



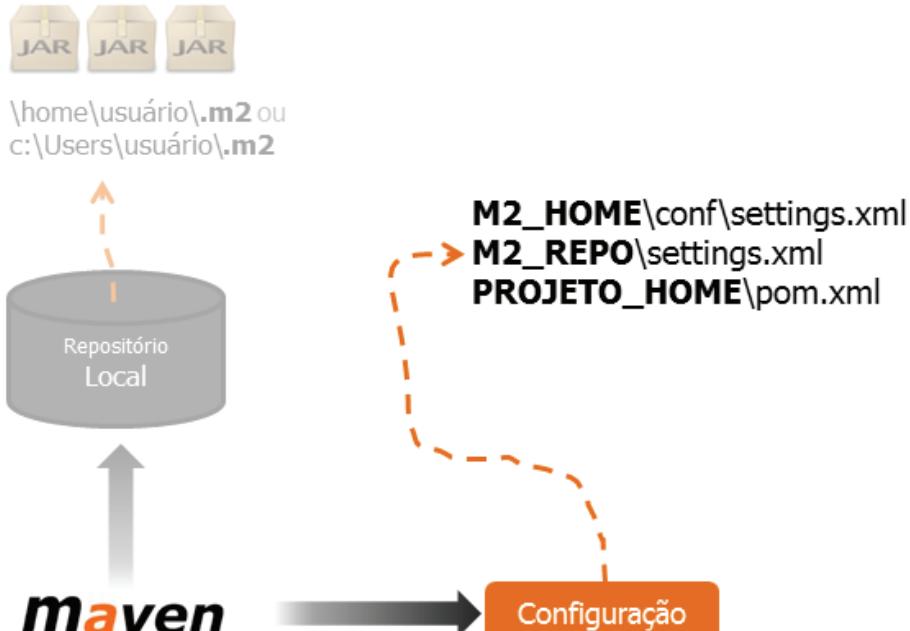
Frameworks



Maven	4
0 que é maven?	5
Project object model - pom.xml	7
Modularização	9
JSF - Java Server Faces	10
0 que é JSF?	11
O Padrão MVC no JSF	12
Ciclo de vida do JSF	13
JSF - Taglib	15
ManagedBean	16
Escopos de Aplicação	17
Conversores	18
Validadores	20
Navegação	22
Facelets	24
Internacionalização	26
Frameworks JSF	28
Primefaces	30
Arquitetura	32
Componentes	33
Primefaces - Mobile	36
Quem usa?	37
Injeção de Dependência e Contextos - CDI	39
0 que é e para que serve a CDI?	40
Injeção de Dependências	42
0 que é um “bean”?	44
Escopos do CDI	46
Qualificadores	48
Interceptadores	49
Produtores	51
CDI e JSF	52

JPA - Java Persistence API Hibernate	53
Mapeamento Objeto Relacional	55
Arquitetura do Hibernate	57
Mapeamento utilizando Anotações	59
Associações	61
Entity Manager	63
JPQL	65
Criteria	66

Frameworks Maven

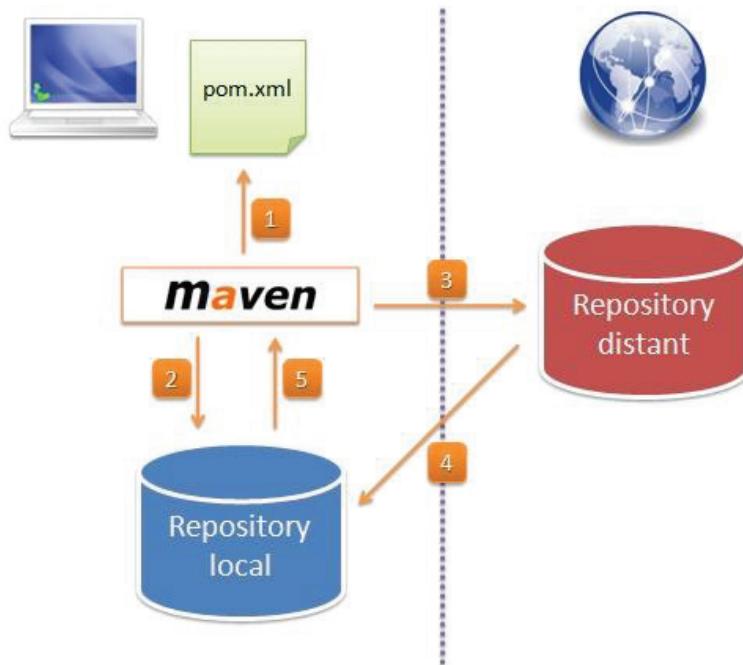


Frameworks

A construção de um software atualmente não é mais simplesmente construir um projeto simples, monolítico e individual. Cada dia novos requisitos são necessários para se criar as chamadas “soluções corporativas”: segurança, robustez, acessibilidade, integração, etc.

