

# Desenvolvimento web Java

## JSF + PrimeFaces

## e Hibernate

**Alex Fernando Egidio**  
Desenvolvedor Java Sênior  
[www.javaavancado.com](http://www.javaavancado.com)

## Sumário

O que é Java EE?.....	5
O protocolo HTTP.....	6
Desenvolvimento web com Java.....	8
Containers.....	9
Introdução ao JSF e Primefaces.....	10
Desenvolvimento desktop ou web?.....	11
O desenvolvimento Web e o protocolo HTTP.....	12
Mesclando desenvolvimento Desktop e Web.....	13
Características do JSF.....	13
Guarda o estado dos componentes.....	14
Separa as camadas.....	14
Especificação: várias implementações.....	14
Primeiros passos com JSF.....	15
Configuração do controlador do JSF.....	17
Faces-config: o arquivo de configuração do mundo JSF.....	18
O que é ManagedBean?.....	18
Principais componentes.....	19
O que é o MVC Design Pattern?.....	20
Ciclo de vida do JSF.....	20
Fase 1: Restore View (Restauração da visão).....	22
Fase 2: Apply Request Values (Aplicar valores da requisição).....	22
Fase 3: Process Validation (Processar as validações).....	22
Fase 4: Update Model Values (Atualizar valores de modelo).....	22
Fase 5: Invoke Application (Invocar aplicação).....	23
Fase 6: Render Response (Renderizar a resposta).....	23
Visão geral.....	24
Anotações e Escopo.....	24
Estrutura de página JSF.....	25
Configurando JSF com Maven.....	26
Ativando o JSF em nosso projeto.....	28
Configurando o Persistence.xml.....	29
Classe Java HibernateUtil.....	29
Mapeando nossa entidade Usuário.....	30
Estrutura padrão de um ManagedBean.....	32
PanelGrid em JSF e EL.....	33
JSF – DataTable.....	34
Mensagens com FacesMessage.....	34
Exibindo mensagens após redirecionamento.....	35
CommandButton JSF.....	36
Quando usar Action ou ActionListener com JSF.....	37
O Action.....	37
O ActionListener.....	38
O setPropertyActionListener.....	39
Utilizando AJAX com JSF de maneira eficiente.....	39
Visão geral do Ajax.....	40
Utilizando Ajax com JSF.....	41
Usando o FilterOpenSessionInView.....	42
Declarando Filter no Web.xml.....	43

Declarando o JPA/Hibernate no Filter.....	44
Tags Básicas.....	45
Parâmetros com JSF.....	45
Conhecendo show case JSF e PrimeFaces.....	46
Datatable se torna super fácil com PrimeFaces.....	50
Gráficos com PrimeFaces.....	53
Capturando erros com ExceptionHandler.....	54
ExceptionHandler com Ajax.....	55
ExceptionHandler sem Ajax.....	57
Confirm Dialog.....	58
Confirm DialogGMap – Basic.....	59
Barcode.....	60
Chamadas de Ajax periódicas.....	60
Growl do PrimeFaces.....	61
Download do manual do PrimeFaces.....	62
Referências.....	63

[CLIQUE AQUI – Conheça o curso Formação Java Web Full-Stack](#)



Alex Fernando Egidio

Autor, criador e fundador do Java Avançado Cursos TI. Atua no mercado como desenvolvedor e engenheiro de sistemas em Java é apaixonado por desenvolvimento web em Java e pela “mágica” que seus frameworks trazem para o dia a dia de nós desenvolvedores.

Com mais de 10 anos de experiência ajuda programadores do Brasil todo a se tornarem profissionais.

[CLIQUE AQUI – Conheça o curso Formação Java Web Full-Stack](#)

## O que é Java EE?

A Java EE (Java Platform, Enterprise Edition) é uma plataforma padrão para desenvolver aplicações Java de grande porte e/ou para a internet, que inclui bibliotecas e funcionalidades para implementar software Java distribuído, baseado em componentes modulares que executam em servidores de aplicações e que suportam escalabilidade, segurança, integridade e outros requisitos de aplicações corporativas ou de grande porte.

A plataforma Java EE possui uma série de especificações (tecnologias) com objetivos distintos, por isso é considerada uma plataforma guarda-chuva. Entre as especificações da Java EE, as mais conhecidas são:

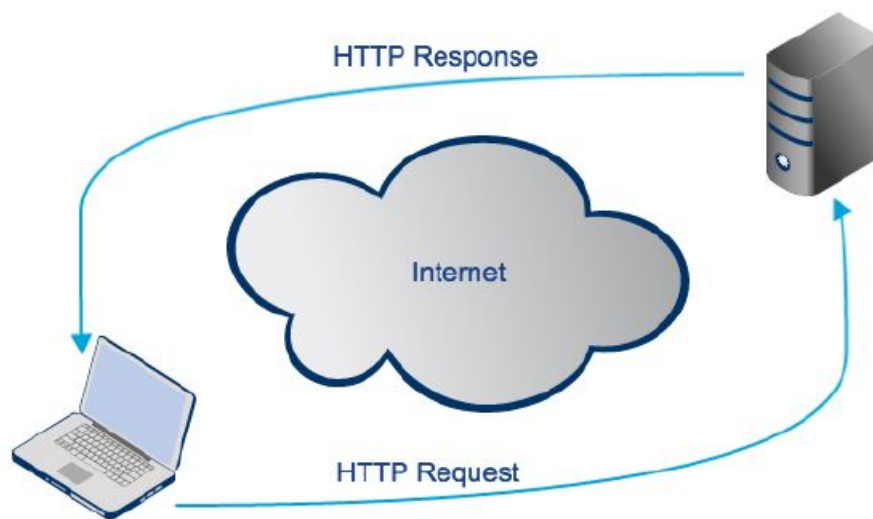
- **Servlets:** são componentes Java executados no servidor para gerar conteúdo dinâmico para a web, como HTML e XML.
- **JSP (JavaServer Pages):** uma especialização de Servlets que permite que aplicações web desenvolvidas em Java sejam mais fáceis de manter. É similar às tecnologias como ASP e PHP, porém mais robusta por ter todas as facilidades da plataforma Java.
- **JSF (JavaServer Faces):** é um framework web baseado em Java que tem como objetivo simplificar o desenvolvimento de interfaces (telas) de sistemas para a web, através de um modelo de componentes reutilizáveis. A proposta é que os sistemas sejam desenvolvidos com a mesma facilidade e produtividade que se desenvolve sistemas desktop (até mesmo com ferramentas que suportam clicar-e-arrastar componentes).
- **JPA (Java Persistence API):** é uma API padrão do Java para persistência de dados, que usa um conceito de mapeamento objeto-relacional. Essa tecnologia traz alta produtividade para o desenvolvimento de sistemas que necessitam de integração com banco de dados. Só para citar, essa API possibilita que você desenvolva aplicações usando banco de dados sem precisar escrever uma linha sequer de SQL.
- **EJB (Enterprise Java Beans):** são componentes que executam em servidores de aplicação e possuem como principais objetivos, fornecer facilidade e produtividade no desenvolvimento de componentes distribuídos, transacionados, seguros e portáteis.

## O protocolo HTTP

O protocolo HTTP é utilizado na navegação de páginas da Internet. Quando você abre uma janela de um browser, acessa uma página Web e navega em seus links, você está, na verdade, utilizando esse protocolo para visualizar, em sua máquina, o conteúdo que está armazenado e/ou é processado em servidores remotos.

O HTTP é um protocolo stateless de comunicação cliente-servidor: o cliente envia uma requisição para o servidor, que processa a requisição e devolve uma resposta para o cliente, sendo que, a princípio, nenhuma informação é mantida no servidor em relação às requisições previamente recebidas.

Assim, quando digitamos o endereço de uma página em um browser, estamos gerando uma requisição a um servidor, que irá, por sua vez, devolver para o browser o conteúdo da página HTML requisitada.



A requisição enviada por um cliente deve conter, basicamente, um comando (também chamado de método), o endereço de um recurso no servidor (também chamado de path) e uma informação sobre a versão do protocolo HTTP sendo utilizado.

Supondo, por exemplo, que queremos buscar o conteúdo do endereço `http://www.uol.com.br/index.html`. Utilizemos o método GET, o path `/index.html` e a versão 1.1 do protocolo HTTP. Temos a seguinte requisição enviada:

```
GET /index.html HTTP/1.1
```

```
Host: www.uol.com.br
```

Existem diversos métodos HTTP que podem ser especificados em requisições, sendo os mais comuns o método GET, normalmente utilizado para obter o conteúdo de um arquivo no servidor, e o método POST, utilizado para enviar dados de formulários HTML ao servidor.

Uma requisição pode conter parâmetros adicionais, chamados headers. Alguns headers comuns são, por exemplo, Host, User-Agent e Accept.

Uma vez processada a requisição, o servidor, por sua vez, manda uma resposta para o cliente, sendo que essa resposta também tem um formato pré-determinado: a primeira linha contém informações sobre a versão do protocolo, um código de status da resposta e uma mensagem associada a esse status.

Em seguida, são enviados os headers da resposta, e finalmente, é enviado o conteúdo da resposta. Veja um exemplo simples de resposta HTTP:

```
HTTP/1.1 200 OK
```

```
Date: Thu, 26 Sep 2013 15:17:12 GMT
```

```
Server: Apache/2.2.15 (CentOS)
```

```
Content-Type: text/html; charset=utf-8
```

```
<html>
```

```
  <body>
```

```
  </body>
```

```
</html>
```

No exemplo anterior, o código de status 200 indica que houve sucesso no atendimento da requisição enviada pelo cliente, e os headers indicam a data e hora do servidor, o servidor usado, tipo do conteúdo e, por fim, temos o código-fonte da página HTML.

Outros códigos de status bastante comuns são o 404, que indica que o recurso não foi localizado no servidor e o código 500, que indica que houve erro no processamento da requisição enviada.

## **Desenvolvimento web com Java**

Com o avanço da tecnologia sobre redes de computadores e com o crescimento da internet, as páginas web estão se tornando cada vez mais atraentes e cheias de recursos que aumentam a interatividade com o usuário.

Quando falamos em aplicações web, estamos nos referindo a sistemas ou sites onde grande parte da programação fica hospedada em servidores na internet, e o usuário (cliente) normalmente não precisa ter nada instalado em sua máquina para utilizá-las, além de um navegador (browser).

O acesso às páginas desses sistemas é feita utilizando o modelo chamado de request e response, ou seja, o cliente solicita que alguma ação seja realizada (request) e o servidor a realiza e responde para o cliente (response).

Na plataforma Java, esse modelo foi implementado através da API de Servlets. Um Servlet estende a funcionalidade de um servidor web para servir páginas dinâmicas aos navegadores, utilizando o protocolo HTTP.

No mundo Java, os servidores web são chamados de Servlet Container, pois implementam a especificação de Servlet. O servidor converte a requisição em um objeto do tipo `HttpServletRequest`. Este objeto é então passado aos componentes web, que podem executar qualquer código Java para que possa ser gerado um conteúdo dinâmico.

Em seguida, o componente web devolve um objeto `HttpServletResponse`, que representa a resposta ao cliente. Este objeto é utilizado para que o conteúdo gerado seja enviado ao navegador do usuário.



## Containers

Containers são interfaces entre componentes e funcionalidades de baixo nível específicas de uma plataforma. Para uma aplicação web desenvolvida em Java ou um componente corporativo ser executado, eles precisam ser implantados em um container.

Os containers também são chamados de servidores de objetos, ou servidores de aplicação, pois oferecem serviços de infra-estrutura para execução de componentes.

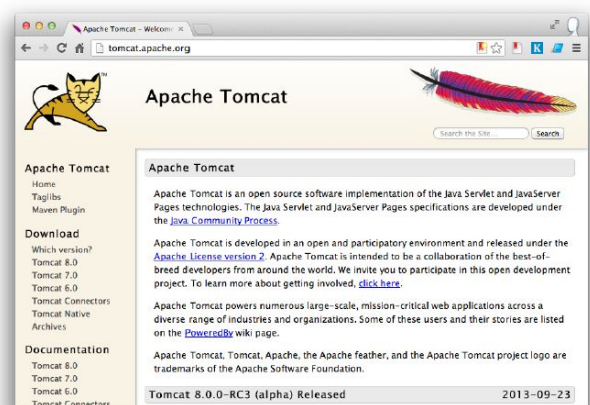
O EJB Container suporta Enterprise JavaBeans (EJB), que são componentes corporativos distribuídos. Os Servlets, JSP, páginas JSF e arquivos estáticos (HTML, CSS, imagens e etc) necessitam de um Web Container para ser executado.

Existem diversas organizações que desenvolvem containers Java EE, por exemplo: Oracle, IBM, Red Hat, Apache, etc. Apesar de tantas ofertas gratuitas, algumas empresas ainda vendem licenças de seus próprios servidores, pois oferecem suporte diferenciado ao cliente e normalmente implementam funcionalidades que os servidores gratuitos talvez não possuam.

Para testar nossos exemplos, usaremos o Apache Tomcat, pois é leve, gratuito e muito popular.

Como estes servidores são baseados nas especificações da tecnologia Java EE, teoricamente, você pode implantar os exemplos que desenvolveremos neste livro em qualquer container compatível com a Java EE.

O download do Apache Tomcat pode ser feito em <http://tomcat.apache.org>.



