

Faça o teste do cadastro de notas tentando passar um dado inválido.

DOCUMENTAÇÃO DO HIBERNATE VALIDATOR

O Hibernate Validator possui uma documentação muito vasta, onde encontramos material o suficiente para validarmos nosso modelo com as possíveis anotações da Bean Validation.

No curso FJ-25 da Caelum, são mostrados mais exemplos e casos de validação. Assim como a definição de constraints no banco de dados com JPA e Hibernate.

A documentação encontra-se no site:
<http://docs.jboss.org/hibernate/stable/validator/reference/en/html/>

Saber inglês é muito importante em TI

galandra O Galandra auxilia a prática de inglês através de *flash cards* e *spaced repetition learning*. Conheça e aproveite os preços especiais.

[Pratique seu inglês no Galandra.](#)

13.5 VALIDAÇÃO COM COMPONENTES JSF

Apesar de possuir a integração natural com a Bean Validations, nem sempre é possível utilizá-lo por motivos de restrições na empresa ou por precisarmos usar o JSF 1.x. Portanto, precisamos realizar as validações de outra maneira.

Para isso, o JSF possui alguns validadores próprios que podemos utilizar. Os validadores são acionados através de Tags as quais associamos com um elemento de entrada de dados. O JSF 2 possui 4 Tags de validação:

- <f:validateDoubleRange> : Verifica se um valor Double se encontra dentro de um intervalo pré-determinado;
- <f:validateLongRange> : Verifica se um valor Long se encontra dentro de um intervalo pré-determinado;
- <f:validateLength> : Verifica se um texto possui uma quantidade de caracteres pré-determinada;
- <f:validateRegex> : Verifica se um texto segue uma determinada expressão regular.

Para indicarmos que o campo de nome do Produto não deve ter mais do que 100 caracteres, poderíamos utilizar a Tag `f:validateLength`:

```
<h:inputText value="#{produtoBean.produto.nome}">
    <f:validateLength maximum="100" />
</h:inputText>
```

Repare que registramos o componente `f:validateLength` **dentro** do `inputText`. Com isso, as validações são disparadas na terceira fase do JSF, onde ele converte os dados submetidos e os valida.

É possível ainda customizar a mensagem de validação quando adicionamos os validadores do JSF:

```
<h:inputText value="#{produtoBean.produto.nome}"
    validatorMessage="Mínimo 100 caracteres"
        <f:validateLength maximum="100" />
</h:inputText>
```

Há ainda outras facilidades, como suporte a marcar campos como requeridos diretamente no componente:

```
<h:inputTextarea value="#{produtoBean.produto.descricao}"
    required="true" requiredMessage="Campo descrição requerido" />
```

13.6 EXERCÍCIOS: COMPONENTES DE VALIDAÇÃO

1. Faça com que o campo do valor do produto seja obrigatório e só aceite valores maiores que 50 centavos. Para isso, utilize os recursos de validação do JSF.

```
<h:inputText value="#{produtoBean.produto.preco}"
    required="true" requiredMessage="Preço obrigatório"
    validatorMessage="O valor mínimo é 0,50">

    <f:validateDoubleRange minimum="0.50"/>

</h:inputText>
```

13.7 VALIDAÇÕES CUSTOMIZADAS COM JSF

Apesar do JSF possuir alguns componentes prontos para realizar a validação, em certos casos, eles podem não ser suficientes ao precisarmos de alguma validação um pouco mais específica. Nesse caso, podemos criar métodos que realizam essa validação para nós. Com isso, ao invés de registrarmos um componente de validação, registraremos o método de validação no componente de entrada de dados.

Todos os componentes de entrada de dados possuem um atributo chamado `validator` para o qual podemos passar como parâmetro uma referência para um método de um Managed Bean que realizará a validação:

```
<h:inputText value="#{produtoBean.produto.nome}"
    validator="#{produtoBean.comecaComMaiuscula}" />
```

No exemplo anterior, queremos verificar que o nome do produto deverá começar com letra

maiúscula e estamos registrando no `inputText` um método que fará essa verificação. Esse método, que no nosso caso se chama `comecaComMaiuscula`, deve receber três parâmetros: um objeto do tipo `FacesContext`, um `UIComponent` e um `Object` representando o valor para ser validado. Caso o erro de validação ocorra, o método deve lançar uma `ValidatorException` indicando ao JSF que houve uma falha de validação:

```
public void comecaComMaiuscula(FacesContext fc,
    UIComponent component, Object value)
    throws ValidatorException {

    String valor = value.toString();
    if (!valor.matches("[A-Z].*")) {
        // dá um erro
    }
}
```

Podemos ainda devolver uma mensagem para o usuário indicando qual erro de validação ocorreu. Para isso, basta passar uma instância de `FacesMessage` para o construtor da `ValidatorException`, contendo a mensagem que deve ser exibida.

```
throw new ValidatorException(
    new FacesMessage("Deveria começar com letra maiúscula"));
```

Seus livros de tecnologia parecem do século passado?



Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**. Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

[Casa do Código, Livros de Tecnologia.](#)

13.8 EXERCÍCIO: CRIANDO MÉTODOS DE VALIDAÇÃO

1. Crie um método no `ProdutoBean` que valide que o nome do `Produto` começa com uma letra maiúscula. Associe-o com o `inputText` adequado e faça com que caso haja erro de validação, seja devolvida uma mensagem informativa ao usuário.

```
public void comecaComMaiuscula(FacesContext fc,
    UIComponent component, Object value)
    throws ValidatorException {

    String valor = value.toString();
    if (!valor.matches("[A-Z].*")) {
```

```

        throw new ValidatorException(
        new FacesMessage(
            "Deveria começar com maiúscula")
        );
    }

}

```

- O que acontece se você precisa dessa validação em outras telas? Será que temos que copiar o método para outro Managed Bean? E parece um boa ideia acoplar essas validações com um managed bean de negócio? E os cuidados com escopos?

13.9 EVITANDO DUPLICIDADES DE VALIDAÇÕES CRIANDO CLASSES VALIDADORAS

Muitas vezes, uma aplicação usa um mesmo tipo de validação em diversos pontos diferentes, com isso, poderíamos utilizar a validação da primeira letra maiúscula em vários lugares.

Uma primeira abordagem para resolver esse problema seria copiar os métodos de validação nos vários Managed Beans que deverá ser utilizado. No entanto, isso dá muito trabalho e começamos a duplicar código, que sempre é algo que deve ser evitado ao máximo.

Uma segunda abordagem para resolvemos esse problema seria apontar em diversas telas para o método de validação que criamos em nosso Managed Bean, no entanto, com isso começamos a complicar nossas telas, pois elas vão começar a depender de diferentes Managed Beans que não possuem relação alguma com os mesmos.

Uma outra solução seria isolar as validações em um outro managed bean específico, por exemplo, um `ValidatorBean`.

Mas melhor que usar managed beans para organizar as validações, o JSF permite que nós isolemos os métodos de validação que vamos utilizar em diversos pontos da nossa aplicação em *classes de validação*. Com isso, sempre que quisermos nos referenciar a um validador, nos referenciamos a sua classe e assim eliminamos a duplicidade.

Para criar uma classe de validação, basta implementar a interface `javax.faces.validator.Validator` e o seu método `validate` da mesma forma de quando criamos um método de validação.

Também devemos registrar o validador no JSF. Nas versões anteriores, éramos obrigados a registrá-lo no arquivo `faces-config.xml`. Já na versão 2.0, basta anotarmos a classe de validação com `@FacesValidator` passando como argumento a identificação da validação para ser usada depois:

```

@FacesValidator("comecaComMaiuscula")
public class ValidadorComecaComMaiuscula implements Validator {

    public void validate(FacesContext fc,

```

```

UIComponent component, Object value)
throws ValidatorException {

    String valor = value.toString();
    if (!valor.matches("[A-Z].*")) {
        throw new ValidatorException(
            new FacesMessage(
                "Deveria começar com letra maiúscula")
        );
    }
}
}

```

Por fim, podemos utilizar a Tag `f:validator` para registrarmos a nossa classe de validação a um `input` :

```

<h:inputText value="#{produtoBean.produto.nome}">
    <f:validator validatorId="comecaComMaiuscula"/>
</h:inputText>

```

Pronto, conseguimos um validador totalmente isolado e desacoplado de nossas lógicas.