Manter o controle do projeto, de forma que seja possível criar uma versão do software que seja consistente e padronizada é um desafio e por isto, várias ferramentas foram criadas: make, Ant, build, entre outras. O Apache Maven, com seu sistema de construção de software, é a evolução deste tipo de ferramenta através da integração com a internet.

O que é maven?



Frameworks

O Maven é um projeto de código livre, mantido pela Apache Software Foundation, criado originalmente para gerenciar o complexo processo de criação do projeto Jakarta Turbine. Desde seu início, o Maven tem sido utilizado por projetos de código livre e também proprietário. O Maven passou de uma ferramenta de construção para uma ferramenta complexa de gestão de construção de software, com várias funcionalidades e aplicável a maioria dos cenários de desenvolvimento de software.

Para compreender o processo de construção de software com o Maven, é importante entender um conjunto de modelos conceituais que explicam como as coisas funcionam.

- **Project object model (POM):** O POM é o componente principal para o Maven. Parte deste modelo faz parte do mecanismo interno do Maven. Outra parte é informada por um arquivo pom.xml editado por você.

- **Dependency management model:** A gestão de dependência é uma parte do processo de construção de software que muitas vezes é ignorada em projetos, mas que é fundamental. O Maven é reconhecidamente uma das melhores ferramentas para a construção de projetos complexos, especialmente pela sua robustez na gestão de dependências.

- **Ciclo de vida de construção e fases:** Associado ao POM, existe a noção de ciclo de vida de construção e fases. É através deste modelo que o Maven cria uma “ponte” entre os modelos conceituais e os modelos físicos. Quando se usam plug-ins, é por meio deste componente que tais plug-ins são gerenciados e passam a seguir um ciclo de vida bem definido.

- **Plug-ins:** estendem as funcionalidades do Maven.

No desenvolvimento de projetos de software, o desenvolvedor encontrará algumas perguntas como:

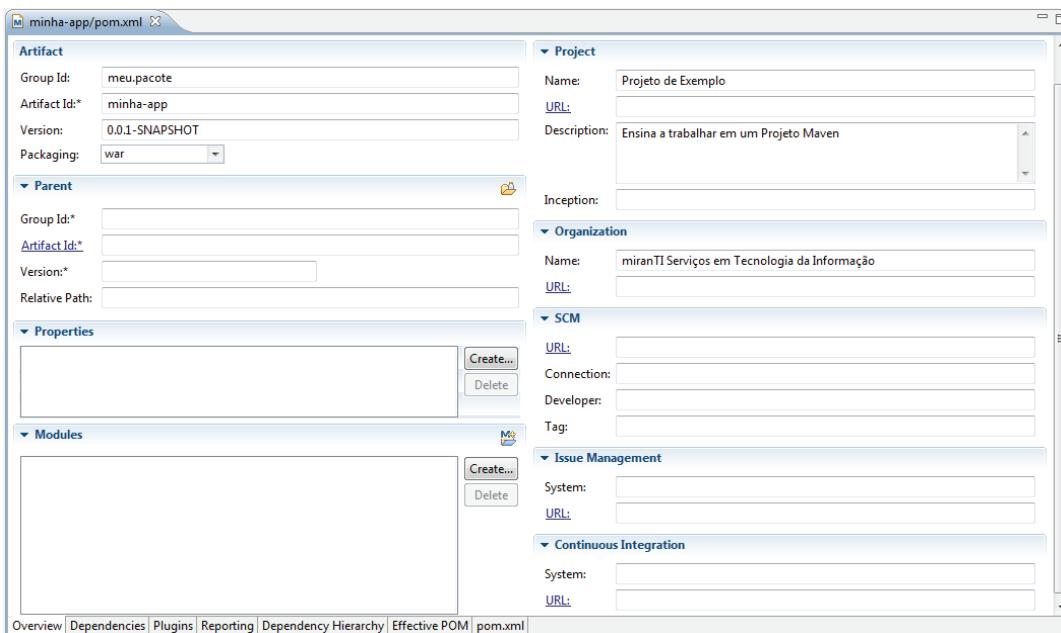
- Qual deve ser a estrutura de diretórios para organizar o projeto?
- Como devem ser estruturados os diretórios de forma a organizar o código fonte, código de teste, bibliotecas, configuração, documentação e relatórios do projeto?
- De onde as bibliotecas Java (JARS) deverão ser baixados?
- Quais versões das bibliotecas (JARS) deverão ser utilizadas?
- Qual a melhor forma de resolver conflitos entre bibliotecas?
- Como manter o projeto com as versões mais recentes das bibliotecas?
- Como as bibliotecas e recursos de compilação, execução e teste devem ser separados?
- Existe alguma forma de executar todos os testes durante o processo de construção e obter um percentual de cobertura dos testes?
 - É possível executar testes para mensurar a qualidade do código ou medir desempenho durante o processo de construção?
 - É possível executar testes de integração durante o processo de construção?