## Introdução ao JSF e Primefaces

Durante muitos anos, os usuários se habituaram com aplicações Desktop. Este tipo de aplicação é instalada no computador local e acessa diretamente um banco de dados ou gerenciador de arquivos.

As tecnologias típicas para criar uma aplicação Desktop são Delphi, VB (Visual Basic) ou, no mundo Java, Swing. Para o desenvolvedor, a aplicação Desktop é construída com uma série de componentes que a plataforma de desenvolvimento oferece para cada sistema operacional.

Esses componentes ricos e muitas vezes sofisticado estão associados a eventos ou procedimentos que executam lógicas de negócio. Problemas de validação de dados são indicados na própria tela sem que qualquer informação do formulário seja perdida.

De uma forma natural, esses componentes lembram-se dos dados do usuário, inclusive entre telas e ações diferentes. Nesse tipo de desenvolvimento são utilizados diversos componentes ricos, como por exemplo, calendários, menus diversos ou componentes drag and drop (arrastar e soltar).

Eles ficam associados a eventos, ou ações, e guardam automaticamente seu estado, já que mantêm os valores digitados pelo usuário.



Esses componentes não estão, contudo, associados exclusivamente ao desenvolvimento de aplicações Desktop. Podemos criar a mesma sensação confortável para o cliente em uma aplicação web, também usando componentes ricos e reaproveitáveis.



## Desenvolvimento desktop ou web?

Existem algumas desvantagens no desenvolvimento desktop. Como cada usuário tem uma cópia integral da aplicação, qualquer alteração precisaria ser propagada para todas as outras máquinas. Estamos usando um cliente gordo, isto é, com muita responsabilidade no lado do cliente.

Note que, aqui, estamos chamando de cliente a aplicação que está rodando na máquina do usuário. Para piorar, as regras de negócio rodam no computador do usuário.

Isso faz com que seja muito mais difícil depurar a aplicação, já que não costumamos ter acesso tão fácil à máquina onde a aplicação está instalada. Em geral, enfrentamos problemas de manutenção e gerenciabilidade.

## O desenvolvimento Web e o protocolo HTTP

Para resolver problemas como esse, surgiram as aplicações baseadas na web. Nessa abordagem há um servidor central onde a aplicação é executada e processada e todos os usuários podem acessá-la através de um cliente simples e do protocolo HTTP.

Um navegador web, como Firefox ou Chrome, que fará o papel da aplicação cliente, interpretando HTML, CSS e JavaScript que são as tecnologias que ele entende. Enquanto o usuário usa o sistema, o navegador envia requisições (requests) para o lado do servidor (server side), que responde para o computador do cliente (client side).

Em nenhum momento a aplicação está salva no cliente: todas as regras da aplicação estão no lado do servidor. Por isso, essa abordagem também foi chamada de cliente magro (thin client).



Isso facilita bastante a manutenção e a gerenciabilidade, pois temos um lugar central e acessível onde a aplicação é executada. Contudo, note que será preciso conhecer HTML, CSS e JavaScript, para fazer a interface com o usuário, e o protocolo HTTP para entender a comunicação pela web.

E, mais importante ainda, não há mais eventos, mas sim um modelo bem diferente orientado a requisições e respostas. Toda essa base precisará ser conhecida pelo desenvolvedor.

Comparando as duas abordagens, podemos ver vantagens e desvantagens em ambas.

No lado da aplicação puramente Desktop, temos um estilo de desenvolvimento orientado a eventos, usando componentes ricos, porém com problemas de manutenção e gerenciamento. Do outro lado, as aplicações web são mais fáceis de gerenciar e manter, mas precisamos lidar com HTML, conhecer o protocolo HTTP e seguir o modelo requisição/resposta.

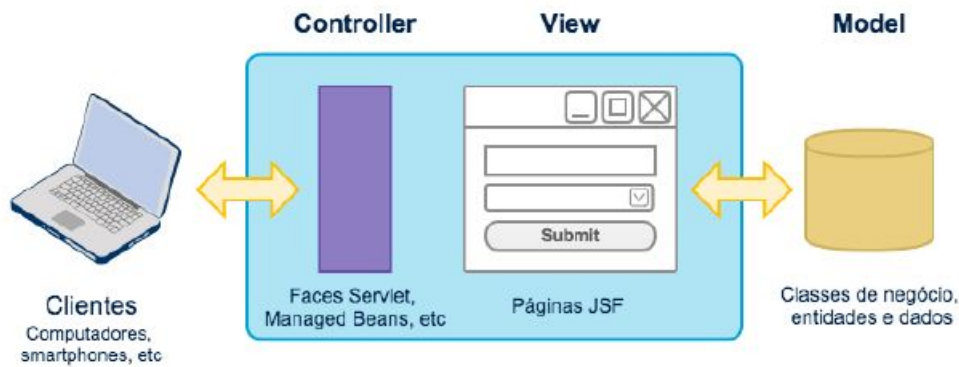
## **Mesclando desenvolvimento Desktop e Web**

Em vez de desenvolver puramente para desktop, é uma tendência mesclar os dois estilos, aproveitando as vantagens de cada um. Seria um desenvolvimento Desktop para a web, tanto central quanto com componentes ricos, aproveitando o melhor dos dois mundos e abstraindo o protocolo de comunicação. Essa é justamente a ideia dos frameworks web baseados em componentes.

No mundo Java há algumas opções como JavaServer Faces(JSF), Apache Wicket, Vaadin, Tapestry ou GWT da Google. Todos eles são frameworks web baseados em componentes

## **Características do JSF**

JSF é uma tecnologia que nos permite criar aplicações Java para Web utilizando componentes visuais pré-prontos, de forma que o desenvolvedor não se preocupe com Javascript e HTML. Basta adicionarmos os componentes (calendários, tabelas, formulários) e eles serão renderizados e exibidos em formato html.



## Guarda o estado dos componentes

Além disso o estado dos componentes é sempre guardado automaticamente (como veremos mais à frente), criando a característica Stateful. Isso nos permite, por exemplo, criar formulários de várias páginas e navegar nos vários passos dele com o estado das telas sendo mantidos.

## Separa as camadas

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 bem separada do conjunto de classes de modelo.

## Especificação: várias implementações

O JSF ainda tem a vantagem de ser uma especificação do Java EE, isto é, todo servidor de aplicações Java tem que vir com uma implementação dela e há diversas outras disponíveis.

A implementação mais famosa do JSF e também a implementação de referência, é a Oracle Mojarra disponível em <http://javaserverfaces.java.net/>. Outra implementação famosa é a MyFaces da Apache Software Foundation em <http://myfaces.apache.org/>.



## Primeiros passos com JSF

A implementação Mojarra do JSF já define o modelo de desenvolvimento e oferece alguns componentes bem básicos. Nada além de inputs, botões e ComboBoxes simples.

The screenshot shows a web form with the following elements:

- A text input field containing 'JSF'.
- A password input field with five asterisks '\*\*\*\*\*'.
- A text area containing the text 'JSF é component-based.'.
- A dropdown menu with 'JSF' selected.
- A row of radio buttons for selecting a framework: ☒ JSF, ☐ Tapestry, ☐ vaadin, ☐ Wicket, and ☐ GWT.
- A list box containing the same framework names: JSF, Tapestry, vaadin, Wicket, and GWT.
- A 'salva' button.
- A blue link labeled 'salva' below the button.

Não há componentes sofisticados dentro da especificação e isso é proposital: uma especificação tem que ser estável e as possibilidades das interfaces com o usuário crescem muito rapidamente.

A especificação trata do que é fundamental, mas outros projetos suprem o que falta. Para atender a demanda dos desenvolvedores por componentes mais sofisticados, há várias extensões do JSF que seguem o mesmo ciclo e modelo da especificação.

Exemplos dessas bibliotecas são PrimeFaces, RichFaces e IceFaces. Todas elas oferecem componentes JSF que vão muito além da especificação.



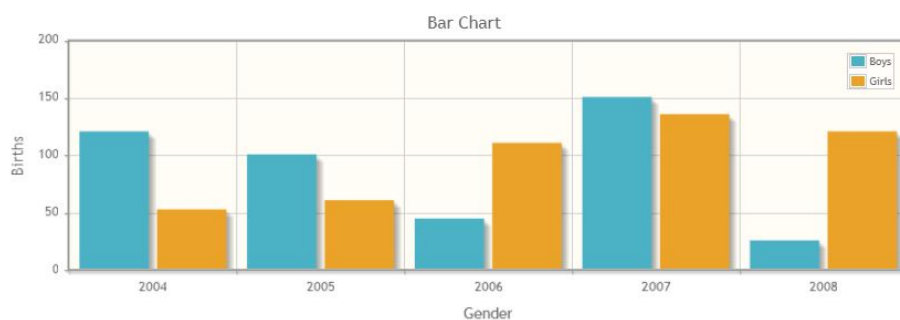
Cada biblioteca oferece ShowCases na web para mostrar seus componentes e suas funcionalidades. Você pode ver o showcase do PrimeFaces no endereço <http://www.primefaces.org>.

Na sua demo online, podemos ver uma lista de componentes disponíveis, como inputs, painéis, botões diversos, menus, gráficos e componentes drag & drop, que vão muito além das especificações, ainda mantendo a facilidade de uso:

October 2018						
S	M	T	W	T	F	S
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31			

### Charts - Bar

Bar chart is created with a BarChartModel.





The screenshot shows a web form with four tabs: 'Personal' (selected), 'Address', 'Contact', and 'Confirmation'. Below the tabs is a section titled 'Personal Details'. Inside this section, there are three input fields: 'Firstname: \*', 'Lastname: \*', and 'Age:'. Below these fields is a checkbox labeled 'Skip to last:'. At the bottom right of the form, there is a blue button with a right arrow and the text 'Next'.

Para a definição da interface do projeto Argentum usaremos Oracle Mojarra com PrimeFaces, uma combinação muito comum no mercado.

## Configuração do controlador do JSF

O JSF segue o padrão arquitetural MVC (Model-View-Controller) e faz o papel do Controller da aplicação. Para começar a usá-lo, é preciso configurar a servlet do JSF no web.xml da aplicação. Esse Servlet é responsável por receber as requisições e delegá-las ao JSF. Para configurá-lo basta adicionar as seguintes configurações no web.xml:

```
5 <welcome-file-list>
6   <welcome-file>index.jsf</welcome-file>
7 </welcome-file-list>
8
9 <servlet>
10   <servlet-name>Faces Servlet</servlet-name>
11   <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
12   <load-on-startup>1</load-on-startup>
13 </servlet>
14
15 <servlet-mapping>
16   <servlet-name>Faces Servlet</servlet-name>
17   <url-pattern>*.jsf</url-pattern>
18 </servlet-mapping>
19 </web-app>
20
```

Ao usar o Eclipse com suporte a JSF 2 essa configuração no web.xml já é feita automaticamente durante a criação de um projeto.

## Faces-config: o arquivo de configuração do mundo JSF

Além disso, há um segundo XML que é o arquivo de configuração relacionado com o mundo JSF, o faces-config.xml. Como o JSF na versão dois encoraja o uso de anotações em vez de configurações no XML, este arquivo torna-se pouco usado. Ele era muito mais importante na primeira versão do JSF. Neste treinamento, deixaremos ele vazio:



```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <faces-config xmlns="http://xmlns.jcp.org/xml/ns/javaee"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
5     http://xmlns.jcp.org/xml/ns/javaee/web-facesconfig_2_2.xsd"
6   version="2.2">
7
8 </faces-config>
9
```

## O que é ManagedBean?

Um sistema de cadastro por exemplo, assim que o usuário terminar de digitar seus dados e clicar em concluir o managedBeans irá receber estas informações e verificar se tem algum erro e retornar uma página dizendo que o cadastro foi feito ou irá retornar uma página informando os erros. O managedBean é o Controller neste caso.

A principal responsabilidade de um managedBean é intermediar a comunicação entre as páginas (componentes do JSF) e nosso modelo. Escutar eventos, processá-los e delegar para a camada de negócios são apenas algumas de suas responsabilidades.

A comunicação entre managedBeans não difere da troca de mensagens entre componentes de software na orientação a objetos. Cada componente (managedBean) conhece a interface pública do outro componente para que eles possam trocar mensagens.

ManagedBeans são componentes, em sua grande maioria, intimamente ligados a(s) página(s) e que deveriam ter apenas o estritamente necessário para representar a GUI. Dificilmente você terá um managedBean genérico (CRUD não conta) que poderia

funcionar em todo lugar. Você precisa fazer um esforço hercúleo para conseguir isso, e as vezes parece que não vale a pena.

Todo managedBean é ligado com uma tela diretamente e podemos identificar isso através da anotação @ManagedBean.

```
13
14 @ManagedBean(name = "usuarioPessoaManagedBean")
15 @ViewScoped
16 public class UsuarioPessoaManagedBean {
17
18     private UsuarioPessoa usuarioPessoa = new UsuarioPessoa();
19     private DaoGeneric<UsuarioPessoa> daoGeneric = new DaoGeneric<>();
20     private List<UsuarioPessoa> list = new ArrayList<UsuarioPessoa>();
21
22     public UsuarioPessoa getUsuarioPessoa() {
23         return usuarioPessoa;
24     }
25
26     public void setUsuarioPessoa(UsuarioPessoa usuarioPessoa) {
27         this.usuarioPessoa = usuarioPessoa;
28     }
29 }
```

## Principais componentes

O verdadeiro poder de JavaServer Faces está em seu modelo de componentes de interface do usuário, que gera alta produtividade aos desenvolvedores, permitindo a construção de interfaces para web usando um conjunto de componentes pré construídos, ao invés de criar interfaces inteiramente do zero.