13.10 EXERCÍCIOS OPCIONAIS: @FACESVALIDATOR

1. Faça com que o método de validação criado no exercício anterior possa ser facilmente utilizado em diferentes telas. Para isso, crie uma *classe de validação* e registre-a no JSF com a anotação `@FacesValidator`. Troque o `inputText` para usar essa classe de validação ao invés do método criado anteriormente.

Agora é a melhor hora de respirar mais tecnologia!



Se você está gostando dessa apostila, certamente vai aproveitar os **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum. Programação, Mobile, Design, Infra, Front-End e Business! Ex-aluno da Caelum tem 15% de desconto, siga o link!

[Conheça a Alura Cursos Online.](#)

13.11 CONVERSORES DE DADOS COM O JSF

Sempre que nosso bean receber os dados em algum atributo que não seja String, ocorre um processo de conversão. O campo `quantidade`, por exemplo, é um `Integer` portanto deve ser convertido com `Integer.parseInt`.

O JSF já possui diversos conversores embutidos para os tipos básicos e Strings. Mas há também componentes conversores para certas configurações adicionais. O papel de um conversor é transformar a String digitada pelo usuário no tipo correto, além de transformar de volta em String na hora de exibir.

Já usamos conversores no curso em alguns momentos. No caso das datas, usamos o componente `<f:convertDateTime>` para especificar o formato das datas:

```
<h:inputText value="#{notaFiscalBean.notaFiscal.data.time}">
    <f:convertDateTime pattern="dd/MM/yyyy" timeZone="America/Sao_Paulo" />
</h:inputText>
```

Usamos também o `<h:convertNumber>` para indicar uma máscara de preço:

```
<h:outputText value="#{item.total}">
    <f:convertNumber pattern="R$ #0.00"/>
</h:outputText>
```

É possível ainda usar diversas outras opções, como:

```
<f:convertNumber type="currency" currencySymbol="R$" locale="pt_BR" />
```

13.12 IMMEDIATE

Um exercício opcional do capítulo 3 trouxe ao nosso sistema a capacidade de cancelar a edição de produtos. É um simples botão *Cancelar Edição* que limpa o formulário chamando um método `cancela` na classe `ProdutoBean`:

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

O botão em si apenas chama esse método e renderiza novamente o formulário, agora em branco:

```
<h:commandButton value="Cancelar edição"
    action="#{produtoBean.cancela}" />
```

Essa implementação do botão de cancelar funcionou muito bem até que adicionamos a validação do produto. Agora, por estarmos fazendo validação de vários campos do produto, qualquer action sendo executada nesse formulário vai executar todas as validações. E não queremos isso quando estamos apenas cancelando.

Faça o teste: selecione algum produto para alteração, preencha o preço com alguma letra e tente cancelar a edição. Note que a validação é executada e esse erro não permitirá o cancelamento da edição. Só é possível cancelar quando todos os campos estão válidos, o que não faz muito sentido.

Porque isso acontece? É reflexo direto do fluxo de execução do ciclo de vida normal de um request JSF. Como vimos, a fase `INVOKER_APPLICATION` só é invocada após a `PROCESS_VALIDATIONS` e apenas se tudo estiver certinho. Mas como executar o método `cancela` sem fazer as validações?

O JSF tem um recurso que nos permite puxar o processamento do componente para a fase

APPLY_REQUEST_VALUES, antes das validações, invertendo o ciclo do JSF. Basta adicionar o atributo `immediate="true"` ao botão. Isso fará com que o método `cancela` seja executado ao fim da fase APPLY_REQUEST_VALUES e, logo em seguida, vá para a RENDER_RESPONSE:

```
<h:commandButton value="Cancelar edição"
    action="#{produtoBean.cancela}"
    immediate="true"/>
```

Apenas essa modificação faz com que o cancelamento ocorra sem que a validação seja executada. Mas um novo problema surge: o formulário segue "sujo" com os dados anteriores, mesmo nosso código fazendo `new Produto()`. O problema é que o componente mantém ainda o valor, no chamado *submitted value*.

13.13 SUBMITTED VALUE

Quando trabalhamos com componentes de input, três valores estão envolvidos durante o ciclo de vida:

- **Submitted value:** Dentro do componente, é um campo que guarda a String do parâmetro HTTP submetido, sem tratamento algum. É o resultado simples da chamada a `request.getParameter`. Esse valor é setado no componente durante a APPLY_REQUEST_VALUES.
- **Local value:** Na fase PROCESSVALIDATIONS, o *submitted value* é convertido e validado. Em caso de sucesso, o valor convertido é setado como *local value*. O *submitted value* é então apagado.
- **Value binding (o valor do modelo):** Na fase seguinte, a UPDatemodel_VALUES, o modelo recebe o valor do *local value*, que logo em seguida é limpo. Nesse ponto, tanto o *submitted* quanto o *local value* são nulos e o valor está apenas no modelo. As fases INVOKE_APPLICATION e RENDER_RESPONSE prosseguem a partir daí.

Na hora de renderizar o componente, o JSF procura por um valor exatamente na ordem acima. Se houver um *submitted value*, ele é usado; caso contrário, ele tenta usar o *local value*. E, em último caso, é visto o valor do modelo.

Mas o que tudo isso tem a ver com nosso problema do formulário sujo quando usamos o `immediate`?

Nosso método `cancela` está limpando apenas o valor no modelo, mas isso não é suficiente. Ao usar o `immediate`, o ciclo de vida foi modificado de maneira que as validações não foram executadas e, portanto, o *submitted value* continua populado. Mesmo zerando o valor do modelo, o JSF usará o *submitted* na hora de renderizar a tela.

Como resolver? A solução é bastante simples, basta limpar o *submitted value* dos componentes

manualmente - há um método `setSubmittedValue` nos inputs. Mas é bastante trabalhoso fazer isso manualmente para todos os componentes (teríamos que percorrer a árvore toda recursivamente e limpar componente por componente).

Uma solução mais simples seria forçar que o `ViewRoot` seja recriado e, portanto, todos os componentes comecem zerados. Nem sempre essa solução é viável, mas no nosso caso é suficiente já queremos justamente jogar tudo fora e começar de novo. Outra forma é causar uma navegação, possivelmente para a mesma tela, invalidando o `ViewRoot` atual:

```
public String cancela() {  
    return "produto";  
}
```

Ou, ainda mais fácil, podemos usar o outcome direto no atributo action do botão:

```
<h:commandButton value="Cancelar edição"  
action="produto" immediate="true" />
```

Editora Casa do Código com livros de uma forma diferente



Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.

Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.](#)

13.14 EXERCÍCIOS: IMMEDIATE

1. Caso ainda não tenha feito o botão de cancelar edição de produtos, faça-o agora. Basta um simples método `cancela` na classe `ProdutoBean` :

```
public void cancela() {  
    System.out.println("Cancela edição");  
    this.produto = new Produto();  
}
```

Crie também o botão que chama esse método:

```
<h:commandButton value="Cancelar edição"  
action="#{produtoBean.cancela}" />
```

2. Faça alguns testes com o botão de cancelamento.

Selecione um elemento já adicionado e clique para editá-lo. Em seguida, apenas cancele a edição. O elemento provavelmente estava válido então o cancelamento funciona.

Agora, teste editar outro produto mas altere algum campo para deixá-lo inconsistente (como deixar o nome em branco). Ao tentar cancelar a edição, ocorrerá o erro de validação e a edição não será cancelada.

3. Faça o método `cancela` ser executado mesmo no caso dos erros de validação, usando `immediate="true"` no botão cancelar.

Refaça os testes. Note que o formulário continua preenchido.

4. Force a reconstrução da árvore de componentes usando uma navegação para a própria página. Altere a `action` do botão para "produto":

```
<h:commandButton value="Cancelar edição"
                  action="produto" immediate="true" />
```

Refaça os testes e note que agora tudo funciona corretamente.

13.15 CONVERSOR DE PRODUTO

Na tela de cadastro de nota fiscal, criamos um combo box para escolher um produto que seria adicionado na nota. Este combo mandava o id do produto para o `NotaFiscalBean` e dentro desse *bean* disparamos uma pesquisa deste produto.

Porém, este modelo é pouco orientado a objetos, o mais correto seria que o `NotaFiscalBean` recebesse diretamente um produto. Mas como fazer isso se forçadamente estamos trabalhando na web e não podemos transitar objetos?