O Maven busca resolver estas questões fornecendo um modelo de construção que pode ser reutilizado por todos os projetos de software. O Maven abstrai a estrutura do projeto e seu conteúdo, seguindo os princípios da “convenção sobre a configuração”, “execução declarativa do processo de ciclo de vida do desenvolvimento”, “reuso da lógica de construção entre projetos” e “organização lógica das dependências dos projetos”.

Em resumo, o Maven:

- Define como o projeto é tipicamente construído.
- Utiliza convenções para facilitar a configuração do projeto e assim, sua construção.
- Ajuda os usuários a compreender e organizar melhor a complexa estrutura dos projetos e suas variações.
- Prescreve e força a utilização de um sistema de gerenciamento de dependências comprovadamente eficaz, permitindo que times de projeto em locais diferentes compartilhem bibliotecas.
- É flexível para usuários avançados, permitindo que definições globais sejam redefinidas e adaptadas de forma declarativa (alterando-se a configuração, alterando metadados ou através da criação de plug-ins).
- É extensível.
- Está em constante evolução, incorporando novas práticas e funcionalidades identificadas como comuns em comunidades de usuários.

Project object model - pom.xml



Frameworks

O Modelo de Objetos de Projeto (POM) define as informações necessárias para que o Maven possa executar um conjunto de metas e que a construção do software possa ser realizada. A representação do POM é feita através de um arquivo XML chamado pom.xml. No conceito do Maven, o projeto é um conceito que vai além de um conjunto de arquivos. O projeto contém arquivos de configuração, uma lista de desenvolvedores que atuam em papéis, um sistema de controle de ocorrências (issues), informações sobre a empresa e licenças, a URL onde o projeto reside, as dependências e todo o código que dá vida ao projeto. De fato, para o Maven, um projeto não precisa ter nada mais do que um simples arquivo pom.xml.

POM (Project Object Model), como já foi dito, é formado por dados internos ao Maven e por um ou mais arquivos **pom.xml**. Os arquivos pom.xml formam uma espécie de hierarquia, com cada um herdando os atributos do seu pai. O próprio Maven provê um POM que está no topo da hierarquia e portanto define os valores *default* para todos os projetos.

As dependências são especificadas como parte do arquivo pom.xml. O Maven identifica as dependências do projeto de acordo com seu modelo de gerenciamento de dependências. O Maven procura por componentes dependentes (chamados **artefatos** na terminologia do Maven) no **repositório local** e em **repositórios globais**.

Artefatos encontrados em repositórios remotos são baixados para o repositório local para um acesso posterior mais eficiente. O **mecanismo de resolução de dependências** do Maven pode identificar **dependências transitivas** (A depende de B que depende de C, logo, A depende de C).

Plug-ins são configurados e descritos no arquivo pom.xml. O plug-in propriamente dito é gerenciado como um artefato pelo sistema de gerenciamento de dependências e baixados quando eles são necessários para realizar alguma atividade de construção.

Cada plug-in pode ser associado a várias fases do ciclo de vida. O Maven mantém uma máquina de estados que transita pelo ciclo de vida e dispara os plug-ins quando necessário.