Existem vários componentes JSF, desde os mais simples, como um Output Label, que apresenta simplesmente um texto, ou um Data Table, que representa dados tabulares de uma coleção que pode vir do banco de dados.

A API de JSF suporta a extensão e criação de novos componentes, que podem fornecer funcionalidades adicionais. Os principais componentes que a implementação de referência do JSF fornece são: formulário, campos de entrada de texto e senhas, rótulos, links, botões, mensagens, painéis, tabela de dados, etc.

## O que é o MVC Design Pattern?

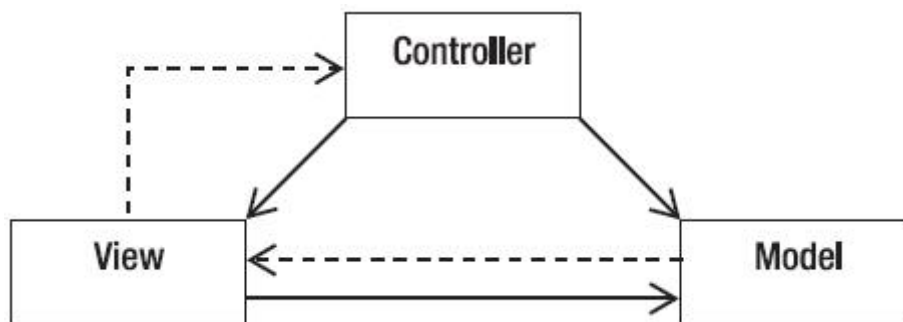
O padrão de design MVC projeta um aplicativo usando três módulos separados

- **Model:** Carrega dados do banco para a tela e da tela para o banco.
- **View:** São as telas que o usuário vê do sistema.
- **Controller:** Manipula o processamento dos dados entre a tela e camada de persistência.

O objetivo do padrão de design do MVC é separar o modelo e a apresentação, permitindo que os desenvolvedores se concentrem em suas habilidades principais e colaborem de forma mais clara.

Web designers têm que se concentrar apenas na camada de visão, em vez de camada de modelo e controlador. Os desenvolvedores podem alterar o código do modelo e normalmente não precisam alterar a camada de visualização.

Os controladores são usados para processar ações do usuário. Nesse processo, o modelo e as visualizações da camada podem ser alterados.



## Ciclo de vida do JSF

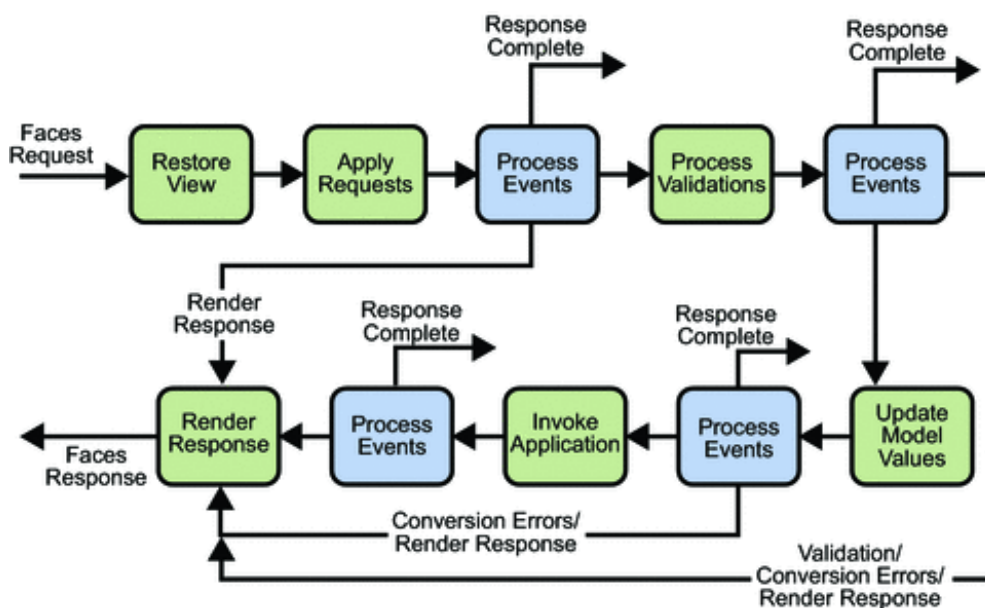
Um dos fundamentos de maior relevância do JSF é seu ciclo de vida que se dá entre a requisição e a resposta do servidor de aplicação. São vários os motivos da existência deste ciclo, dentre estes, temos:

- Manter o controle de estado dos componentes de interface;

- Alinhar ouvintes de eventos com seus respectivos eventos;
- Controle de navegação entre páginas, que deve ser realizado pelo servidor;
- Permitir que validações e conversões sejam realizadas no lado do servidor.

Indo direto ao assunto, o ciclo de vida do JSF se divide em seis fases (veja a figura abaixo), que são:

- Fase 1: Restore View (Restauração da visão);
- Fase 2: Apply Request Values (Aplicar valores da requisição);
- Fase 3: Process Validation (Processar as validações);
- Fase 4: Update Model Values (Atualizar valores de modelo);
- Fase 5: Invoke Application (Invocar aplicação);
- Fase 6: Render Response (Renderizar a resposta).



Vejamos agora o que acontece em cada uma.

## **Fase 1: Restore View (Restauração da visão)**

A partir de uma requisição proveniente do Servlet FacesServlet, é identificado qual visão está sendo requisitada por meio do ID desta que é determinado pelo nome da página JSF. Tendo identificado a página, esta é salva no FacesContext (caso ainda não tenha sido salva) e sua respectiva árvore de componentes é construída.

## **Fase 2: Apply Request Values (Aplicar valores da requisição)**

Fase 2: Apply Request Values (Aplicar valores da requisição) Nesta fase, cada componente da visão, criado ou recuperado, passa a ter o seu valor. Nesse contexto, existem algumas diferenças ocasionadas pelo valor do atributo “immediate” em cada componente:

- immediate = false: Neste caso, que é o padrão, os valores são apenas convertidos para o tipo apropriado. Se o valor é um Integer, é convertido para Integer.
- immediate = true: Os valores são convertidos e validados.

## **Fase 3: Process Validation (Processar as validações)**

Esta é a primeira manipulação de eventos do ciclo, aqui serão executadas as validações definidas pelo servidor em cada componente. Não existindo valores inválidos, o ciclo segue para a Fase 4. Existindo, uma mensagem de erro será gerada (adicionada ao contexto do Faces, FacesContext) e o componente é marcado como inválido. Neste caso, o ciclo segue para a Fase 6 (Renderizar a resposta).

## **Fase 4: Update Model Values (Atualizar valores de modelo)**

Fase 4: Update Model Values (Atualizar valores de modelo) Os valores enviados pela requisição e validados pela fase 3, são atualizados em seus respectivos atributos contidos nos backings beans, onde somente as propriedades enviadas são atualizadas. É importante dizer que, mesmo após a fase de validação, fase 3, os valores enviados podem estar inválidos a nível de negócio ou a nível de conversão de tipos, o que pode ser verificado pelo próprio bean.

## **Fase 5: Invoke Application (Invocar aplicação)**

Nesta fase, os valores dos componentes da requisição, estão validados convertidos e disponíveis nos backings beans. Assim a aplicação tem os insumos necessários para aplicar a lógica de negócio.

Outro fator importante dessa fase, é o direcionamento do usuário de acordo com as submissões realizadas pelo mesmo. Por exemplo, se houve sucesso no processamento dos dados enviados, o usuário é redirecionado para uma determinada página, se não, permanece na mesma página.

## **Fase 6: Render Response (Renderizar a resposta)**

O processo ‘Renderizar a resposta’ consiste na apresentação da página referente ao resultado da aplicação ao usuário. Neste contexto existem três possibilidades:

- Caso seja a primeira requisição da página: Os componentes associados são criados e associados a visão;
- Caso seja a primeira submissão: Os componentes são traduzidos para o HTML;
- Caso tenha ocorrido algum erro: Existindo os marcadores `<f:message />` ou `<f:messages />` na página, os erros são exibidos ao usuário.

## Visão geral

No desenvolvimento em JSF existem dois perfis diferentes de desenvolvedores: os que desenvolveram aplicações JSF e os que desenvolveram componentes. No primeiro perfil, as fases de maior foco são a 2, 3, 4 e 5. Já no segundo, são a 1 e 6, que estão mais relacionadas a árvore de componentes no lado do servidor e de componentes da visão. Sendo assim podemos agrupar as fases da seguinte forma:

- Fases focadas no desenvolvimento de aplicações JSF
- Fase 2: Apply Request Values (Aplicar valores da Requisição)
- Fase 3: Process Validation (Processar as Validações)
- Fase 4: Update Model Values (Atualizar Valores de Modelo)
- Fase 5: Invoke Application (Invocar Aplicação)
- Fases focadas no desenvolvimento de componentes de visão
- Fase 1: Restore View (Restauração da Visão)
- Fase 6: Render Response (Renderizar a Resposta)

## Anotações e Escopo

Anotações de escopo definem o escopo no qual o bean gerenciado será colocado. Se o escopo não for especificado, o bean será padronizado escopo de `@Request`.

Quando referenciamos um `managedBean` via EL, o framework do JSF instanciará um objeto da classe do `managedBean`, ou recuperará uma instância existente. Todas as instâncias possuem um tempo de vida, que é definido dependendo do escopo usado no `managedBean`. Os escopos de `managedBeans` JSF podem ser definidos através de anotações do pacote `javax.faces.bean`. Os principais são:

**@NoneScoped:** o bean será instanciado a cada vez que for referenciado.



**@RequestScoped** (padrão): tem vida curta, começando quando é referenciado em uma única requisição HTTP e terminando quando a resposta é enviada de volta ao cliente.

**@ViewScoped**: a instância permanece ativa até que o usuário navegue para uma próxima página.

**@SessionScoped**: mantém a instância durante diversas requisições e até mesmo navegações entre páginas, até que a sessão do usuário seja invalidada ou o tempo limite é atingido. Cada usuário possui sua sessão de navegação, portanto, os objetos não são compartilhados entre os usuários.

**@ApplicationScoped**: mantém a instância durante todo o tempo de execução da aplicação. É um escopo que compartilha os objetos para todos os usuários do sistema.

## Estrutura de página JSF

As páginas em JSF são estruturas em XML, no início sempre ficam declarados as bibliotecas e a referência deles para as tags JSF.

Logo após termos sempre o HEAD que é o nosso cabeçalho, onde são definidos CSS, JavaScript, título e outros recursos.

E a coisa mais importante que você deve saber é que para o JSF funcionar todos os elementos devem estar dentro do FORM e o este dentro do BODY.

Tendo está estrutura inicial dentro do FORM será onde criaremos toda a estrutura de uma página, por exemplo um cadastro de pessoa.

```
1 <?xml version="1.0" encoding="ISO-8859-1" ?>
2
3 <html xmlns="http://www.w3.org/1999/xhtml"
4     xmlns:f="http://java.sun.com/jsf/core"
5     xmlns:h="http://java.sun.com/jsf/html"
6     xmlns:p="http://primefaces.org/ui">
7
8 <h:head>
9 </h:head>
10
11 <h:body>
12     <h:form>
13
14     </h:form>
15 </h:body>
16
17 </html>
18
```

## Configurando JSF com Maven

Para usarmos nosso projeto JSF precisamos não apenas do JSF e também precisamos do Hibernate, PostgreSQL, PrimeFaces e para isso tem que ser baixado uma lista enorme de bibliotecas o principal benefício do Maven é isso baixar tudo que precisa de uma vez e com versões compatíveis.

Pra começar as configurações vou listar abaixo o código XML do pom.xml do Maven.

Configurando PostgreSQL com Maven.

```
<!-- https://mvnrepository.com/artifact/org.postgresql/postgresql -->
<dependency>
  <groupId>org.postgresql</groupId>
  <artifactId>postgresql</artifactId>
  <version>9.2-1003-jdbc4</version>
</dependency>
```

Configurando o Hibernate + JPA com Maven.

```
<!-- HIBERNATE -->
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>5.2.6.Final</version>
</dependency>

<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-entitymanager</artifactId>
  <version>5.2.6.Final</version>
</dependency>
```

Configurando JSF com Maven.