É exatamente para isso que servem os conversores do JSF.

Neste conversor, receberemos o id do produto e através dele buscaremos o objeto completo no banco. Para isso precisamos de um `entityManager` ou um `dao`.

```
@FacesConverter(forClass=Produto.class)
public class ProdutoConverter implements Converter {

    public Object getAsObject(FacesContext ctx,
                             UIComponent comp, String value) {
        Long id = Long.valueOf(value);
        return entityManager.find(Produto.class, id);
    }

    public String getAsString(FacesContext ctx,
                            UIComponent comp, Object value) {
        Produto produto = (Produto) value;
        return String.valueOf(produto.getId());
    }
}
```

Mas o JSF, por padrão, não consegue injetar nem um, nem outro em conversores. É exatamente nessa deficiência que o CDI ajuda. Para o CDI, os conversores podem ser tratados como um componente qualquer. Neste caso, podemos inclusive receber dependências através da anotação `@Inject` e escolher o escopo do conversor.

```
@FacesConverter(forClass=Produto.class)
@RequestScoped
public class ProdutoConverter implements Converter {

    @Inject
    private EntityManager entityManager;

    public Object getAsObject(FacesContext ctx,
        UIComponent comp, String value) {
        Long id = Long.valueOf(value);
        return entityManager.find(Produto.class, id);
    }

    public String getAsString(FacesContext ctx,
        UIComponent comp, Object value) {
        Produto produto = (Produto) value;
        return String.valueOf(produto.getId());
    }
}
```

13.16 EXERCÍCIOS OPCIONAIS

1. Escreva um conversor para a classe produto, este conversor deve receber o id do produto e convertê-lo para um produto, buscando-o no banco e vice-versa:

```
@FacesConverter(forClass=Produto.class)
@RequestScoped
public class ProdutoConverter implements Converter{

    @Inject
    private EntityManager entityManager;

    public Object getAsObject(FacesContext ctx, UIComponent comp, String value) {
        Long id = Long.valueOf(value);
        return entityManager.find(Produto.class, id);
    }

    public String getAsString(FacesContext ctx, UIComponent comp, Object value) {
        Produto produto = (Produto) value;
        return String.valueOf(produto.getId());
    }
}
```

2. Modifique o combo box de produto no arquivo `notafiscal.xhtml` para que ele trabalhe agora com produto diretamente não mais com id:

```
<h:selectOneMenu value="#{notaFiscalBean.item.produto}">
    <f:selectItems value="#{produtoBean.produtos}" var="produto"
        itemValue="#{produto}" itemLabel="#{produto.nome}" />
</h:selectOneMenu>
```

Na classe `NotaFiscalBean`, remova o atributo `idProduto` e a lógica de buscar o produto que havia no método `guardaItem`.

3. Para garantir a integridade dos valores, o JSF busca o objeto retornado pelo conversor dentro da lista de opções do combo box, conferindo se o objeto veio da árvore mesmo, ele faz essa verificação através do método `equals`, por isso devemos implementar o `equals` e `hashCode` da classe `Produto`:

- Abra a classe `Produto`%
- Entre no menu *Source* e escolha a opção *Generate hashCode e equals*
- Deixe marcado todos os atributos
- Clique em ok

Já conhece os cursos online Alura?



A Alura oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Programação, Front-end, Mobile, Design & UX, Infra e Business, com um plano que dá acesso a todos os cursos. Ex aluno da Caelum tem 15% de desconto neste link!

[Conheça os cursos online Alura.](#)

APÊNDICE - UM POUCO MAIS SOBRE O JSF

"A grandeza não consiste em receber honras, mas em merecê-las." -- Aristóteles

A integração do JSF 2 com CDI abre novas possibilidades para nosso projeto. Muita coisa pode ser simplificada com o novo modelo de desenvolvimento do Java EE.

14.1 LIDANDO COM REQUISIÇÕES GET NO JSF

No JSF todas as requisições disparadas por uma árvore de componentes eram do tipo POST, elas são chamadas de *Postback*, caso seja feita uma requisição do tipo GET o JSF recebe a requisição, mas não processa as etapas intermediárias, somente é executada a primeira (*RESTORE_VIEW*) e a última (*RENDER_RESPONSE*), dessa maneira ele apenas monta a árvore de componentes e renderiza o HTML de resposta.

Essa limitação impedia que JSF execute alguma operação na primeira requisição (geralmente do tipo GET) ao sistema. Um exemplo dessa limitação é a seguinte requisição:

`http://localhost:8080/produto.xhtml?produto.id=10`

É extremamente complicado fazer essa requisição abrir o produto de id 10 já em modo de edição logo na primeira requisição, pois para o JSF essa requisição não veio de uma árvore e por isso não podemos executar nenhuma action.

Porém na versão mais nova do JSF é possível executar uma ação, mesmo em requisições GET, através da tag `<f:metadata>`. Com ela podemos receber parâmetros e executar ações. Essas ações são executadas imediatamente antes do da fase *RENDER_RESPONSE*, como se fosse um `PhaseListener`.

Vamos trabalhar com a situação que queremos que logo na primeira requisição possamos abrir um produto para edição.

Na página de produtos, dentro da tag `h:body` colocaremos o componente `f:metadata`.

```
<h:body>
  <f:metadata>
    </f:metadata>
</h:body>
```

Para ler um parâmetro de requisição, precisamos de um componente para recebê-lo e um atributo em um Managed Bean para guardá-lo. O componente que recebe um valor do *request* é o `<f:viewParam>`. Ele tem dois atributos, um é o nome do parâmetro a ser lido (`name`) e o outro é o atributo do managed bean que será feito o *binding* (`value`).

```
<h:body>
    <f:metadata>
        <f:viewParam name="produto.id" value="#{produtoBean.produtoId}" />
    </f:metadata>
</h:body>
```

Quando for executada a requisição passando o parâmetro `produto.id`, o JSF setará esse id no atributo `produtoId` do `produtoBean`.

Saber inglês é muito importante em TI

galandra O **Galandra** auxilia a prática de inglês através de *flash cards* e *spaced repetition learning*. Conheça e aproveite os preços especiais.

[Pratique seu inglês no Galandra.](#)

14.2 NOVIDADE DO JAVA EE 7: VIEWACTION DO JSF

Mas só isso não basta para abrir a tela em modo edição, ainda precisamos buscar o produto no banco a partir do id que veio da requisição, para isso usamos o componente `<f:viewAction>`, com ele podemos escolher um método que será disparado na etapa *Invoke_Application*.

```
<h:body>
    <f:metadata>
        <f:viewParam name="produto.id" value="#{produtoBean.produtoId}" />
        <f:viewAction action="#{produtoBean.carregaProduto}" />
    </f:metadata>
</h:body>
```

Como o formulário da página de produto já está associado ao atributo `produto`, basta que esse `<f:viewAction>` dispare uma consulta simples ao banco e guardar a resposta no atributo `produto`.

Uma das exigências da especificação é que a tag `<f:metadata>` só será corretamente ativada se for filha da tag `<f:view>` e que a requisição disparada seja feita à página que contém o `<f:metadata>`. Por exemplo se fizermos a requisição para o endereço <http://localhost:8080/produto.xhtml?produto.id=10> o arquivo que deve ter o `<f:metadata>` é `produto.xhtml`.

14.3 EXERCÍCIOS: F:METADATA E F:VIEWACTION

1. Adicione ao sistema a capacidade de abrir a página de alteração de produto, diretamente através de passagem de parâmetros.

- Abra o arquivo **produto.xhtml**

- Adiciona antes da tag `ui:define` o seguinte trecho de código

```
<f:metadata>
    <f:viewParam name="produto.id" value="#{produtoBean.produtoId}" />
    <f:viewAction action="#{produtoBean.carregaProduto}" />
</f:metadata>
```

- Abra a classe `ProdutoBean`

- Adicione um novo atributo para receber o id do Produto

```
private Long produtoId;
```

- Crie o **getter** e **setter** desse atributo

- Crie um novo método para carregar um produto caso o parâmetro esteja preenchido

```
public void carregaProduto() {
    if(produtoId != null && produtoId != 0){
        this.produto = dao.buscaPorId(this.produtoId);
    }
}
```

- 2.