Modularização

Aplicação

Aplicação-Modelo

Aplicação-Controle

Aplicação-Visão



Frameworks

Trabalhamos com um projeto só e relativamente fácil utilizar o Maven. Entretanto, em muitos casos, será necessário trabalhar com um conjunto de diferentes projetos, a fim de se manter uma boa modularização. Por exemplo, podemos ter bibliotecas separadas das aplicações web ou mesmo duas ou mais aplicações web. Organizar tudo isto, onde temos várias dependências intra-projetos e com artefatos externos, pode ser muito complicado. Felizmente, o Maven foi projetado para simplificar o trabalho.

Quando trabalhamos com vários projetos ou módulos de código que formam um projeto de desenvolvimento de software, precisamos criar um projeto principal para que o Maven possa construir toda a aplicação. O projeto principal é formado simplesmente pelo arquivo pom.xml. Geralmente, este arquivo estará no diretório que contém os sub-diretórios para os módulos.

Frameworks

JSF – Java Server Faces

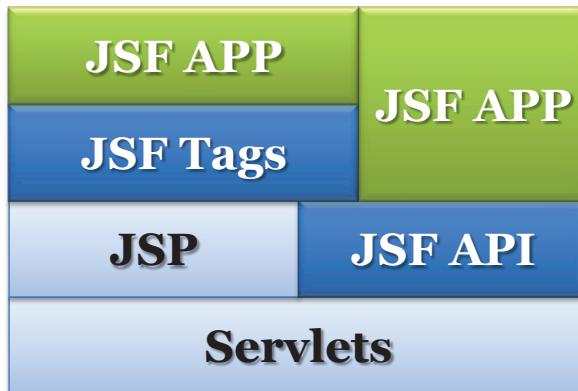


Frameworks

As tecnologias voltadas para o desenvolvimento de aplicações WEB têm mudado constantemente. Inicialmente os sites e/ou portais possuíam apenas conteúdo estático, com a evolução das tecnologias passaram a oferecer páginas com conteúdos dinâmicos e personalizados. Diversas tecnologias estão envolvidas no desenvolvimento das aplicações WEB como, por exemplo, CGI (Common Gateway Interface), Servlets e JSP (Java Server Pages).

Com a grande utilização dos design patterns, principalmente no “mundo Java”, começaram a surgir diversos frameworks para auxiliar no desenvolvimento de aplicações WEB. A tecnologia JavaServer Faces foi criada para mudar a forma de desenvolver aplicações Web com Java. Neste modulo tem por objetivo a apresentação desta tecnologia, falamos sobre o que é e como funciona JSF e apresentamos os principais componentes da página JSF.

O que é JSF?



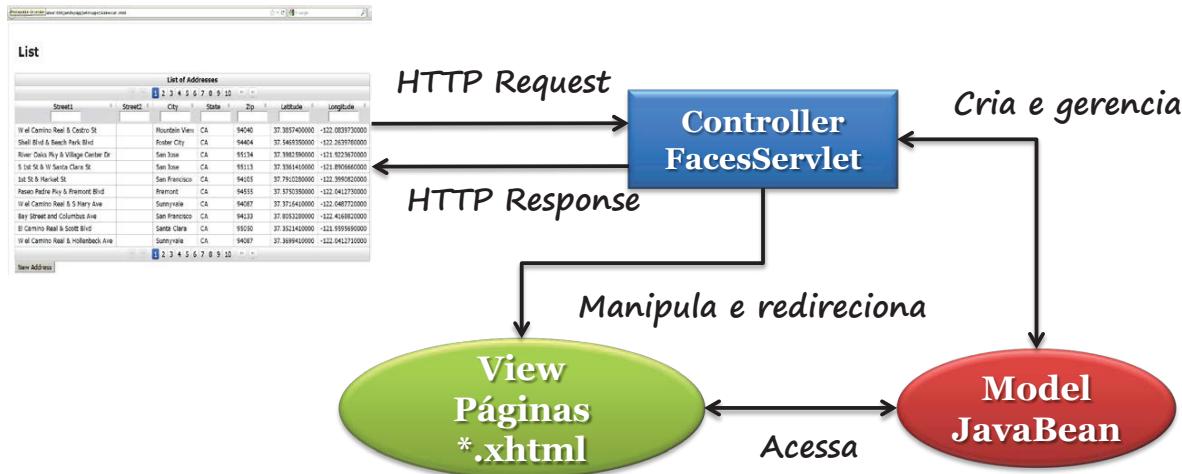
Frameworks

JSF é mais do que um framework e uma especificação Java que incorpora características de um framework MVC para WEB e de um modelo de interfaces gráficas baseado em eventos. Por basear-se no padrão de projeto MVC, uma de suas melhores vantagens é a clara separação entre a visualização e regras de negócio.