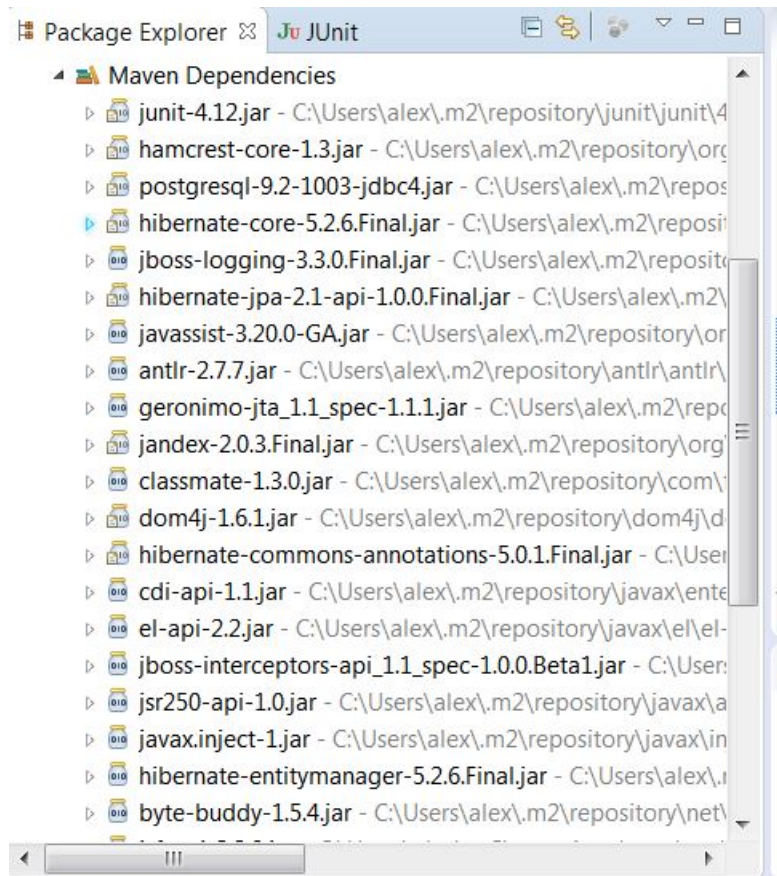
```
<!-- https://mvnrepository.com/artifact/com.sun.faces/jsf-api -->
<dependency>
  <groupId>com.sun.faces</groupId>
  <artifactId>jsf-api</artifactId>
  <version>2.2.0</version>
  <scope>compile</scope>
</dependency>

<!-- https://mvnrepository.com/artifact/com.sun.faces/jsf-impl -->
<dependency>
  <groupId>com.sun.faces</groupId>
  <artifactId>jsf-impl</artifactId>
  <version>2.2.0</version>
  <scope>compile</scope>
</dependency>
```

Configurando PrimeFaces com Maven.

```
<dependency>
  <groupId>org.primefaces</groupId>
  <artifactId>primefaces</artifactId>
  <version>6.2</version>
</dependency>
```

Após configurar e salvar todas as alterações o Maven irá automaticamente baixar da internet todas as bibliotecas/JAR do Java e dos frameworks para podermos iniciar nosso projeto.



## Ativando o JSF em nosso projeto

Para o nosso projeto Java enxergar o uso do JSF em nosso projeto temos que ativar o FacesServlet em nosso projeto, essa classe é a principal do JSF.

FacesServlet é um Servlet que gerencia o ciclo de vida de processamento de pedidos para aplicativos da web que estão utilizando o JavaServer Faces para construir a interface com o usuário.

```
9  <servlet>
10  <servlet-name>Faces Servlet</servlet-name>
11  <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
12  <load-on-startup>1</load-on-startup>
13  </servlet>
14
15  <servlet-mapping>
16  <servlet-name>Faces Servlet</servlet-name>
17  <url-pattern>*.jsf</url-pattern>
18  </servlet-mapping>
```

## Configurando o Persistence.xml

Nesse arquivo ficam as configurações do banco de dados e das classes mapeadas como entidades de nosso projeto.

```
<persistence-unit name="pos-java-maven-hibernate">
  <class>model.UsuarioPessoa</class>
  <class>model.TelefoneUser</class>
  <properties>
    <!-- Dados de conexão com o banco -->
    <property name="hibernate.connection.driver_class" value="org.postgresql.Driver" />
    <property name="hibernate.connection.url"
      value="jdbc:postgresql://localhost:5432/posjavahibernate" />
    <property name="hibernate.dialect" value="org.hibernate.dialect.PostgreSQLDialect" />
    <property name="hibernate.connection.username" value="postgres" />
    <property name="hibernate.connection.password" value="admin" />
    <property name="hibernate.format_sql" value="false" />
    <property name="hibernate.hbm2ddl.auto" value="update" />
  </properties>
</persistence-unit>
```

## Classe Java HibernateUtil

Essa classe é responsável por ler o arquivo persistence.xml e criar a conexão com o banco de dados e todas as opções configuradas.

Nesta mesma classe que é criado o EntityManagerFactory. Interface usada para interagir com a fábrica do gerenciador de entidades para a unidade de persistência.



Quando o aplicativo terminar de usar a fábrica do gerenciador de entidades e / ou no encerramento do aplicativo, o aplicativo deverá fechar a fábrica do gerenciador de entidades.

Uma vez que um EntityManagerFactory tenha sido fechado, todos os seus gerentes de entidade são considerados em estado fechado.

```
7 public class HibernateUtil {
8     public static EntityManagerFactory factory = null;
9
10    static {
11        init();
12    }
13    private static void init() {
14        try {
15            if (factory == null) {
16                factory = Persistence.
17                    createEntityManagerFactory("pos-java-maven-hibernate");
18            }
19        } catch (Exception e) {
20            e.printStackTrace();
21        }
22    }
23 }
```

## Mapeando nossa entidade Usuário

A Java Persistence API (JPA), parte da especificação EJB 3.0 do Java Enterprise Edition 5, simplifica muito a persistência de Java e fornece uma abordagem de mapeamento objeto-relacional que permite declarativamente definir como mapear objetos Java para tabelas de banco de dados relacional de uma maneira padrão e portátil que funciona tanto dentro de um servidor de aplicativos Java EE 5 quanto fora de um contêiner EJB em um aplicativo Java Standard Edition 5 (Java SE 5).

Usando o JPA, você pode designar qualquer classe POJO como um JPA entidade - um objeto Java cujos campos não transitórios devem ser mantidos em um banco de dados relacional usando os serviços de um gestor da entidade obtido de uma APP provedor de persistência (dentro de um contêiner EJB Java EE ou fora de um contêiner EJB em um aplicativo Java SE).

Uma entidade tem as seguintes características:

- É compatível com o EJB 3.0;
- É leve;
- Ele gerencia dados persistentes em conjunto com um gerenciador de entidades JPA;
- Realiza lógica de negócios complexa;
- Potencialmente usa vários objetos Java dependentes;
- Pode ser identificado exclusivamente por uma chave primária.

Entidades representam dados persistentes armazenados em um banco de dados relacional automaticamente usando persistência gerenciada por contêineres. Eles são persistentes porque seus dados são armazenados de forma persistente em alguma forma de sistema de armazenamento de dados, como um banco de dados: eles sobrevivem a uma falha de servidor ou rede. Quando uma entidade é reinstanciada, o estado da instância anterior é automaticamente restaurado.

Uma entidade modela uma entidade de negócios ou várias ações dentro de um único processo de negócios. As entidades geralmente são usadas para facilitar os serviços de negócios que envolvem dados e cálculos nesses dados. Por exemplo, você pode implementar uma entidade para recuperar e executar cálculos em itens dentro de um pedido de compra. Sua entidade pode gerenciar vários objetos persistentes dependentes na execução de suas tarefas.

Entidades podem representar objetos persistentes refinados, porque não são componentes remotamente acessíveis.

Uma entidade pode agregar objetos juntos e efetivamente persistir dados e objetos relacionados usando os serviços transacionais, de segurança e de simultaneidade de um provedor de persistência JPA.

```
15 @Entity
16 public class UsuarioPessoa {
17
18     @Id
19     @GeneratedValue(strategy = GenerationType.AUTO)
20     private Long id;
21
22     private String nome;
23     private String sobrenome;
24     private String email;
25     private String login;
26     private String senha;
27     private int idade;
28 }
```

## Estrutura padrão de um ManagedBean

Todo ManagedBean por mais simples que seja o cadastro possui uma estrutura padrão para o seu funcionamento.

Abaixo cito e demonstro em código esses pontos principais:

- Objeto de regra de negócio da tela.
- DAO de persistência.
- Lista que carrega os objetos cadastrados.

```
14 @ManagedBean(name = "usuarioPessoaManagedBean")
15 @ViewScoped
16 public class UsuarioPessoaManagedBean {
17
18     // Objeto de regra de negócio da tela
19     private UsuarioPessoa usuarioPessoa = new UsuarioPessoa();
20
21     //Dao de persistência
22     private DaoGeneric<UsuarioPessoa> daoGeneric = new DaoGeneric<>();
23
24     //Lista que carrega os objetos cadastrados
25     private List<UsuarioPessoa> list = new ArrayList<UsuarioPessoa>();
26 }
```



## PanelGrid em JSF e EL

A tag `h:panelGrid` renderiza um elemento "table" em HTML.

A nova versão da EL contempla adições significativas na linguagem de expressões, principalmente por conta do suporte a expressões lambda e das funcionalidades para manipulação de coleções inspiradas no modelo de programação funcional que está sendo incorporado à Java SE 8.

A partir de agora os desenvolvedores poderão ordenar, filtrar e transformar coleções de forma bastante simplificada utilizando a EL e expressões lambda.

Outro ponto bastante importante é que a EL 3.0 desacopla a linguagem de expressões das tecnologias web (JSP e JSF) e cria um grande potencial ao trazer suas funcionalidades para fora do container Java EE através da API standalone. Com essa nova API os desenvolvedores poderão escrever e resolver expressões EL dentro do bom e velho código Java.

Além dessas principais mudanças, a EL 3.0 traz outros recursos e melhorias à linguagem de expressões a fim de atender pedidos antigos da comunidade e maximizar a produtividade dos desenvolvedores. Entre essas melhorias estão a introdução de novos operadores (incluindo um para concatenação de Strings) e o acesso a construtores e membros estáticos de classes Java.

A forma que ligamos os atributos com os dados da página JSF com o managedBean é pela expression language em JSF que é identificado pelo símbolo `#{}`.

```
12
13     <h:messages showDetail="true" showSummary="false" />
14     <h:panelGrid columns="2">
15         <h:outputLabel>Id:</h:outputLabel>
16         <h:inputText readonly="true"
17             value="#{usuarioPessoaManagedBean.usuarioPessoa.id}" />
18
19         <h:outputLabel>Nome:</h:outputLabel>
20         <h:inputText value="#{usuarioPessoaManagedBean.usuarioPessoa.nome}" required="true"
21
22         <h:outputLabel>Sobrenome:</h:outputLabel>
23         <h:inputText
24             value="#{usuarioPessoaManagedBean.usuarioPessoa.sobrenome}" required="true" requ
25
26         <h:outputLabel>Email:</h:outputLabel>
27         <h:inputText value="#{usuarioPessoaManagedBean.usuarioPessoa.email}" />
```

## JSF – DataTable

O JSF fornece um controle avançado chamado DataTable para renderizar e formatar tabelas html.

- DataTable pode iterar em uma coleção ou matriz de valores para exibir dados.
- O DataTable fornece atributos para modificar seus dados de maneira fácil.

Esse componente carrega os dados que estão em um managedBean e sempre utilizamos uma lista para isso e no datatable o atributo value apontamos essa lista e trabalhamos os dados com uma variável que representa o objeto ou item da lista.

```
<h:dataTable value="#{usuarioPessoaManagedBean.list}" var="user"
  cellpadding="15" border="1">
  <f:facet name="header">Lista de usuários</f:facet>
  <h:column style="width:150px;">
    <f:facet name="header">Id</f:facet>
    <h:outputText value="#{user.id}" />
  </h:column>
```

## Mensagens com FacesMessage

FacesMessage representa uma única mensagem de validação (ou outra), que é normalmente associada a um componente específico na exibição. Uma FacesMessage instância pode ser criada com base em um específico messageId. A especificação define o conjunto de messageIds para os quais deve haver FacesMessage instâncias.

```
FacesContext.getCurrentInstance().addMessage(null,
  new FacesMessage(FacesMessage.SEVERITY_INFO, "Informação: "
    , "Salvo com sucesso!"));
```

No formulário em JSF o componente que exibe as mensagens enviadas para o FacesMessage é o h:messages.

```
<h:messages showDetail="true" showSummary="false" />
```

## Exibindo mensagens após redirecionamento

Para conseguir com que as mensagens sobrevivam ao redirecionamento, basta usar o código abaixo:

```
FacesContext.getCurrentInstance().getExternalContext().  
getFlash().setKeepMessages(true);
```

Essa linha vai fazer com que suas mensagens sejam armazenadas no escopo de Flash.

A duração desse escopo vai ser o suficiente para que você consiga exibir as mensagens adicionadas depois do redirecionamento.

O código completo do método ficaria assim:

```
public String salvar() {  
    daoGeneric.salvar(usuarioPessoa);  
  
    FacesContext.getCurrentInstance().addMessage(null,  
        new FacesMessage(FacesMessage.SEVERITY_INFO,  
            "Informação: ", "Salvo com sucesso!"));  
  
    FacesContext.getCurrentInstance().getExternalContext().  
    getFlash().setKeepMessages(true);  
  
    return "user-saovo-sucesso.xhtml?faces-redirect=true";  
}
```

## CommandButton JSF

A tag `h:commandButton` renderiza um elemento de entrada HTML do tipo "submit".

Usamos o `commandButon` para saber, atualizar, remover e consultar dados, ele faz o envio do formulário para o lado do servidor e para isso é usado o atributo `action` onde colocamos o método do `managedBean` que queremos chamar.

```
<h:commandButton value="Salvar"
    action="#{usuarioPessoaManagedBean.salvar}" />
<h:commandButton value="Novo"
    action="#{usuarioPessoaManagedBean.novo}" />
```

Então, para entendermos bem o processo quando clicamos no botão o JSF irá fazer todas as validações necessários caso haja e enviará os dados para o lado do `managedBean` e é nesse momento que os dados da tela são adicionados/setado para o objeto que representa a tela e após seta o método irá ser chamado.