- Entre na página de login e logue-se no sistema
- Agora acesse a url <http://localhost:8080/fj26-notas-fiscais/produto.xhtml?produto.id=30> (não esqueça de passar como parâmetro um id existente no banco de dados).

14.4 DISPARANDO REQUISIÇÕES DO TIPO GET ATRAVÉS DE LINKS JSF

No exemplo anterior adicionamos suporte à requisições do tipo GET à página de produtos, porém nosso link de alteração não usa esta funcionalidade.

Para permitir que o usuário adicione aos favoritos o link da edição de um produto, precisamos alterar como o JSF dispara esta requisição. Atualmente o componente `<h:commandLink>` sempre dispara uma requisição do tipo POST e não passa nenhum parâmetro junto do endereço.

Neste caso então, o componente `<h:commandLink>` não nos serve, precisamos de um outro componente, que faça requisições GET e dê suporte à adição de parâmetros na URL. O componente que faz isso é o `<h:outputLink>`, este componente permite fazer requisições do tipo GET, inclusive para outros sites, como no exemplo abaixo:

```
<h:outputLink value="http://www.caelum.com.br">
    <h:outputText value="Site da Caelum"/>
</h:outputLink>
```

No nosso caso além de uma chamada GET temos que adicionar o id do produto a ser editado, usando o `<f:param>` com *name* e *value* dentro do `<h:outputLink>` podemos adicionar este parâmetro:

```
<h:outputLink value="produto.xhtml">
    <h:outputText value="Editar Produto via GET"/>
    <f:param name="produto.id" value="#{produto.id}"/>
</h:outputLink>
```

Seus livros de tecnologia parecem do século passado?



Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**. Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

[Casa do Código, Livros de Tecnologia.](#)

14.5 EXERCÍCIOS: REQUISIÇÕES DO TIPO GET

1. Adicione mais uma coluna na listagem de produto:

```
<p:column headerText="Editar">
</p:column>
```

2. Dentro desta nova coluna adicione um link que faça requisições do tipo GET, passando o id do produto como parâmetro:

```
<h:outputLink value="produto.xhtml">
    <h:outputText value="Editar Produto via GET"/>
    <f:param name="produto.id" value="#{produto.id}"/>
</h:outputLink>
```

3. Teste este novo link e tente adicioná-lo aos favoritos do seu browser e veja que podemos acessar um produto diretamente agora!

14.6 BINDING DE COMPONENTES DO JSF

Um componente JSF é muito mais que uma simples tag de view. Como sabemos, há todo o gerenciamento de estado além de funcionalidades integrados a todo o ciclo de vida do JSF.

Cada componente, além de ter uma tag para ser usada na View facilmente, possui também uma classe Java correspondente que podemos usar para manipular programaticamente o componente. Por

exemplo, um `<h:dataTable>` pode ser referenciado em Java como `HtmlDataTable`, ou um `<p:dataTable>` através de `org.primefaces.component.datatable.DataTable`.

Através dessa classe, podemos obter informações importantes sobre o componente, como a linha que foi clicada com `getRowData`.

Recuperar o componente no código Java é simples: basta usarmos o atributo `binding` no xhtml referenciando a propriedade no managed bean que receberá o objeto do componente:

```
<h:dataTable value="#{notaFiscalBean.notaFiscal.itens}" var="item"
    styleClass="dados" rowClasses="impar,par" id="tabelaNota"
    binding="#{meuBean.tabela}">
```

Repare que estamos tratando de um novo tipo de binding. Até agora fazíamos apenas o binding do valor do componente com o atributo `value`. Além dele, agora temos o atributo `binding` que nos permitirá recuperar e manipular o componente como um todo, não apenas seu valor:

```
public class MeuBean {
    private HtmlDataTable tabela;

    // ...
}
```

Vamos usar o binding para implementar uma nova funcionalidade no sistema: a remoção de Itens da nota fiscal ainda durante o cadastro da mesma. Lembre que esses dados ficam em memória até o momento em que é feita a adição no banco. Os dados, portanto, ficam no managed bean e nos componentes da tela e é justo deles que precisamos remover.

A tela é simples: vamos apenas adicionar uma coluna a mais na tabela de itens com um link de remoção:

```
<h:column>
    <h:commandLink action="#{meuBean.removeItem}" value="Remover"/>
</h:column>
```

A lógica de remoção acessa o `Item` e o remove da lista:

```
@Named @ViewScoped
public class NotaFiscalBean {

    // ... outros atributos e métodos

    private HtmlDataTable tabela;

    public void removeItem() {
        Item item = (Item) tabela.getRowData();
        notaFiscal.getItens().remove(item);
    }

    // getter e setter da tabela
}
```

O código anterior mostra ainda um recurso interessante de se manipular o componente direto no código Java: conseguimos acessar propriedades internas do mesmo e executar certas lógicas. No caso,

chamamos o método `getRowData()` da classe `HtmlDataTable` que nos devolve o `Item` da tabela que teve o link *Remover* clicado.

Para saber mais: O `@ManagedProperty`

O JSF puro possui um recurso mínimo de injeção onde conseguimos acessar os próprios managed beans do JSF dentro de outros managed beans. Não é um mecanismo completo de injeção como CDI, mas resolveria o caso onde queremos acessar a `NotaFiscalBean` de dentro da `RemoveItemBean`.

Esse recurso se chama **managed property** e existe desde o JSF 1. Mas no JSF 2, foi bastante simplificado com a anotação `@ManagedProperty`. Ela permite que acessemos uma propriedade via EL de dentro do managed bean:

```
@ManagedBean
public class RemoveItemBean {

    @ManagedProperty("#{notaFiscalBean}")
    private NotaFiscalBean notaFiscalBean;

    private HtmlDataTable tabela;

    public void removeItem() {
        Item item = (Item) tabela.getRowData();
        notaFiscalBean.getNotaFiscal().getItens().remove(item);
    }

    // ...
}
```

Como estamos usando CDI e nosso bean não é mais `@ManagedBean`, não podemos usar *managed properties*. E mesmo que tenhamos um caso onde seja necessário injetar um outro managed bean, com CDI podemos usar o `@Inject` que é bem mais simples e poderoso.

Mas se fossemos usar JSF puro, ainda esbarriaríamos em mais um problema. O *managed property* não nos permite acessar um outro bean que seja *view scoped*, como é o caso do `NotaFiscalBean`.

O código até rodará mas o objeto injetado será um novo e não aquele que gostaríamos de recuperar. Uma forma de resolver é aumentar o escopo do bean para `@SessionScoped` ou melhor manter tudo em `@ViewScoped``.

14.7 EXERCÍCIOS: BINDING

Implemente a remoção de Itens na tabela do cadastro de notas fiscais.

1. Adicione o atributo `tabela` na `NotaFiscalBean` e gere seu getter e setter:

```
private HtmlDataTable tabela; //get e set
```

Importante: se sua tela usa a tabela do PrimeFaces, use a classe `DataTable` pro binding.

2. O próximo passo é criar o método `removeItem` ainda na classe `NotaFiscalBean`. Ele deve acessar o valor da tabela através do `getRowData`:

```
public void removeItem() {  
    Item item = (Item) tabela.getRowData();  
    notaFiscal.getItens().remove(item);  
}
```

3. Faça o binding da propriedade com o `<h:dataTable>` do arquivo `notafiscal.xhtml`:

```
<h:dataTable value="#{notaFiscalBean.notaFiscal.itens}" var="item"  
    styleClass="dados" rowClasses="impar,par" id="tabelaNota"  
    binding="#{notaFiscalBean.tabela}">
```

4. Crie uma nova coluna no `<h:dataTable>` com um link para remoção e que invoque o método `removeItem`.
5. Reinicie o Glassfish e faça os testes adicionando e removendo Itens durante o cadastro da nota fiscal.
6. (opcional) De que outras formas você poderia ter implementado essa funcionalidade?

Agora é a melhor hora de respirar mais tecnologia!



Se você está gostando dessa apostila, certamente vai aproveitar os **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum. Programação, Mobile, Design, Infra, Front-End e Business! Ex-aluno da Caelum tem 15% de desconto, siga o link!

[Conheça a Alura Cursos Online.](#)

14.8 TRABALHANDO COM STAGING DA APLICAÇÃO

Umas das novidades presentes no JSF 2.0 é a feature de stage presente na implementação de referência o Mojarra.