O JSF oferece facilidade de uso das seguintes formas:

- Separação entre as camadas de apresentação e de aplicação
- É uma especificação que faz parte do *Java Enterprise Edition* (Java EE). Por ser uma especificação, ele é mantido dentro do *Java Community Process* (JCP).
- Facilita a construção de uma Interface usando um conjunto de componentes reutilizáveis
- Ajuda a gerenciar o estado da Interface nas solicitações do servidor
- Oferece um modelo simples para conectar os eventos gerados pelo cliente ao código da aplicação do servidor
- Permite personalizar os componentes de Interface para que sejam facilmente construídos e reutilizados

O Padrão MVC no JSF



Frameworks

No JavaServer Faces, o controle fica por conta de um servlet chamado Faces Servlet, por arquivos de configuração (ex.: faces-config.xml), pelos Backing Beans e pelos validadores e conversores. O *FacesServlet* é responsável por receber requisições da WEB, redirecioná-las para o modelo e então remeter uma resposta. Os arquivos de configuração são responsáveis por definirem a navegação entre páginas e o mapeamento de ações. Já os validadores e conversores permitem um maior controle sobre os dados que serão enviados.

O modelo representa os objetos de negócio e executa uma lógica de negócio ao receber os dados vindos da camada de visualização. Finalmente, a visualização é composta por *component trees* (hierarquia de componentes UI), tornando possível unir um componente ao outro para formar interfaces mais complexas.

Visão:

- Componentes de Interfaces em páginas JSP/XHTML
- Kits renderizadores (HTML, XML, etc.)

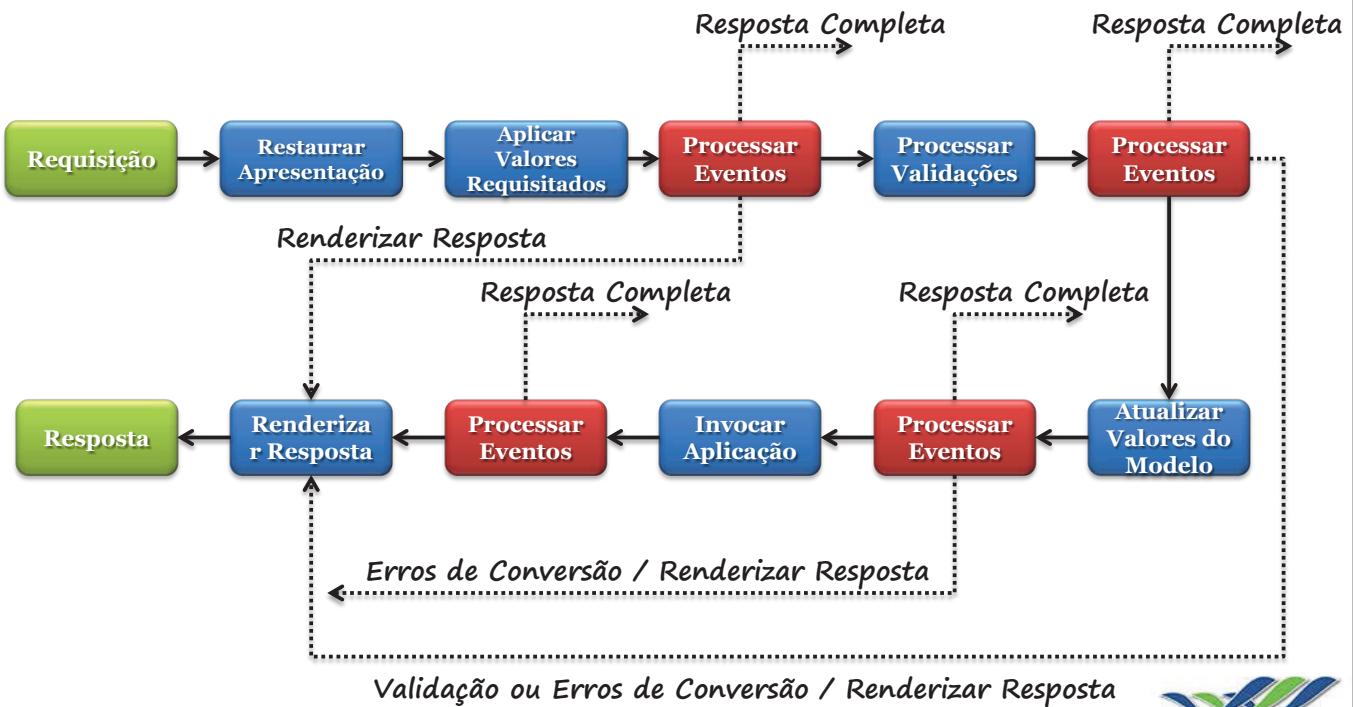
Controlador:

- Faces Servlet (Front Controller)
- Backing Bean (Page Controller ou Modelo)

Modelo:

- Entidades e regras de negócio
- Java Beans

Ciclo de vida do JSF



Frameworks

Muitos programadores não entendem todo o ciclo de vida do JSF, claro que é possível se programar sem conhecer o ciclo de vida, mas muitas das dúvidas não existiriam se o programador entendesse sobre o ciclo de vida (Informação nunca é demais).

Restaurar a apresentação

Essa fase começa toda vez que uma requisição de uma página JSF é feita através de um link ou botão clicado. O Controller(Faces Context) examina o request e extrai o ID da visão (tela, view) que é determinado pelo nome da página JSP. O JSF utiliza esse ID para localizar os componentes para a tela atual. Se a visão ainda não existe ela será criada pelo framework, se ela já existir será apenas reutilizada pelo controller. A visão possui os componentes GUI's (Graphical User Interface). Esta fase possui 3 instâncias: nova visão, visão inicial e o retorno (postback) da visão. Cada instância dessa é tratada de forma diferente.

Na nova visão, o JSF constroi a visão para o Faces e vincula esta página aos handlers e validators. A visão é salva no FacesContext como um objeto. O FacesContext contém todas as informações de como se deverá manipular este GUI para a requisição corrente. O FacesContext guarda as visões na propriedade ViewRoot o qual possui todos os componentes da visão atual.

Na visão inicial, o JSF cria uma visão em branco. Esta visão será populada de acordo com o evento solicitado pelo usuário. Da visão inicial o JSF avança diretamente para a fase de Render Response.

No caso do postback (o usuário retornou a página que acessou anteriormente), a visão já existe sendo necessário apenas ser restaurada.

Aplicar valores requisitados

O propósito desta fase é fazer com que cada componente recupere seu estado corrente. O Componente deve primeiro ser criado ou recuperado a partir do FacesContext, seguido por seus valores. Os valores dos componentes são geralmente recuperados dos parâmetros de request.

Se um componente não está com seu atributo **Immediate** setado para True, o valor será apenas convertido. Se o campo for Inteiro, o valor será convertido para um inteiro. Se a conversão falhar, será adicionado no FacesContext uma mensagem de erro que será exibida durante a fase de Render response.

Se um componente estiver com seu atributo **immediate** setado para true, o valor será convertido para o tipo específico do atributo e logo após será validado, o valor convertido será guardado no componente.

É nesta fase onde ocorrem as conversões de valores, ou seja, onde os Converter's criados pelo usuário serão utilizados ou os conversores padrão do JSF.

Processar validações

Enesta fase onde ocorrem as validações. Esta validação pode ser criada pelo desenvolvedor ou “obtida” diretamente do JSF. Os valores são validados de acordo com as regras de validação da aplicação. Se um valor estiver errado, é gerado uma mensagem de erro e adicionada no FacesContext e o componente é marcado como inválido. Se algum componente estiver marcado como inválido, o JSF avançará automaticamente para a fase de Render Response, que irá exibir o erro de validação.

Se não houver nenhum erro de validação, o JSF irá avançar para a fase de Atualizar valores do modelo.

Atualizar valores do modelo

Nessa fase, os valores já estão validados e registrados nos componentes que serão atribuídos à respectiva propriedade na classe Bean. Esse processo envolve conversores dos tipos de dados, como por exemplo, conversores personalizados ou data e número.