De acordo com os botões acima pode ver abaixo quais serão os métodos que serão chamados.

```
public String salvar() {
    daoGeneric.salvar(usuarioPessoa);
    FacesContext.getCurrentInstance().addMessage(null,
        new FacesMessage(FacesMessage.SEVERITY_INFO,
            "Informação: ", "Salvo com sucesso!"));
    return "";
}

public String novo() {
    usuarioPessoa = new UsuarioPessoa();
    return "";
}
```

## Quando usar Action ou ActionListener com JSF

Basicamente temos duas formas de executar uma ação no managedBean após clicar em um botão ou link com JSF. Podemos passar uma EL (Expression Language) para o atributo action ou para o atributo ActionListener do componente. Apesar de ambos os atributos invocarem um método no managedBean, eles possuem uma diferença sutil que costuma levar a pergunta: quando e qual devo usar?

Não é difícil decidir qual usar. Para isso precisamos antes entender a motivação por trás de cada um e suas peculiaridades.

### O Action

Você deveria usar uma action se sua intenção é executar uma lógica de negócio ou navegar entre páginas. O método da action pode retornar uma String indicando a regra de navegação.

Por exemplo, para executar uma lógica de negócio que grava um usuário no banco de dados teríamos algo como:

```
<p:commandButton value="Salvar" ajax="false"  
    action="#{usuarioPessoaManagedBean.salvar}" />
```

E no managedBean teríamos este método:

```
public String salvar() {  
    daoGeneric.salvar(usuarioPessoa);  
    FacesContext.getCurrentInstance().addMessage(null,  
        new FacesMessage(FacesMessage.SEVERITY_INFO,  
            "Informação: ", "Salvo com sucesso!"));  
    return "usuario-salvo";  
}
```

Repare que após gravar o usuário o managedBean executa uma regra de navegação para que o usuário seja redirecionado para a página de listagem de usuários.

Esta regra de navegação é conhecida como outcome. Se o método retorna null ou possui retorno void, o usuário é mantido na mesma página e o JSF reaproveita a mesma árvore de componentes.

O mesmo pode ser obtido se uma String vazia é retornada ou o outcome retornado leve para a mesma página, nesse caso a diferença é que uma nova árvore de componentes é criada.

## O ActionListener

Você deveria usar uma ActionListener se o que você quer fazer é executar uma lógica relacionada a view ou disparar uma ação antes de uma lógica de negócio.

A lógica invocada por uma ActionListener está mais ligada a detalhes da tela do que puramente regras de negócio. Para usá-lo em uma h:commandButton bastaria termos o código a seguir:

A diferença principal entre o action e o ActionListener é que o ActionListener não faz redirecionamento e nem envio do formulário e mesmo assim ainda podemos executar métodos Java do lado do servidor.

```
<h:commandButton value="Novo"  
    actionListener="#{usuarioPessoaManagedBean.novo}" />
```



## O setPropertyActionListener

No JSF, a tag “ f: setPropertyActionListener ” permite que você defina um valor diretamente na propriedade do seu bean de apoio.

Quando vamos editar, excluir ou consultar um objeto em JSF é o setPropertyActionListener que será usado, ele seta para o managedBean o objeto selecionado em tela e conseguindo enviar o objeto que queremos para o lado do servidor fica fácil criar qualquer rotina.

```
<p:column headerText="Editar">
    <h:commandLink value="Editar">
        <f:setPropertyActionListener value="#{user}"
            target="#{usuarioPessoaManagedBean.usuarioPessoa}" />
    </h:commandLink>
</p:column>

<p:column headerText="Remover">
    <h:commandLink value="Remover"
        action="#{usuarioPessoaManagedBean.remover}">
        <f:setPropertyActionListener value="#{user}"
            target="#{usuarioPessoaManagedBean.usuarioPessoa}" />
    </h:commandLink>
</p:column>
```

## Utilizando AJAX com JSF de maneira eficiente

Inicialmente, as páginas WEBs eram desenvolvidas de maneira estática, ou seja, para uma atualização refletir ao cliente, toda a página tinha que ser carregada. Com isso, para atualizar algum pequeno conteúdo, obrigatoriamente o cliente precisava realizar uma requisição, de maneira completa, a aplicação WEB.

Esse problema consiste quando se precisa atualizar repetidamente a aplicação, podendo afetar o desempenho da mesma.

O Ajax veio para solucionar este tipo de problema.

Ajax é um conjunto de tecnologias para desenvolvimento WEB que permite que possam ser desenvolvidas páginas dinâmicas, que respondam a algumas situações de maneira assíncrona, sem que toda a aplicação seja atualizada. Com Ajax, as

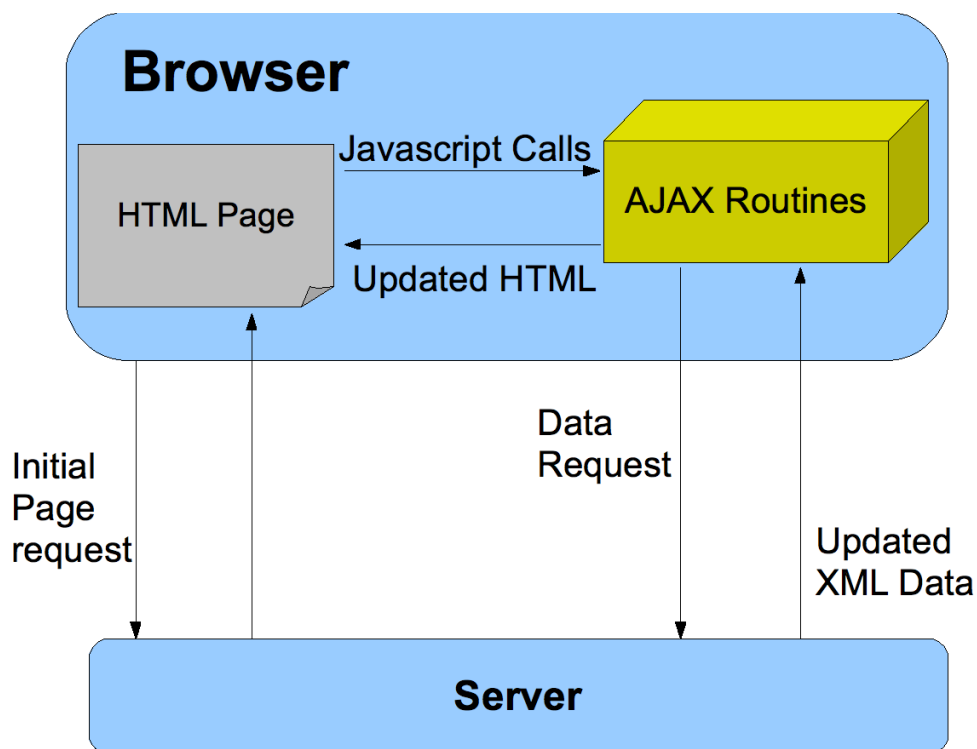
aplicações webs podem recuperar informações do servidor de maneira altamente responsiva com o cliente.

## Visão geral do Ajax

Ajax é baseado em JavaScript e XML, tecnologias utilizadas para desenvolvimentos de aplicações WEBS dinâmicas e assíncronas.

O funcionamento da tecnologia Ajax se baseia no seguinte processo, é enviada uma solicitação assíncrona para o servidor, e este envia de volta uma resposta no modelo DOM para atualizar a página.

Uma observação importante é que a comunicação cliente e servidor não se limitam apenas ao modelo DOM, ela também pode utilizar o formato JSON.





## Utilizando Ajax com JSF

AJAX significa asynchronous JavaScript and Xml.

Ajax é uma técnica para usar XMLHttpRequest de JavaScript para enviar dados para o servidor e receber dados do servidor de forma assíncrona. Assim, usando a técnica Ajax, o código javascript troca dados com o servidor, atualiza partes da página da web sem recarregar a página inteira.

Para entender o funcionamento desta técnica, primeiro é necessário saber como funciona o modelo cliente-servidor. Neste modelo, o cliente envia uma requisição ao servidor e aguarda pela resposta, ou seja, solicita ao servidor para que ele processe algo e espera por um retorno. O servidor processa essa requisição assim que ela chega, e quando esta operação estiver concluída, uma resposta é gerada e enviada para o cliente.

No caso do AJAX, o cliente referenciado no cenário acima, é o browser do usuário, que executará o código escrito na linguagem JavaScript. Este é desenvolvido de forma que dispare requisições para o servidor conforme o usuário interage com a página. Essas requisições são assíncronas porque elas são enviadas para o servidor e processadas de maneira não ordenada, ou seja, não possuem sincronia.

Assim que as requisições chegam ao servidor, elas são processadas, e ao finalizar o processamento da requisição, uma resposta é gerada e enviada para o cliente, geralmente em XML. Quando essa resposta chega ao cliente, o código JavaScript a interpreta e atualiza os valores da página, caso necessário.

Nos dias de hoje existem vários frameworks que auxiliam no desenvolvimento de aplicações AJAX. O JavaServer Faces é um deles, e em sua versão 2.0, passou a suportá-lo nativamente. Deste modo, as requisições e os códigos JavaScript são encapsulados dentro de componentes, logo, na maioria dos casos, não é necessário escrever código em JavaScript.

O JSF fornece suporte excepcional para fazer chamadas ajax. Ele fornece tag `f:ajax` para lidar com chamadas ajax.

A funcionalidade Ajax pode ser adicionada em uma aplicação JSF por meio da importação de suas bibliotecas. Toda aplicação que fizer o seu uso estará representada pela tag `<f:Ajax>`.

Duas opções são as mais importantes quando usamos ajax:

**Execute:** que é o atributo onde especificamos o ID do componente que será enviado para o lado do servidor.

**Render:** que é o atributo que identificamos o componente em tela que será recarregado depois que o execute for processado.

Então supondo que queremos digitar um texto e esse mesmo texto ser recarregado em outro componente temos o exemplo abaixo.

```
<p:inputText value="#{usuarioPessoaManagedBean.usuarioPessoa.nome}"
  required="true" id="inputNome" requiredMessage="Informe o nome!">
  <f:ajax execute="inputNome" render="saidaNome" />
</p:inputText>

<p:outputLabel value="#{usuarioPessoaManagedBean.usuarioPessoa.nome}"
  id="saidaNome"/>
```

## Usando o FilterOpenSessionInView

Um problema que todos que estão trabalhando com Hibernate não está livre de se deparar com ele. É o velho LazyInitializationException que acontece e deixa o pobre do desenvolvedor estressado.

Pretendo ser menos teórico e mais prático, pois tem muita coisa na net explicando o porque temos o LazyInitializationException, porém poucos explicaram de forma prática a solução.

Então aqui vou mostrar como resolver o problema, pois também fui afetado com este problema e levei umas quase 15 horas para resolver. E agora vejo que era algo simples, mas eu precisava entender o por que.

Além de resolver muitos problemas em projetos esse filtro resolve um grande problema na parte de persistência de dados, porque pode estabelecer um ponto único para abrir a sessão, commitar e dar roolback nas transações com o banco de dados e isso mantém a consistência dos dados em uma base de dados do nossos sistemas.

Um segundo problema também que conseguimos resolver e que com esse filtro pode fazer o Hibernate já subir a conexão e todas as configurações quando o servidor estiver sendo estartado.

Então o que temos que fazer é criar uma classe e implementas a interface Filter, mas especificamente do pacote javax.servlet.Filter do JavaEE e também anotar com @WebFilter e por ultimo adicionar a sua declaração no arquivo web.xml para que seja possível a leitura dele pelo projeto, esse padrão permite interceptar toda requisição e resposta em um único ponto do sistema.

```
@WebFilter(filterName = "conexaoFilter", urlPatterns = "/")
public class FilterOpenSessionInView implements Filter {

    @Override
    public void destroy() { // executado quando os recursos são descartados
    }

    @Override // Executado em toda a requisição e resposta
    public void doFilter(ServletRequest servletRequest,
        ServletResponse servletResponse, FilterChain chain)
        throws IOException, ServletException {

        chain.doFilter(servletRequest, servletResponse);
    }

    @Override // Executado uma unica vez quando o servidor sobe no AR
    public void init(FilterConfig arg0) throws ServletException {
    }
}
```

## Declarando Filter no Web.xml

Precisamos nos atentar a dois passos:

- Declarar o filter passando todo o caminho do pacote da classe para sua identificação.
- Fazer o mapeamento de URL para o filter.

```
<filter>
    <filter-name>conexaoFilter</filter-name>
    <filter-class>filter.FilterOpenSessionInView</filter-class>
</filter>
<filter-mapping>
    <filter-name>conexaoFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

Outra opção é mapear com anotações que substituem as configurações no web.xml

```
5
6 @WebFilter(filterName = "conexaoFilter", urlPatterns = { "/" })
7 public class FilterOpenSessionInView implements Filter {
8
```

## Declarando o JPA/Hibernate no Filter

Primeiro passo é declarar o objeto EntityManager para usarmos em nosso filtro.

```
5
6 @WebFilter(filterName = "conexaoFilter", urlPatterns = { "/" })
7 public class FilterOpenSessionInView implements Filter {
8
9     private static EntityManager sf; // Declaração do Entity manager para o filter
10
```

Dentro do nosso método doFilter que é chamado em todas as requisições e respostas nós fazemos a verificação se a sessão está aberta e se sim executamos o SQL em memória diretamente do banco com o flush() e damos o commit();

E por último fechamos a sessão. Temos que tratar qualquer erro que venha acontecer em nosso sistema e para isso o filter que implementamos resolve muito bem qualquer problema então caso venha acontecer algum erro dentro do Exception damos o rollback para rever alterações no banco de dados feita na transação atual.