Quando os desenvolvedores desejam validar se algum campo da tela foi preenchido ou não, uma das perguntas mais comuns em fóruns de discussão como o GUJ, são: *Mas o que aconteceu? Por que a validação não foi feita? ou Cadê a mensagem de erro de validação?*

E a primeira resposta geralmente sempre é a mesma: "Faltou a tag `<h:messages/>`". Esta é uma das questões que a funcionalidade de Project Stage nos auxilia.

Project Stage é uma feature que configuramos no arquivo `web.xml` da nossa aplicação. Podemos

configurá-lo ajustando o parâmetro através de uma enumeration que contém as seguintes constantes:

- Production
- Development
- UnitTest
- SystemTest
- Extension

```
<context-param>
    <param-name>javax.faces.PROJECT_STAGE</param-name>
    <param-value>Development</param-value>
</context-param>
```

Se não configurarmos explicitamente, vale o valor default que é em *Production*.

Se habilitarmos o Project Stage em modo *Development*, não precisamos nos preocupar em colocar a tag `<h:messages/>` para exibir os erros de validação, porque o JSF colocará automaticamente se identificar que estamos em modo de desenvolvimento.

O JSF faz grande parte do seu trabalho utilizando a linguagem JavaScript. Porém quando olhamos o código JavaScript gerado, ele não é legível e muito difícil de encontrar erros.

Se executarmos o sistema em modo *Development*, o código JavaScript é muito mais legível e nos permite a verificação de erros através de ferramentas como firebug.

14.9 EXERCÍCIOS: ALTERANDO O STAGE DA NOSSA APLICAÇÃO

1. Vamos alterar o stage da nossa aplicação para modo de desenvolvimento para observarmos os efeitos.

- Primeiro **altere** o arquivo `web.xml` adicionando as seguintes linhas de xml:

```
<context-param>
    <param-name>javax.faces.PROJECT_STAGE</param-name>
    <param-value>Development</param-value>
</context-param>
```

- Agora **remova** do arquivo `login.xhtml` a tag `<h:messages/>`
- Verifique se há alguma validação no input, basta `required="true"`
- Restart o Glassfish e tente logar com um usuário e/ou senha inválido(s). Observe que a mensagem de erro do login, continua aparecendo. Graças ao uso do Project Stage em modo *Development*
- Abra o código fonte da sua página e observe como o código JavaScript do arquivo `jsf.js` é mais legível e mais limpo.

2. (Opcional) Adicione ao template, no cabeçalho, o nome do stage:

```
<h:outputText value="#{facesContext.application.projectStage}" />
```

14.10 VALIDAÇÃO DE MÚLTIPLOS CAMPOS COM JSF

Nativamente, a validação do JSF está associada a um único componente que geralmente está associado apenas a um atributo, consequentemente aquela validação só consegue validar o valor deste atributo.

Para validar múltiplos campos de um formulário é preciso implementar a funcionalidade manualmente.

O primeiro passo é criar uma classe `Validador` como já foi visto antes:

```
@FacesValidator("nomeEDescricao")
public class ValidadorNomeEDescricao implements Validator {

    @Override
    public void validate(FacesContext context,
        UIComponent component, Object value) throws ValidatorException {
        //implementacao
    }
}
```

No segundo passo precisamos buscar os componentes que gostaríamos de validar na árvore de componentes. No projeto importado já existe uma classe que sabe achar um componente recursivamente. Chamamos essa classe `ComponentResolver`. Vamos injetar essa classe no validator:

```
@Inject
private ComponentResolver resolver;
```

O resolver possui um método que sabe achar um componente pela id. O uso é simples, vamos recuperar o `ViewRoot` pelo `FacesContext` e procurar o componente. No exemplo abaixo procuramos um componente com a id `nome` na tela:

```
UIViewRoot arvore = context.getViewRoot();
UIComponent componente = resolver.findComponent(arvore, "nome");
```

Uma vez que temos o componente em mãos podemos recuperar o `submittedValue` do `UIInput`:

```
UIInput inputNome = (UIInput) componente;
String descricao = (String) inputNome.getSubmittedValue();
```

Este procedimento deve ser repetido para cada componente que queremos validar.

Para facilitar mais ainda o trabalho já criamos um método dentro do `ComponentResolver` que devolve o `submittedValue` dada a id do componente. Assim sobra apenas uma simples chamada:

```
String nome = resolver.getSubmittedValue("nome");
```

Veja o exemplo abaixo de uma validação que confere se o nome e a descrição de um produto não são iguais.

```
@FacesValidator("nomeEDescricao")
public class ValidadorNomeEDescricao implements Validator {
```

```

@Inject
private ComponentResolver resolver;

public void validate(FacesContext ctx, UIComponent component, Object value)
    throws ValidatorException {

    String nome = resolver.getSubmittedValue("nome");
    String descricao = resolver.getSubmittedValue("descricao");

    System.out.printf("Nome %s, Descricao %s %n", nome, descricao);

    if (nome != null && descricao != null && nome.equals(descricao)) {
        throw new ValidatorException(
            new FacesMessage("Nome e Descrição não podem ser iguais")
        );
    }
}
}

```

Para ativar este validador ainda temos que adicionar o componente `f:validator` ao formulário que estamos validando. O problema é que o `f:validator` sempre deve ser associado ao *input*. Podemos usar algum *input* existente do formulário ou criar um novo *fake* com o único propósito de chamar o validador. No exemplo abaixo usamos um `h:inputHidden` sem binding nenhum, só para declarar o `f:validator`:

```

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">

    <h:form>
        ...
        <h:inputHidden>
            <f:validator validatorId="nomeEDescricao" />
        </h:inputHidden>

        <p:inputText id="nome" value="#{produtoBean.produto.nome}" >
        ...
        </p:inputText>

        <p:inputTextarea id="descricao"
            value="#{produtoBean.produto.descricao}" />
        ...
    </h:form>
</html>

```

Injeção do EntityManager

Mesmo um Validador sendo um componente do JSF, podemos injetar qualquer coisa nele, até mesmo um `EntityManager` caso a validação necessite de informações do banco:

```

@FacesValidator("nomeJaExistente")
public class NomeJaExistenteValidator implements Validator, Serializable {

    @Inject

```

```

private EntityManager manager;

@Override
public void validate(FacesContext ctx,
    UIComponent component, Object value)
    throws ValidatorException {

    Query q = manager.createQuery("select count(p)
        from Produto p where p.nome like :nome");
    q.setParameter("nome", String.valueOf(value));
    Long count = (Long) q.getSingleResult();

    if (count != 0){
        throw new ValidatorException(
            new FacesMessage("Nome de Produto já Existente")
        );
    }
}
}

```

Editora Casa do Código com livros de uma forma diferente



Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.

Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.](#)

14.11 EXERCÍCIOS OPCIONAIS: VALIDAÇÃO DE MÚLTIPLOS CAMPOS

1. **Crie** a classe de validação que verifica que o nome e a descrição de um produto não sejam iguais. Você deve usar a interface padrão de validação do JSF, a `Validator` mas injetar a classe auxiliar `ComponentResolver`. Com essa classe fica fácil de recuperar algum valor pela id do componente:

```

@FacesValidator("nomeEDescricao")
public class ValidadorNomeEDescricao implements Validator {

    @Inject
    private ComponentResolver resolver;

    @Override
    public void validate(FacesContext ctx, UIComponent component, Object value)
        throws ValidatorException {

        String nome = resolver.getSubmittedValue("nome");
        String descricao = resolver.getSubmittedValue("descricao");
    }
}

```

```

        System.out.printf("Nome %s, Descricao %s \n", nome, descricao);

        if (nome != null && descricao != null && nome.equals(descricao)) {
            throw new ValidatorException(
                new FacesMessage("Nome e Descrição não podem ser iguais")
            );
        }
    }
}

```

É importante ter certeza que os ids dos componentes batem com os usados no `produto.xhtml`.

- Adicione a tag de validação **dentro do formulário** de produtos no arquivo `produto.xhtml`. Para declarar o validador crie um novo input logo *acima do botão de submit*:

```

<h:inputHidden>
    <f:validator validatorId="nomeEDescricao" />
</h:inputHidden>

```

Verifique que o nome do validador é o mesmo usado na anotação `@FacesValidator`

- Reinic peace o Glassfish e teste sua tela de produtos colocando o mesmo nome e descrição para um produto.
- (opcional) Crie um validador que confira se o nome do produto já existe no banco de dados.