```
29
30     try {
31
32         chain.doFilter(servletRequest, servletResponse);
33
34         if (sf.isOpen()) {
35             sf.flush();
36             sf.getTransaction().commit();
37         }
38
39         if (sf.isOpen()) {
40             sf.close();
41         }
42
43     } catch (Exception e) {}
44     sf.getTransaction().rollback();
45 }
46
```

## Tags Básicas

- h: `inputText` Processa uma entrada HTML de `type = "text"`, caixa de texto.
- h: `inputSecret` Processa uma entrada HTML de `tipo = "senha"`, caixa de texto.
- h: `inputTextarea` Processa um campo `textarea` HTML.
- h: `inputHidden` Processa uma entrada HTML de `type = "hidden"`.
- h: `selectBooleanCheckbox` Processa uma única caixa de seleção HTML.
- h: `selectManyCheckbox` Processa um grupo de caixas de seleção de HTML.
- h: `selectOneRadio` Processa um único botão de opção HTML.
- h: `selectOneListbox` Processa uma caixa de listagem única em HTML.
- h: `selectManyListbox` Renderiza uma caixa de listagem múltipla em HTML.
- h: `selectOneMenu` Processa uma caixa de combinação HTML.
- h: `outputText` Processa um texto HTML.
- h: `outputFormat` Processa um texto HTML. Aceita parâmetros.
- h: `graphicImage` Processa uma imagem.
- h: `outputStylesheet` Inclui uma folha de estilo CSS na saída HTML.
- h: `outputScript` Inclui um script na saída HTML.
- h: `commandButton` Processa uma entrada HTML do botão `type = "submit"`.
- h: `Link` Processa uma âncora HTML.
- h: `commandLink` Processa uma âncora HTML.
- h: `outputLink` Processa uma âncora HTML.
- h: `panelGrid` Renderiza uma Tabela HTML em forma de grade.
- h: `mensagem` Processa mensagem para um componente de UI JSF.
- h: `mensagens` Processa todas as mensagens para os Componentes de UI JSF.
- f: `param` Passar parâmetros para o componente de interface do usuário do JSF.
- f: `attribute` Passe o atributo para um componente de interface do usuário do JSF.
- f: `setPropertyActionListener` Define o valor da propriedade de um bean gerenciado.

## Parâmetros com JSF

A tag `f:param` fornece as opções para passar parâmetros para um componente ou passar parâmetros de requisição.

Ele funciona passando parâmetros pela URL ou também passando para componentes e isso é uma coisa muito bacana e funcional.

Abaixo temos o exemplo de como passamos parâmetros e fazemos a captura dele do lado do servidor.

```
<h:commandButton action="#{user.editAction}">
    <f:param name="action" value="delete" />
</h:commandButton>
```

E do lado do servidor pegamos o valor do action com o seguinte código.

```
Map<String,String> params =
    FacesContext.getExternalContext().getRequestParameterMap();
String action = params.get("action");
..
```

Agora na variável action temos o valor passado como parametros para assim definir o que processar do lado do servidor.

## Conhecendo show case JSF e PrimeFaces

PrimeFaces é uma coleção de componentes de UI ricos para JavaServer Faces. Todos os widgets são open source e gratuitos para uso sob licença Apache. O PrimeFaces é desenvolvido pela PrimeTek Informatics, um fornecedor com anos de experiência no desenvolvimento de soluções de UI de código aberto.

A forma mais correta e fácil de aprender a usar os componentes de um frameworks é olhando a documentação e nessa parte o PrimeFaces é excelente ele possui um show case onde tem o exemplo como ficará em seu sistema, tem o código fonte na página XHTML e também o código do ManagedBean, para todos os componentes do primefaces existe um exemplo bem práticos e fácil demonstrando a construção do mesmo, para usar recursos mais avançados e específicos de cada componente é comente baixar o PDF da documentação do PrimeFaces procurar pelo capítulo do componente que existe uma tabela de todas as opções para cada um dos componentes.

Vamos aprender um pouco como usar o show case do PrimeFaces.



Para acessar a área de exemplo basta ir para o link <https://www.primefaces.org/showcase/>.

A esquerda existem todos os componentes disponíveis no frameworks e para cada categoria existe uma lista de componentes mais sofisticados ou mais simples e cada um tem o seu objetivo em solucionar um problema ou ser implementado no sistema para uma solução.

Show case do AjaxCore: Um método java pode ser invocado em uma requisição ajax usando a opção listener de p:ajax.

### Ajax Framework - Listener

A java method can be invoked in an ajax request using listener option of p:ajax.

Keyup:

listener.xhtml

ListenerView.java

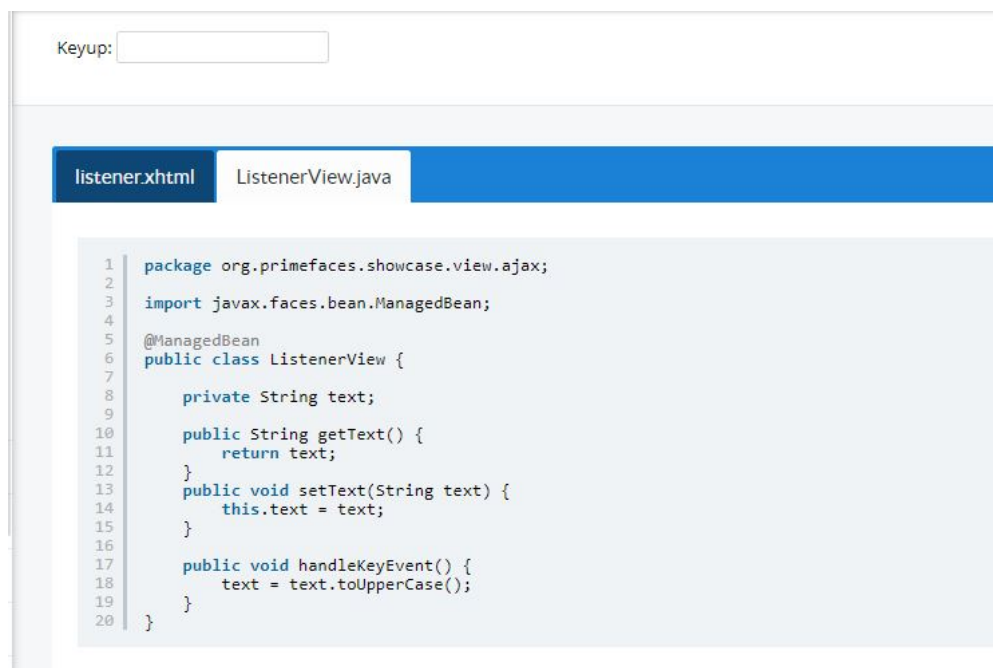
```
1 <h:form>
2   <h:panelGrid columns="3">
3     <h:outputText value="Keyup: " />
4     <p:inputText id="counter" value="#{listenerView.text}">
5       <p:ajax event="keyup" update="out" listener="#{listenerView.handleKeyEvent}" />
6     </p:inputText>
7     <h:outputText id="out" value="#{listenerView.text}" />
8   </h:panelGrid>
9 </h:form>
```



Então todos seguem o mesmo padrão onde em cima é como fica o componente e em baixo a primeira parte sempre é o código JSF e a segunda parte o código Java do lado no managedBean (Servidor).

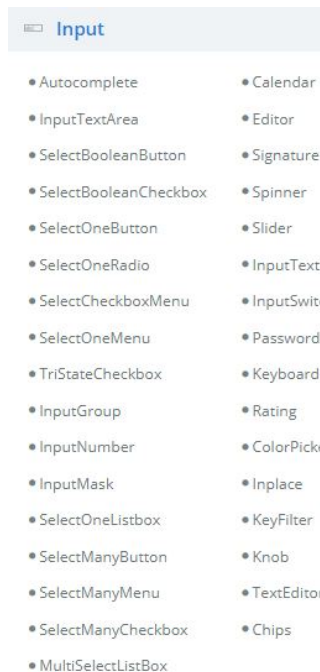


Seguindo essa linha de pensamento na documentação fica superfácil criar qualquer sistema em JSF e PrimeFaces.



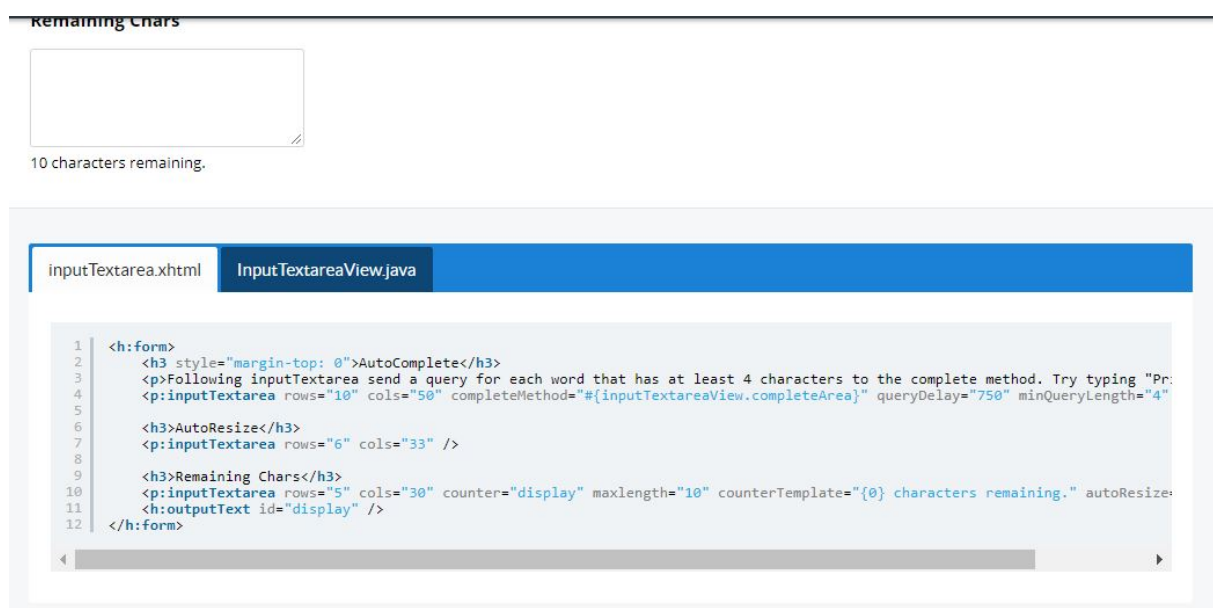
Show case dos componentes Input:

A maior variedade está nessa parte onde possui a maior quantidade de componentes para usar em todos os tipo de sistemas que podem ser pensados.

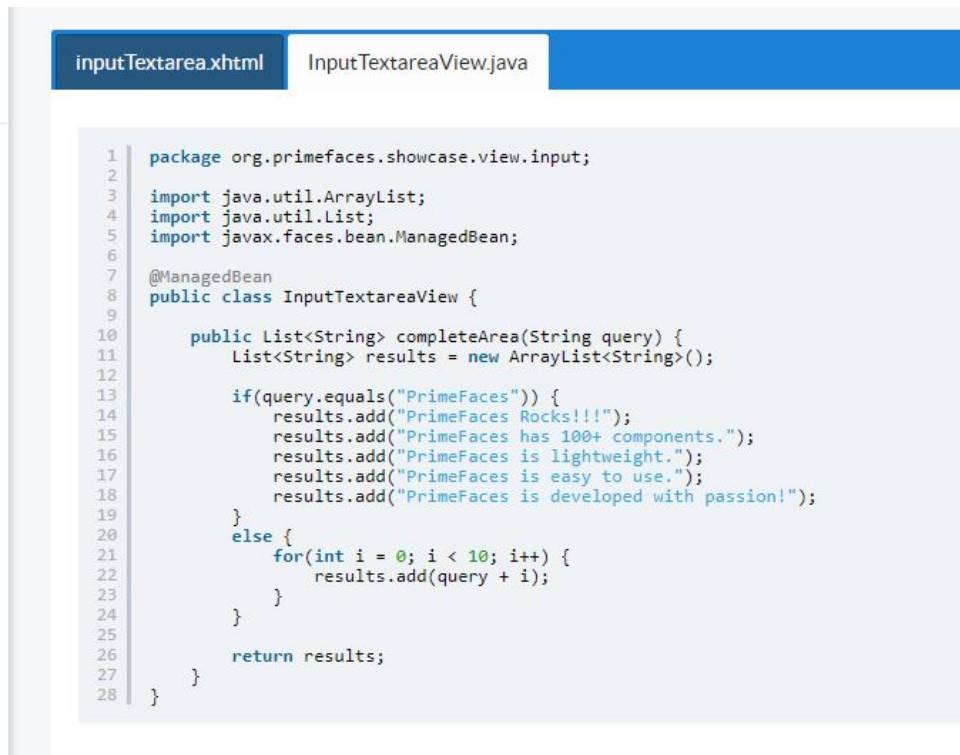


E assim você é livre pra escolher os componentes que solucionam o problema e aplicar em seu projeto.

Código JSF na página XHTML:



Código ManagedBean:



```
1 package org.primefaces.showcase.view.input;
2
3 import java.util.ArrayList;
4 import java.util.List;
5 import javax.faces.bean.ManagedBean;
6
7 @ManagedBean
8 public class InputTextareaView {
9
10     public List<String> completeArea(String query) {
11         List<String> results = new ArrayList<String>();
12
13         if(query.equals("PrimeFaces")) {
14             results.add("PrimeFaces Rocks!!!");
15             results.add("PrimeFaces has 100+ components.");
16             results.add("PrimeFaces is lightweight.");
17             results.add("PrimeFaces is easy to use.");
18             results.add("PrimeFaces is developed with passion!");
19         }
20         else {
21             for(int i = 0; i < 10; i++) {
22                 results.add(query + i);
23             }
24         }
25
26         return results;
27     }
28 }
```

## Datatable se torna super fácil com PrimeFaces

DataTable exibe dados em formato tabular.

Id	Year	Brand	Color
04c4bc8e	1984	Volkswagen	Silver
28bfeb96	1991	Honda	Silver
999b69a9	1970	Fiat	Brown
b9e9e66d	1967	Ford	Black
4686815a	1979	Volkswagen	White
bd54c64f	2009	Mercedes	Brown
c5d46dd5	1978	Volvo	Green
c9dfbfe2	1965	Ford	Green
d04837ab	1970	Fiat	Black
e19221c9	1961	Renault	Green

basic.xhtml BasicView.java CarService.java

```
1 <p:dataTable var="car" value="#{dtBasicView.cars}">
2   <p:column headerText="Id">
3     <h:outputText value="#{car.id}" />
4   </p:column>
5
6   <p:column headerText="Year">
7     <h:outputText value="#{car.year}" />
8   </p:column>
9
10  <p:column headerText="Brand">
11    <h:outputText value="#{car.brand}" />
12  </p:column>
13
14  <p:column headerText="Color">
15    <h:outputText value="#{car.color}" />
16  </p:column>
17 </p:dataTable>
```

O segredo de um datatable é ter uma lista de objetos carregados no managedBean e apontar ela no atributo value do componente dataTable.

Acima vimos como fica o JSF e abaixo vimos como fica o managedBean, e neste possui um exemplo que carrega dados estaticos em uma classe Java a parte que em nosso caso seria o banco de dados.

```
basic.xhtml BasicView.java CarService.java

1 package org.primefaces.showcase.view.data.datatable;
2
3 import java.io.Serializable;
4 import java.util.List;
5 import javax.annotation.PostConstruct;
6 import javax.faces.bean.ManagedBean;
7 import javax.faces.bean.ManagedProperty;
8 import javax.faces.bean.ViewScoped;
9 import org.primefaces.showcase.domain.Car;
10 import org.primefaces.showcase.service.CarService;
11
12 @ManagedBean(name="dtBasicView")
13 @ViewScoped
14 public class BasicView implements Serializable {
15
16     private List<Car> cars;
17
18     @ManagedProperty("#{carService}")
19     private CarService service;
20
21     @PostConstruct
22     public void init() {
23         cars = service.createCars(10);
24     }
25
26     public List<Car> getCars() {
27         return cars;
28     }
29
30     public void setService(CarService service) {
```

Abaixo nossa classe simulando o banco de dados.

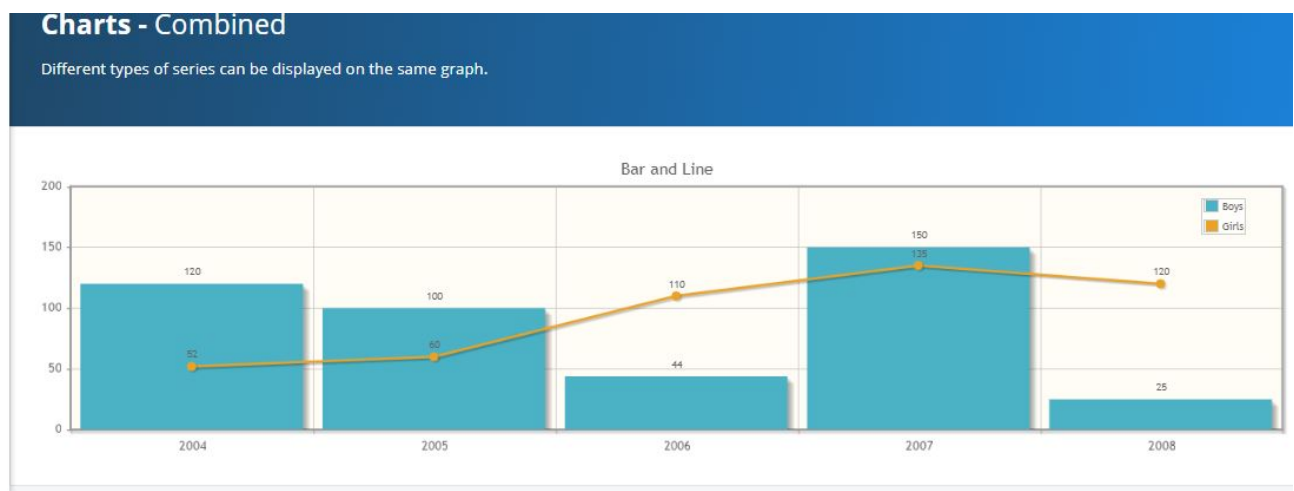
```
basic.xhtml BasicView.java CarService.java

1 package org.primefaces.showcase.service;
2
3 import java.util.ArrayList;
4 import java.util.List;
5 import java.util.UUID;
6 import javax.faces.bean.ApplicationScoped;
7 import javax.faces.bean.ManagedBean;
8 import org.primefaces.showcase.domain.Car;
9
10 @ManagedBean(name = "carService")
11 @ApplicationScoped
12 public class CarService {
13
14     private final static String[] colors;
15
16     private final static String[] brands;
17
18     static {
19         colors = new String[10];
20         colors[0] = "Black";
21         colors[1] = "White";
22         colors[2] = "Green";
23         colors[3] = "Red";
24         colors[4] = "Blue";
25         colors[5] = "Orange";
26         colors[6] = "Silver";
27         colors[7] = "Yellow";
28         colors[8] = "Brown";
29         colors[9] = "Maroon";
30
31         brands = new String[10];
32         brands[0] = "BMW";
33         brands[1] = "Mercedes";
34         brands[2] = "Volvo";
35         brands[3] = "Audi";
36         brands[4] = "Renault";
37         brands[5] = "Fiat";
38         brands[6] = "Volkswagen";
```

## Gráficos com PrimeFaces

Frameworks como o PrimeFaces tem a grande capacidade de transformar coisa complicadas em simples e isso é ótimo para nós programadores porque podemos focar na regra de negocio e em ter produtividade na entrega e na criação dos sistemas.

Vamos ver um exemplo de um grafico combinado.



A facilidade do JSF com PrimeFaces é incrível, abaixo conseguir criar um gráfico apenas com uma linha na tela JSF e na próxima imagem veremos do lado do servidor o carregamento de dados.

```
combined.xhtml | ChartView.java
```

```
1 | <p:chart type="bar" model="#{chartView.Model.combinedModel}" style="height:300px" />
```

Especificamos o tipo que é bar e o model onde é carregado a lista de dados, lembrando que para cada tipo de dados existe a estrutura correta da lista de objetos para ser carregada.



```
13 @ManagedBean
14 public class ChartView implements Serializable {
15
16     private CartesianChartModel combinedModel;
17
18     @PostConstruct
19     public void init() {
20         createCombinedModel();
21     }
22
23     public CartesianChartModel getCombinedModel() {
24         return combinedModel;
25     }
26
27     private void createCombinedModel() {
28         combinedModel = new BarChartModel();
29
30         BarChartSeries boys = new BarChartSeries();
31         boys.setLabel("Boys");
32
33         boys.set("2004", 120);
34         boys.set("2005", 100);
35         boys.set("2006", 44);
36         boys.set("2007", 150);
37         boys.set("2008", 25);
38
39         LineChartSeries girls = new LineChartSeries();
40         girls.setLabel("Girls");
41
42         girls.set("2004", 52);
43         girls.set("2005", 60);
44         girls.set("2006", 110);
45         girls.set("2007", 135);
46         girls.set("2008", 120);
47
48         combinedModel.addSeries(boys);
49         combinedModel.addSeries(girls);
50
51         combinedModel.setTitle("Bar and Line");
52         combinedModel.setLegendPosition("ne");
53         combinedModel.setMouseoverHighlight(false);
54         combinedModel.setShowDatatip(false);
55         combinedModel.setShowPointLabels(true);
56         Axis yAxis = combinedModel.getAxis(AxisType.Y);
57         yAxis.setMin(0);
58         yAxis.setMax(200);
59     }
60 }
```

Classes e objetos usados para criar a estrutura da lista para formar o gráfico são do próprio PrimeFaces então manual e o showcase se tornam de suma importância para o aprendizado.

## Capturando erros com ExceptionHandler

Todos os que codificam aplicativos da Web Java EE precisam prestar atenção ao tratamento de exceções. Quando um programa encontra um erro, os desenvolvedores podem exibir mensagens amigáveis para os usuários finais, o que aumenta sua



confiança no aplicativo. Além disso, adicionando o tratamento adequado de exceções, você pode solucionar problemas e depurar defeitos do aplicativo.

Desde a versão 2.0, a estrutura JavaServer Faces suporta um mecanismo de tratamento de exceções para fornecer um local centralizado para lidar com exceções em aplicativos JSF. Veja como você pode utilizá-lo para escrever aplicativos profissionais.

Pra isso temos opções diferentes:

- Tratar exceções com Ajax.
- Tratar exceções sem Ajax

Como já sabemos quando dá uma exceção em uma requisição Ajax temos que capturar isso na resposta e mostrar para o usuário e quando não é Ajax a página deve ser recarrega para ser mostrada o erro.

## ExceptionHandler com Ajax

Primeiramente precisamos ter um Dialog do PrimeFaces que irá abrir uma tela para o usuário e dentro dele temos que desenvolver a rotina para a captação da mensagem de erro.

```
<p:dialog id="exceptionDialog" header="Exception '#{pfExceptionHandler.type}' occurred!" widgetVar="exceptionDialog"
height="500px">
  Message: #{pfExceptionHandler.message} <br/>
  StackTrace: <h:outputText value="#{pfExceptionHandler.formattedStackTrace}" escape="false" /> <br />

  <p:button onclick="document.location.href = document.location.href;"
value="Reload!"
rendered="#{pfExceptionHandler.type == 'javax.faces.application.ViewExpiredException'}" />
</p:dialog>
```

E no botão como estamos utilizando Ajax e também não precisamos de nenhum redirecionamento vamos usar o ActionListener e passamos no atributo ajax o valor true do nosso botão e então teremos efeito em Ajax e sem redirecionar.

```
<p:commandButton actionListener="#{exceptionHandlerView.throwNullPointerException}"
ajax="true"
value="Throw NullPointerException!" />
```

E para testarmos a nossa rotina de captura de erros podemos fazer um método em nosso managedBean que lança algum erro/exceção do Java e o PrimeFaces irá se encarregar de mostrar o erro em nosso Dialog.

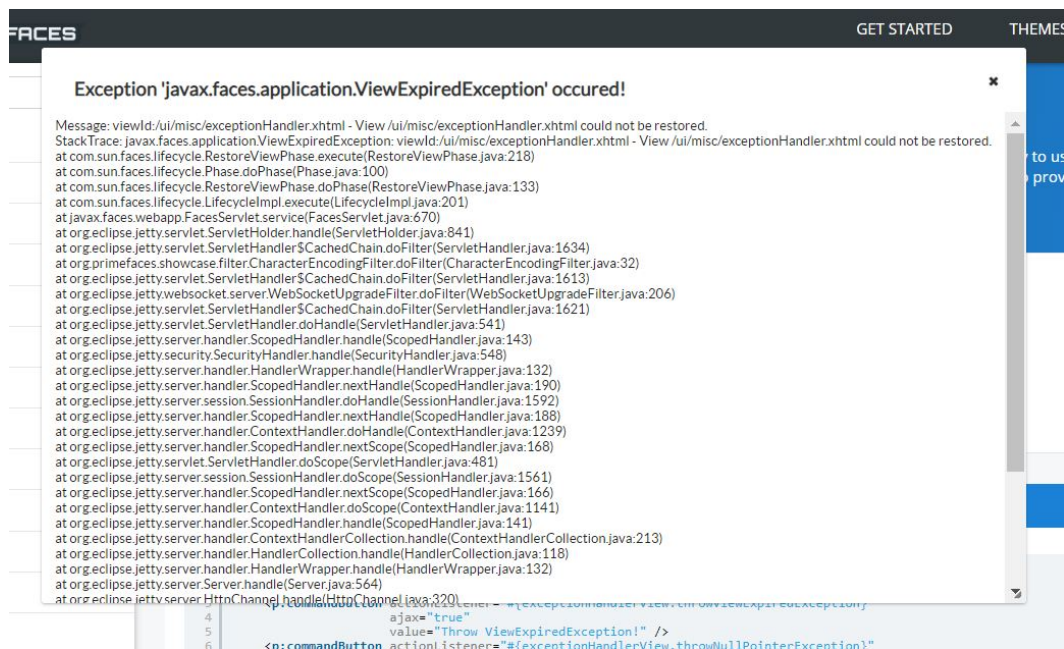
```
@ManagedBean
@RequestScoped
public class ExceptionHandlerView {

    public void throwNullPointerException() {
        throw new NullPointerException("A NullPointerException!");
    }

    public void throwWrappedIllegalStateException() {
        Throwable t = new IllegalStateException("A wrapped IllegalStateException!");
        throw new FacesException(t);
    }

    public void throwViewExpiredException() {
        throw new ViewExpiredException("A ViewExpiredException!",
            FacesContext.getCurrentInstance().getViewRoot().getViewId());
    }
}
```

O resultado que vamos ter é um Dialog com a lista de erros.

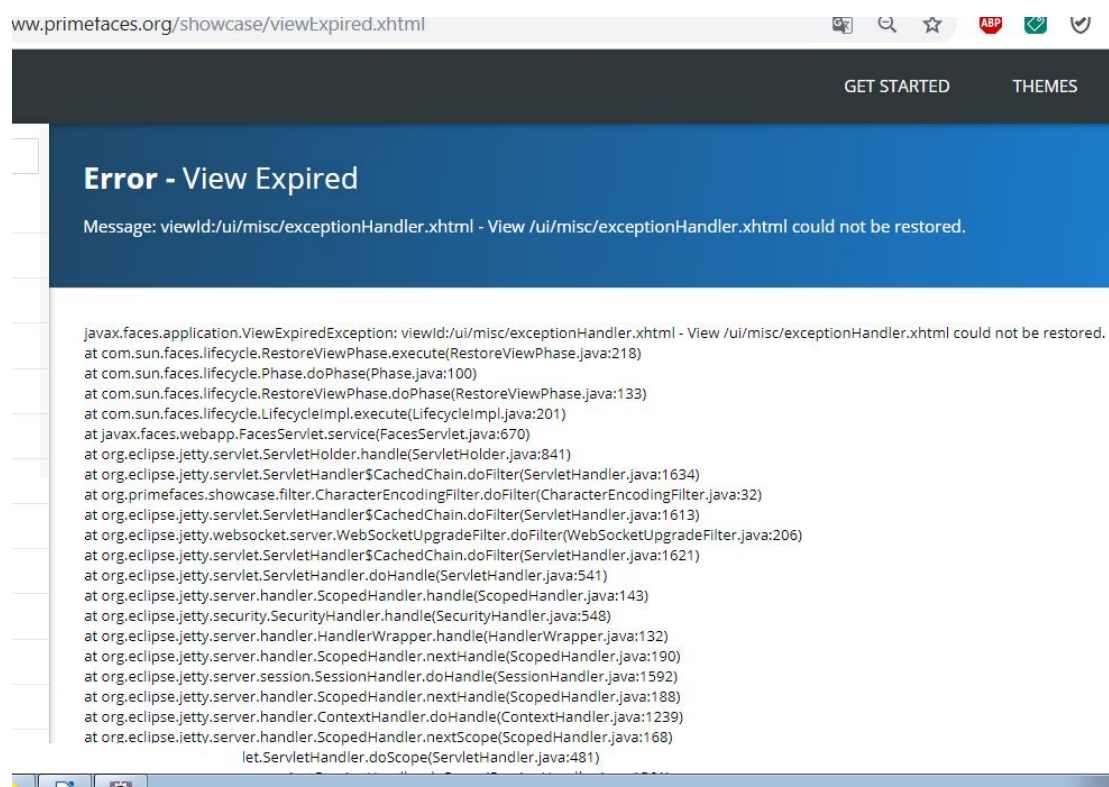


## ExceptionHandler sem Ajax

A única diferença do que fizemos acima é passar no atributo do nosso botão o Ajax para ser false, assim, o redirecionamento é feito e será exibido dentro do próprio navegador a pilha de erro.

```
<h3>Non-AJAX</h3>
<p:commandButton actionListener="#{exceptionHandlerView.throwViewExpiredException}"
  ajax="false"
  value="Throw ViewExpiredException!" />
<p:commandButton actionListener="#{exceptionHandlerView.throwNullPointerException}"
  ajax="false"
  value="Throw NullPointerException!" />
```

Assim a pilha de erro não sendo exibida dentro do Dialog e sim dentro do navegador teremos um resultado abaixo.



## Confirm Dialog

ConfirmDialog é integrado com o comportamento de confirmação e usado como um substituto para o utilitário de confirmação de javascript.

A rotina mais normal em um sistema é pedir para o usuário se ele realmente quer efetuar uma operação sim ou não.

Usando o PrimeFaces temos um opção bem legal que é p:confirm onde se o usuário clicar em sim então o método do managedBean do lado do servidor será invocado, caso contrário nada será feito.

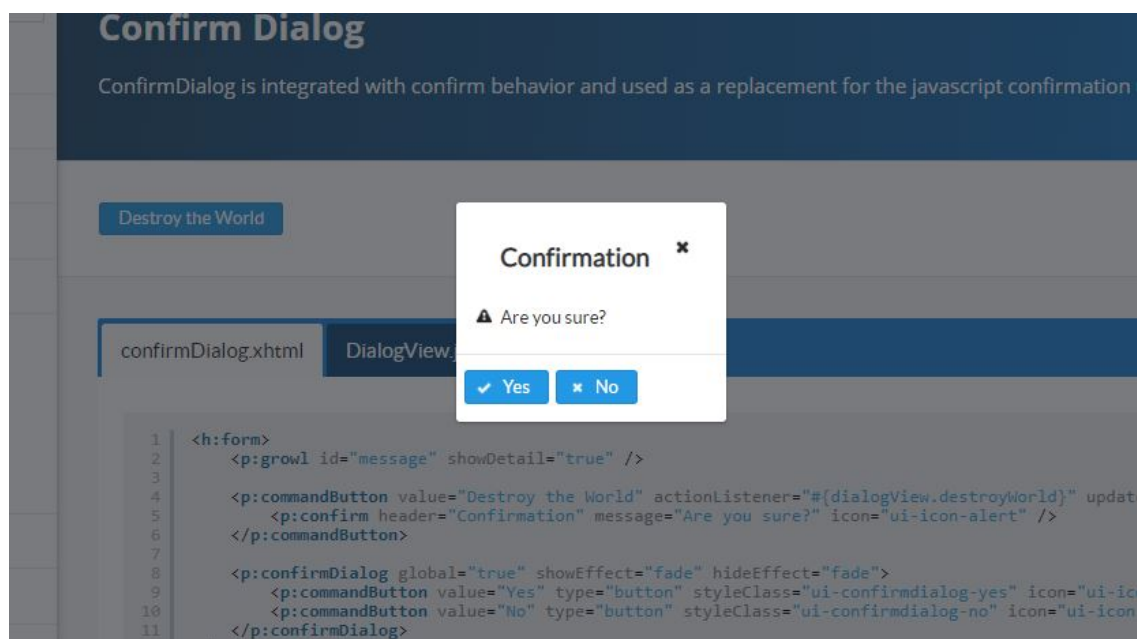
Normalmente é feito em botões de excluir ou em rotinas mais importantes como alterações de dados e exclusões importantes.

```
<h:form>
  <p:growl id="message" showDetail="true" />

  <p:commandButton value="Destroy the World" actionListener="#{dialogView.destroyWorld}" update="message">
    <p:confirm header="Confirmation" message="Are you sure?" icon="ui-icon-alert" />
  </p:commandButton>

  <p:confirmDialog global="true" showEffect="fade" hideEffect="fade">
    <p:commandButton value="Yes" type="button" styleClass="ui-confirmdialog-yes" icon="ui-icon-check" />
    <p:commandButton value="No" type="button" styleClass="ui-confirmdialog-no" icon="ui-icon-close" />
  </p:confirmDialog>
</h:form>
```

Teremos o seguinte feito, quando clicarmos no botão de excluir ou editar que assim seja depende do que você quer fazer em seu sistema.





As mensagens são customizadas e fáceis de escrever o que quisermos para melhorar o sistema.

Então clicando em não nada será feito e clicando em sim será executado o nosso metodo dentro ActionListener do commandButton.

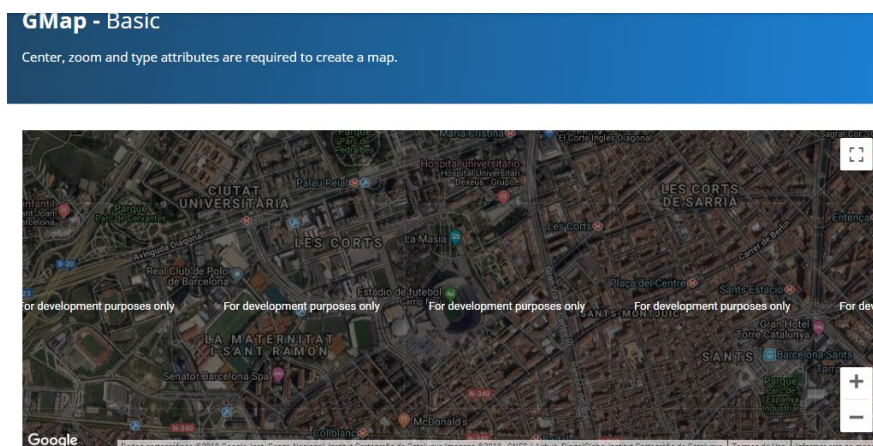
## Confirm DialogGMap – Basic

Google Maps é um serviço de pesquisa e visualização de mapas e imagens de satélite da Terra gratuito na web fornecido e desenvolvido pela empresa estadunidense Google. Atualmente, o serviço disponibiliza mapas e rotas para qualquer ponto nos Estados Unidos, Canadá, na União Europeia, Austrália e Brasil, entre outros.

Mas como implementar isso em nosso sistema? A resposta é fácil até demais, como já sabemos a muitos anos lá em nossas aulas de geografia que as localizações são por latitude e longitude, então, a forma mais simples de implantar isso em nosso sistema é passando essas localizações para o componente do PrimeFaces.

```
basic.xhtml
1 | <p:gmap center="41.381542, 2.122893" zoom="15" type="HYBRID" style="width:100%;height:400px" />
```

Então teremos o seguinte resultado abaixo.



## Barcode

O componente de código de barras gera vários tipos de códigos de barras nos formatos PNG ou SVG. Os navegadores sem suporte a SVG recuam automaticamente para o PNG.

Códigos de barrar são muito usados em quaisquer sistemas comercial onde não precisa digitar e podem ser lidos com leitores. Parece complicado mas quando temos um Frameworks que faz tudo por nós a única coisa que precisamos passar pra ele é o valor do código de barras e o tipo a ser gerado e pronto está feito o código para ser impresso e lido.

Na página JSF temos o seguinte código do Primefaces.

```
<h:outputText value="Interleaved 2 of 5" />  
<p:barcode value="0123456789" type="int2of5" />
```

E o resultado em tela é o seguinte.



Aqui neste exemplo estamos passando o valor fixo mas para tornar isso dinâmico basta trazer do banco de dados em um objeto controlado com o managedBean e pronto temos os códigos de barras dinâmicos.

## Chamadas de Ajax periódicas

Um recurso muito mas muito bacana mesmo e muito utilizado e executar rotinas em determinado tempo em um sistema e é claro que o PrimeFaces iria nos surpreender cada vez mais.

O `p:poll` faz chamadas Ajax com intervalos de tempo em nossos sistemas, então vamos supor que queríamos mostrar no canto dos sistemas quantas mensagens não

lidas o usuário tem pra isso essa seria a solução para quem está criando projetos com PrimeFaces e JSF.

Vou dar um exemplo bem simples que é incrementar um numero automaticamente do lado do servidor e fazendo apenas isso sabemos que podemos processar qualquer coisa por vai complexa que seja e retornar na tela o resultado.

Então em um exemplo simples na tela JSF usando o p:poll teríamos o código simples abaixo.

```
1 <h:form>
2   <h:outputText id="txt_count" value="#{counterView.number}" />
3   <p:poll interval="3" listener="#{counterView.increment}" update="txt_count" />
4 </h:form>
```

Do lado do servidor no managedBean teríamos o seguinte código.

```
@ManagedBean
@ViewScoped
public class CounterView implements Serializable {

    private int number;

    public int getNumber() {
        return number;
    }

    public void increment() {
        number++;
    }
}
```

## Growl do PrimeFaces

O p:growl é uma forma elegante de mostrar mensagens em um sistema para o usuário.

Na página JSF ficaria o seguinte código.

```
<p:growl id="growl" showDetail="true" sticky="true" />
```

Em nosso managedBean seria o seguinte código.



```
public void saveMessage() {  
    FacesContext context = FacesContext.getCurrentInstance();  
  
    context.addMessage(null, new FacesMessage("Successful", "Your message: " + message) );  
    context.addMessage(null, new FacesMessage("Second Message", "Additional Message Detail"));  
}
```

## Download do manual do PrimeFaces

Atualmente no momento que estou escrevendo esse super e-book o PrimeFaces se encontra na versão 6.2 para baixar o e-book completo passo a passo de todo os componentes para criar aplicações acesse o link [CLIQUE AQUI](#)

Na página oficial selecione a versão do manual para baixar o PDF oficial.



### PrimeFaces User's Guide

#### User Guides

- [6.2](#)
- [6.1](#)
- [6.0](#)
- [5.3](#)
- [5.2](#)
- [5.1](#)
- [5.0](#)
- [4.0](#)
- [3.5](#)
- [3.4](#)
- [3.3](#)

## Referências

<https://www.primefaces.org/documentation/>

<https://www.primefaces.org/showcase/>

<https://www.tutorialspoint.com/jsf/>

<https://www.javatpoint.com/jsf-tutorial>