14.12 PARA SABER MAIS: CONVERSORES CUSTOMIZADOS

Existem alguns momentos que os valores digitados pelos usuários nas telas, não são os valores que desejamos receber em nossos Managed Beans. Em nossa aplicação, o CNPJ digitado pelo usuário na tela de cadastro de Nota Fiscal chega ao objeto `NotaFiscal`, como uma `String` com a máscara do `<p:inputMask>` do Prime Faces. Porém, pode ser que desejamos trabalhar com o valor sem a formatação.

O que precisamos fazer é aproveitarmos a fase de conversão do JSF, que é invocada antes da fase de ajuste dos valores em nossos beans, e convertermos o valor para que o mesmo fique sem a máscara.

Criar conversores no JSF é uma tarefa simples, basta criarmos uma classe que implemente a interface `javax.faces.convert.Converter` e implementarmos os métodos presentes na interface.

O primeiro método é chamado de `getAsObject` é invocado pelo JSF quando algum valor recebido na requisição precisa ser ajustado dentro de um bean, ou seja, o valor que foi digitado pelo usuário e trafegado dentro do `request` como `String` será convertido neste método.

O segundo método é chamado de `getAsString` que faz o trabalho contrário ao método anterior. O objetivo dele é fazer uma conversão de um determinado valor presente no bean, para um `String` que será exibida para o usuário.

Em nosso cenário o método `getAsObject` fará o papel de remover a máscara vindia da tela, e o

método `getAsString` fará o papel de recolocar a máscara para que o dado seja exibido para o usuário de uma maneira mais legível.

O único detalhe de configuração que precisamos fazer é anotar a classe de conversor com `@FacesConverter` ajustando o atributo `value` com o nome que desejamos nos referenciar posteriormente para aplicarmos a conversão.

```
@FacesConverter("cnpj")
public class ConversorCNPJ implements Converter {

    public Object getAsObject(FacesContext context,
        UIComponent component, String valor) {
        //Aqui vai o valor não formatado
        return null;
    }

    public String getAsString(FacesContext context,
        UIComponent component, Object valor) {
        //Aqui vai o valor formatado
        return null;
    }
}
```

Precisamos implementar o conteúdo dos dois métodos seguindo o que definimos anteriormente. Visando seguir a premissa de não reinventar a roda, vamos utilizar algo pronto que faça o trabalho de formatar e remover a formatação. Existe um projeto open-source mantido pelos desenvolvedores da Caelum, chamado **Caelum Stella**, uma poderosa biblioteca para desenvolvedores brasileiros. Ele é capaz de fazer validações, conversões e formatações.

Para utilizá-lo, basta incluirmos dentro da pasta WEB-INF/lib do nosso projeto o stella-core.jar presente na pasta **Caelum/26/jars-stella**.

A versão mais atual do Stella se encontra no site do projeto:

<http://stella.caelum.com.br/>

Podemos utilizar uma classe do Stella chamada `CNPJFormatter`. Ela possui dois métodos que recebem uma String e cada um deles é responsável ou por incluir a formatação ou removê-la.

Finalizando nossa ideia com o Stella:

```
@FacesConverter("cnpj")
public class ConversorCNPJ implements Converter {

    public Object getAsObject(FacesContext context,
        UIComponent component, String valor) {
        if(valor == null || valor.trim().isEmpty()) {
            return valor;
        }

        return new CNPJFormatter().unformat(valor);
    }
}
```

```

public String getAsString(FacesContext context,
    UIComponent component, Object valor) {

    if(valor == null){
        return null;
    }

    if(valor.toString().trim().isEmpty()) {
        return "";
    }

    return new CNPJFormatter().format(valor.toString());
}
}

```

Pronto! Nosso conversor já faz o trabalho necessário, basta habilitarmos a conversão no <p:inputMask> ajustando o atributo converter com o valor que colocamos na anotação @FacesConverter da seguinte maneira:

```
<p:inputMask value="#{notaFiscalBean.notaFiscal.cnpj}" mask="99.999.999/9999-99"
    converter="cnpj" converterMessage="CNPJ inválido"/>
```

Porém, temos um problema! Lembre-se que estamos validando o cnpj que chega no bean NotaFiscal, e, de acordo com as fases do JSF, a conversão é efetuado antes da fase de validação. Como a máscara é removida na conversão, na fase de validação o valor nunca estará consistente. Isto prova mais uma vez que as fases do JSF realmente são muito bem definidas.

Como resolver este problema? Neste caso, removendo a validação que fizemos anteriormente, retirando o atributo @Pattern do atributo cnpj da classe NotaFiscal .

Fazendo isto, o valor será gravado no banco de dados sem a máscara, porém para o usuário é mais interessante visualizar o valor com a máscara. Isto já é feito no método getAsString do conversor e para aplicá-lo basta adicioná-lo na exibição do CNPJ.

Isto é feito na tabela que lista as notas fiscais e que foi feita anteriormente. Para adicionar o conversor basta especificá-lo no <h:outputText> que imprime o CNPJ.

```
<h:outputText converter="cnpj" value="#{notaFiscal.cnpj}" />
```

14.13 EXERCÍCIOS OPCIONAL: CONVERSORES PERSONALIZADOS

1. Copie os arquivos stella-core-2.x.jar e caelum-stella-bean-validation-2.x.jar e caelum-stella-faces-2.x.jar encontrados na pasta **Caelum/26/jars-stella/** para dentro da pasta WEB-INF/lib do projeto.
2. Crie uma classe de conversão no pacote br.com.caelum.notasfiscais.conversores , chame-a de ConversorCNPJ e implemente a interface javax.faces.convert.Converter .
3. Implemente os métodos da interface fazendo a formatação no método getAsString e a remoção da formatação no método getAsObject utilizando o Stella.
4. Aplique a conversão no <p:inputMask> e também no <h:outputText> da listagem de notas

fiscais feita no exercício anterior. Aproveite e coloque o atributo `required="true"` no `<p:inputMask>`.

5. Remova a anotação `@Pattern` da classe `NotaFiscal`, já que agora o CNPJ não será mais enviado com a formatação. Também usa a anotação `@CNPJ` da Stella (**cuidado** importe a anotação do pacote `'br.com.caelum.stella.bean.validation%%'`).

Já conhece os cursos online Alura?



A Alura oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Programação, Front-end, Mobile, Design & UX, Infra e Business, com um plano que dá acesso a todos os cursos. Ex aluno da Caelum tem 15% de desconto neste link!

[Conheça os cursos online Alura.](#)

APÊNDICE - INTERNACIONALIZAÇÃO: SUA APLICAÇÃO EM VÁRIAS LÍNGUAS

"A língua é um sabre que pode trespassar um corpo" -- Yonsan

15.1 FORMAS DE INTERNACIONALIZAR A APLICAÇÃO

Muitas aplicações desenvolvidas atualmente possuem como requisito ficar disponível para usuários de diferentes que compreendem diferentes idiomas. Isso é um requisito bastante comum, por exemplo, em multinacionais onde um mesmo software é utilizado em diversos países e portanto precisa estar acessível em diferentes línguas.

Esse trabalho de permitir que o suporte para várias línguas na aplicação é um processo chamado *internacionalização (i18n)*, no entanto, como ele pode ser conseguido?

Existem diferentes maneiras de internacionalizar uma aplicação cada uma com suas vantagens e desvantagens. Uma primeira solução simples, porém inocente para uma aplicação web seria criar duas versões da mesma aplicação, cada uma com as páginas em uma língua e disponibilizar essa aplicação web em domínios diferentes. Por exemplo, a aplicação em português ficaria acessível em:

`http://pt.aplicacao.com` enquanto a aplicação em espanhol ficaria acessível em: `http://es.aplicacao.com`. O principal ponto negativo dessa abordagem é a manutenção da aplicação, pois estamos duplicando todas as páginas e funcionalidades. Quando uma nova língua deve ser suportada precisamos replicar as páginas. Dessa forma, cria-se mais pontos de manutenção em progressão aritmética.

Uma outra abordagem que por vezes é adotada é a criação de tabelas no banco de dados para armazenar as mensagens em diferentes línguas. Essa abordagem possui como ponto positivo a flexibilidade de poder contar com uma interface visual da própria aplicação para editar as mensagens nas diferentes línguas, no entanto torna a aplicação muito mais pesada, tendo em vista que consultas ao banco de dados devem ser disparadas para buscar as mensagens necessárias para a tela.

Saber inglês é muito importante em TI

galandra

O **Galandra** auxilia a prática de inglês através de *flash cards* e *spaced repetition learning*. Conheça e aproveite os preços especiais.

[Pratique seu inglês no Galandra.](#)

15.2 UTILIZANDO O JSF PARA INTERNACIONALIZAR

A grande maioria dos frameworks MVC de diferentes plataformas (Java, Ruby, .NET etc) atualmente possuem suporte nativo a internacionalização. O JSF permite isso através da criação de *message bundles*, que são arquivos que armazenam as mensagens que a aplicação utilizará para formar suas telas.

Esses arquivos devem possuir a extensão `.properties` e possuem uma estrutura de chave/valor onde identificamos cada mensagem da aplicação através de um identificador único e lhe atribuímos um valor adequado. Dessa forma, para nossa aplicação poderíamos criar um arquivo de mensagens chamado `Resources.properties` que deve estar no classpath da nossa aplicação. Para mantermos os nossos arquivos de mensagens organizados, podemos criar um pacote chamado `br.com.caelum.notasfiscais.mensagens` e guardá-los dentro desse diretório.

Para que o JSF comprehenda esse arquivo de mensagens, precisamos declará-lo no `faces-config.xml`, para isso, precisamos adicionar dentro da Tag `application` uma nova Tag chamada `resource-bundle`. Dentro dessa Tag devemos indicar onde se localiza o nosso arquivo de mensagens através de uma nova Tag chamada `base-name` contendo o caminho com os pacotes para o arquivo `.properties` que nesse caso será o valor `br.com.caelum.notasfiscais.mensagens.Resources`. Por fim, precisamos indicar através de qual variável teremos acesso às mensagens dentro dos nossas páginas `xhtml`. Fazemos isso através da Tag `var`. Dessa forma, o XML ficara semelhante a:

```
<resource-bundle>
    <base-name>br.com.caelum.notasfiscais.mensagens.Resources</base-name>
    <var>bundle</var>
</resource-bundle>
```

O próximo passo é adicionarmos as mensagens dentro do arquivo `.properties`.

Podemos tomar como exemplo a página `login.xhtml` onde temos diversas mensagens em português que estão fixas dentro de nossa página. Precisamos movê-las da página para o arquivo `.properties`. Para isso, podemos adicionar a mensagem `Login no Sistema` dentro do arquivo precedido de uma chave, por exemplo `label_loginSistema`. Dessa forma, teremos no `Resources.properties`:

```
label_loginSistema=Login no Sistema
```

O próximo passo utilizarmos essa chave que atribuímos a mensagens no `login.xhtml` através da *Expression Language*. Para isso, podemos utilizar a variável que declaramos no `faces-config.xml` e recuperar a chave do arquivo de propriedades, como em `#{{bundle.label_loginSistema}}`. Poderíamos seguir esse passo para todas as outras mensagens da nossa aplicação. Dessa forma teríamos todas elas isoladas em um único ponto e de fácil manutenção. No entanto, como fazemos para termos suporte à diferentes idiomas?

O processo de traduzir a aplicação para diferentes idiomas é conhecido também como *localização* (*l10n*). Podemos localizar uma aplicação JSF através da criação de arquivos `.properties` para as línguas suportadas. Cada arquivo deve possuir como sufixo de seu nome o código da língua a que o arquivo se refere. Dessa forma, o arquivo para as mensagens em português deve estar em um arquivo chamado `Resources_pt.properties` enquanto as mensagens para o idioma russo deveriam estar em `Resources_ru.properties`.

Além disso, o JSF precisa conhecer previamente os idiomas suportados pela nossa aplicação. Para isso, devemos declará-los no `faces-config.xml` junto de um idioma padrão. Os idiomas suportados são declarados na Tag `locale-config` que deve aparecer dentro da `<application>`.

Dentro de `locale-config` indicamos quais os idiomas suportados através de entradas `<supported-locales>` e para o idioma padrão `<default-locale>`, dessa forma poderíamos ter a seguinte configuração no XML:

```
<locale-config>
    <default-locale>pt</default-locale>
    <supported-locale>fr</supported-locale> <!-- suporte para francês -->
    <supported-locale>en</supported-locale> <!-- suporte para inglês -->
</locale-config>
```

Para o exemplo acima, portanto deveríamos ter 3 arquivos de mensagens: `Resources_pt.properties`, `Resources_en.properties` e `Resources_fr.properties`.

O `default-locale` é importante, pois quando é indicado um idioma não suportado ele é o utilizado, servindo como uma espécie de *fall-back*. Mas como o JSF sabe se queremos mensagens em português e não em francês para visualizarmos as páginas?

Sempre que uma requisição é feita através do navegador, é enviado um cabeçalho chamado `Accept-Language` onde é determinado quais os idiomas que o cliente deseja receber a resposta. No caso de receber em francês, por exemplo é passado o valor `fr`. Na maioria dos navegadores é possível dizer quais são os idiomas preferidos para receber as respostas.

15.3 EXERCÍCIOS: COMEÇANDO A INTERNACIONALIZAÇÃO E LOCALIZAÇÃO DA APLICAÇÃO

1. Vamos começar a internacionalizar e localizar a nossa aplicação para diversos idiomas. A princípio assumiremos os idiomas português, francês e inglês, no entanto, sinta-se a vontade para suportar outros idiomas.
 - Primeiramente, precisamos declarar no `faces-config.xml` da nossa aplicação o local onde armazenaremos os arquivos de mensagens e quais serão os idiomas suportados:

```
<application>
    <resource-bundle>
        <base-name>
            br.com.caelum.notasfiscais.mensagens.Resources
        </base-name>
        <var>bundle</var>
    </resource-bundle>
    <locale-config>
        <default-locale>pt</default-locale>
        <supported-locale>fr</supported-locale>
        <supported-locale>en</supported-locale>
    </locale-config>
</application>
```

 - Precisamos criar os arquivos para as mensagens. Para isso, como definimos no XML, crie o pacote `br.com.caelum.notasfiscais.mensagens` e dentro dele os arquivos `Resources.properties` para os 3 idiomas que serão suportados pela nossa aplicação.
2. Agora precisamos utilizar as mensagens em nossa aplicação. Para isso, comece pela página `login.xhtml` por ser a mais simples. Remova as mensagens que estão estáticas e utilize a *Expression Language* para referenciar as mensagens do arquivo `.properties`.
3. Se necessário faça ajustes nos arquivos dos idiomas francês e inglês para que as chaves se adequem às chaves escolhidas por você.
4. Teste sua aplicação alterando o idioma do navegador. Peça ajuda para seu instrutor.

15.4 ALTERANDO O IDIOMA MANUALMENTE

Selecionar o idioma através de configurações do navegador pode não ser trivial para alguns usuários menos experientes. Para isso, as aplicações geralmente permitem que o usuário ele seja alterado através de alguma funcionalidade. Normalmente esse funcionalidade é disponibilizada através de links, que quando clicados mudam internamente o *Locale* da aplicação.

Vamos alterar nossa aplicação para que ela tenha esses links. Para isso, utilizaremos imagens com pequenas bandeiras. Para adicionar esses links em nossa aplicação, poderíamos fazer:

```
<h:form styleClass="i18n">
    <h:commandLink value="EN" />
    <h:commandLink value="PT" />
</h:form>
```

O código acima poderia estar dentro do `_template.xhtml` para ser reaproveitado em todas as páginas do nosso sistema.

O próximo passo é fazermos com que ao clicar nos links, o locale utilizado seja o correspondente. Para isso, podemos criar um novo Managed Bean para tratarmos a troca de idiomas. Vamos chamá-lo de `InternacionalizacaoBean`. Esse Managed Bean deve ter um método que efetuará a troca do locale para cada idioma:

```
@Named  
public class InternacionalizacaoBean {  
    private @Inject FacesContext context;  
  
    public void mudaPara(String lingua) {  
        // adicionaremos código aqui...  
    }  
}
```

Esse método será invocado pelos respectivos links que criamos na tela. Dessa forma, precisamos adicionar o atributo `action` com a *Expression Language* apontando para os métodos o nosso novo Managed Bean e passando a língua correta:

```
<h:form styleClass="i18n">  
    <h:commandLink action="#{internacionalizacaoBean.mudaPara('en')}"  
        value="EN" />  
    <h:commandLink action="#{internacionalizacaoBean.mudaPara('pt')}"  
        value="PT" />  
</h:form>
```

Para forçarmos a troca do *Locale* precisamos conseguir a instância atual do `FacesContext`. Para isso, podemos fazer `FacesContext.getCurrentInstance()`. Em seguida, no `FacesContext` recuperado, precisamos recuperar também o `ViewRoot` através do método `getViewRoot` e invocarmos o método `setLocale` passando como parâmetro um objeto do tipo `Locale` para mudar o locale da tela atual. O locale de aplicação inteira é alterado pelo método `facesContext.getApplication().setDefaultLocale(locale)`:

```
public void mudaPara(String lingua) {  
    Locale locale = new Locale(lingua);  
    //muda a locale para a tela atual  
    context.getViewRoot().setLocale(locale);  
  
    //muda a locale para a aplicação inteira  
    context.getApplication().setDefaultLocale(locale);  
}
```

Seus livros de tecnologia parecem do século passado?



Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**. Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

[Casa do Código, Livros de Tecnologia.](#)

15.5 EXERCÍCIOS: DEFININDO O IDIOMA ATRAVÉS DE LINKS

1. Crie o managed bean `InternacionalizacaoBean` que gerenciará a troca do locale escolhido pelo usuário:

```
@Named  
public class InternacionalizacaoBean {  
  
    private @Inject FacesContext context;  
  
    public void mudaPara(String lingua) {  
        Locale locale = new Locale(lingua);  
        context.getViewRoot().setLocale(locale);  
        context.getApplication().setDefaultLocale(locale);  
    }  
}
```

2. Crie os links que invocarão o método do Managed Bean que criamos. Para isso, adicione ao arquivo `_template.xhtml` os links com as respectivas línguas.
3. Reinicie o servidor e acesse a aplicação para testar a internacionalização.

15.6 INTERNACIONALIZANDO MENSAGENS DE ERRO DO BEAN VALIDATION

Até o momento, grande parte da nossa aplicação está internacionalizada, isso não acontece com as mensagens de validação dos nossos modelos. Quando não preenchemos o nome do produto, a mensagem não segue a regra da nossa aplicação com relação ao idioma. No entanto, para alterarmos as mensagens de validação é simples. Basta criarmos um arquivo chamado `ValidationMessages.properties` para cada idioma na raiz do nosso classpath.

Devido às mensagens de erro serem referentes à um framework específico (no caso o Hibernate Validator), as chaves das mensagens para fazermos a internacionalização já estão pré-definidas. Para isso, podemos consultar o arquivo internacionalizado que já vem com o próprio Hibernate Validator,

para conseguirmos as chaves e internacionalizar.

Fazendo isso, percebemos que para alterarmos a mensagem para a anotação `@NotEmpty` basta adicionarmos a seguinte entrada no `ValidationMessages_pt.properties`:

```
org.hibernate.validator.constraints.NotEmpty.message=Não podia estar vazio
```

15.7 EXERCÍCIOS: INTERNACIONALIZANDO MENSAGENS DE ERRO

1. Crie os arquivos de mensagens para internacionalizar as validações dentro do diretório `src` do seu projeto. Para isso, baseie-se no arquivo que já vem pronto e está no `jar` do Hibernate Validator dentro do pacote `org.hibernate.validator`.
2. Após alterar as mensagens, reinicie a aplicação e faça acontecer erros de validação. Veja se as mensagens foram localizadas corretamente.

Agora é a melhor hora de respirar mais tecnologia!



Se você está gostando dessa apostila, certamente vai aproveitar os **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum. Programação, Mobile, Design, Infra, Front-End e Business! Ex-aluno da Caelum tem 15% de desconto, siga o link!

[Conheça a Alura Cursos Online.](#)

15.8 INTERNACIONALIZANDO AS MENSAGENS DENTRO DOS MANAGED BEANS

A aplicação está internacionalizada de quase todas as formas possíveis até o momento, mas ainda não internacionalizamos as mensagens que disparamos de dentro dos nossos Managed Beans, como é o caso de quando um produto novo é salvo. Precisamos que ao criar o `FacesMessage` para a mensagem ela seja buscada de um arquivo de mensagens e não esteja fixa dentro do Managed Bean, o que dificulta a manutenção.

Para conseguirmos acesso de forma programática às mensagens, precisamos de uma instância do classe `ResourceBundle` que podemos adquirir através de um método estático da própria classe chamado `getBundle`, onde devemos passar como parâmetro o nome do bundle (o mesmo que configuramos no `faces-config.xml`) e o `Locale` atual que podemos recuperar através do `FacesContext`. Dessa forma teremos:

```

FacesContext facesContext = FacesContext.getCurrentInstance();
String bundleName = "br.com.caelum.notasfiscais.mensagens.Resources";
ResourceBundle bundle = ResourceBundle.getBundle(bundleName,
    facesContext.getViewRoot().getLocale());

```

O próximo passo é a partir da variável `bundle` recuperarmos a mensagem que queremos utilizar através de sua respectiva chave. Fazemos isso invocando o método `getString` passando a chave como parâmetro:

```
String mensagem = bundle.getString(key);
```

Por fim, precisamos passar essa variável mensagem para o `FacesMessage` que queremos disponibilizar para a tela através do `FacesContext`:

```
facesContext.addMessage(null, new FacesMessage(mensagem));
```

Como esse código é muito frequente em aplicações que utilizam JSF, normalmente optamos por deixá-lo em uma classe utilitária, que podemos chamar de `JsfUtil` com um método para encapsular toda essa lógica. Com isso essa classe terá o seguinte código:

```

public class JsfUtil {
    public void addMessage(String key) {
        FacesContext facesContext = FacesContext.getCurrentInstance();
        String bundleName = "br.com.caelum.notasfiscais.mensagens.Resources";
        ResourceBundle bundle = ResourceBundle.getBundle(bundleName,
            facesContext.getViewRoot().getLocale());
        String mensagem = bundle.getString(key);

        facesContext.addMessage(null, new FacesMessage(mensagem));
    }
}

```

Para melhorar esse código, podemos aproveitar a nossa infraestrutura e deixar a classe `JsfUtil` gerenciada pelo CDI. Além disso, podemos injetar o `FacesContext` com Seam:

```

@Named @RequestScoped
public class JsfUtil {

    public void addMessage(String key) {

        FacesContext facesContext = FacesContext.getCurrentContext();

        String bundleName = "br.com.caelum.notasfiscais.mensagens.Resources";
        ResourceBundle bundle = ResourceBundle.getBundle(bundleName,
            facesContext.getViewRoot().getLocale());
        String mensagem = bundle.getString(key);

        facesContext.addMessage(null, new FacesMessage(mensagem));
    }
}

```

Agora em todos os lugares da nossa aplicação que precisarmos utilizar uma mensagem internacionalizada basta injetar o `JsfUtil` e chamar o método `addMessage` como abaixo:

```

@Inject JsfUtil JsfUtil;
//no método

```

```
jsfUtil.addMessage("msg_erroLogin");
```

15.9 EXERCÍCIOS: INTERNACIONALIZANDO AS MENSAGENS DO MANAGED BEANS

1. Vamos internacionalizar as mensagens que utilizamos dentro de nossos Managed Beans. Crie a classe `JsfUtil` em um pacote que achar conveniente (**pense sobre o nome do pacote a ser escolhido**).
2. Na classe `JsfUtil` crie um método para fazer a adição da mensagem. Esse método deve recuperar um objeto do tipo `ResourceBundle` para o *Locale* correto e partir do `ResourceBundle` recuperado consiga a mensagem através da chave correta que deve ser recebida como parâmetro do método.
Por fim, esse método deve adicionar uma nova `FacesMessage` ao `FacesContext`.
3. Injete o `JsfUtil` para gerar uma mensagem, por exemplo no `ProdutoBean`. Gere uma mensagem que confirma a gravação/alteração de um produto.

15.10 EXERCÍCIO OPCIONAL: INTERNACIONALIZANDO A APLICAÇÃO POR COMPLETO

1. Internacionalize todas as mensagens da sua aplicação, seja ela em Managed Beans, páginas ou validação pelo menos para o português.

Editora Casa do Código com livros de uma forma diferente



Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.

Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.](#)