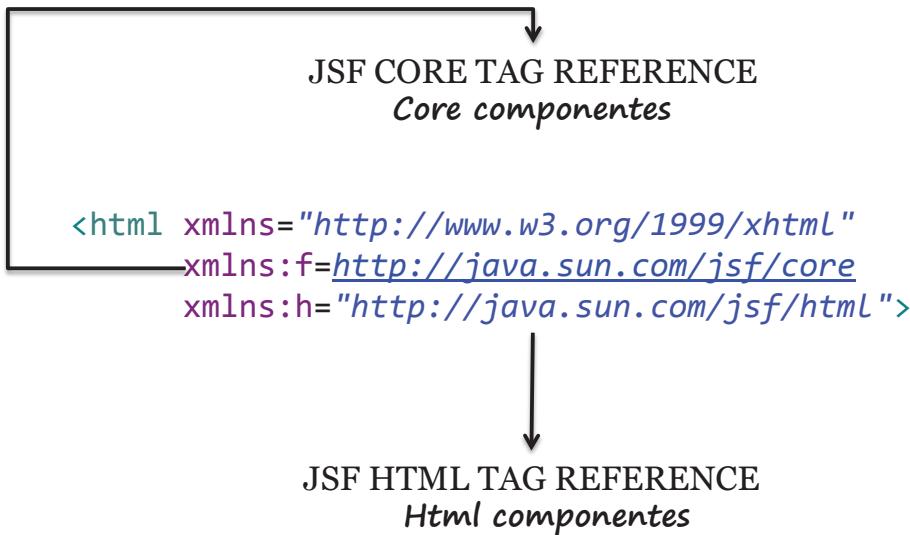
Invocar a aplicação

Nessa fase, o JSF manipula qualquer nível de evento da aplicação, desde o envio de um formulário ou chamada para outra página através de um link. Depois que todos os valores já estarem validados, convertidos e atribuídos para as propriedades da classe Bean, se for o caso, o JSF adicionará o método da classe Bean que adicionou a requisição.

Renderizar resposta

Por ser a fase final, é exigida que quando a página for construída e devolvida para o browser, o JSF solicite que cada componente de tela que têm suas propriedades, comportamentos e forma, faça a geração do próprio HTML.

JSF - Taglib



Frameworks

Taglib é usado para informar que utilizaremos na página algumas tags **JSTL** específicas do framework JSF. JSTL são bibliotecas, conjuntos de códigos personalizados, para serem utilizados dentro das páginas. Na taglib informamos a propriedade **prefix**, que é um apelido para utilizarmos esta JSTL dentro da página.

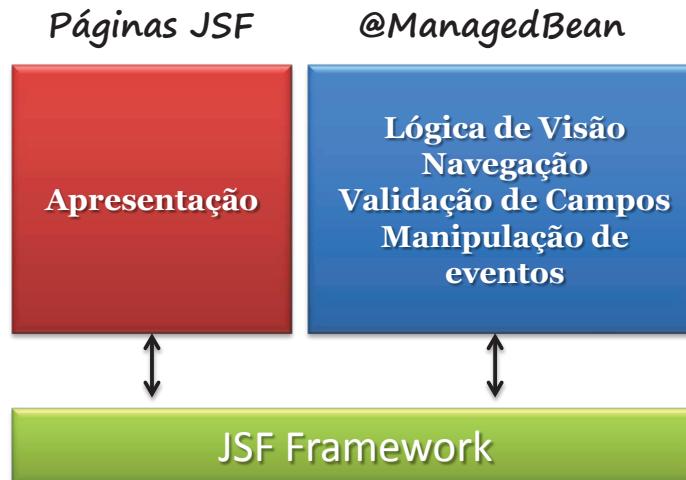
JSF Core Tag Library

A biblioteca de tags Core JSF contém tags, em geral para a conversão, validação, Ajax e outros casos de uso. Veja em: <http://www.jsftoolbox.com/documentation/help/12-TagReference/core/index.jsf>.

JSF HTML Tag Library

Esta biblioteca JSF que contém tags de componentes para todas as combinações UIComponent + HTML RenderKit Renderer definidas na especificação JavaServer Faces. Veja em <http://www.jsftoolbox.com/documentation/help/12-TagReference/html/index.jsf>.

ManagedBean



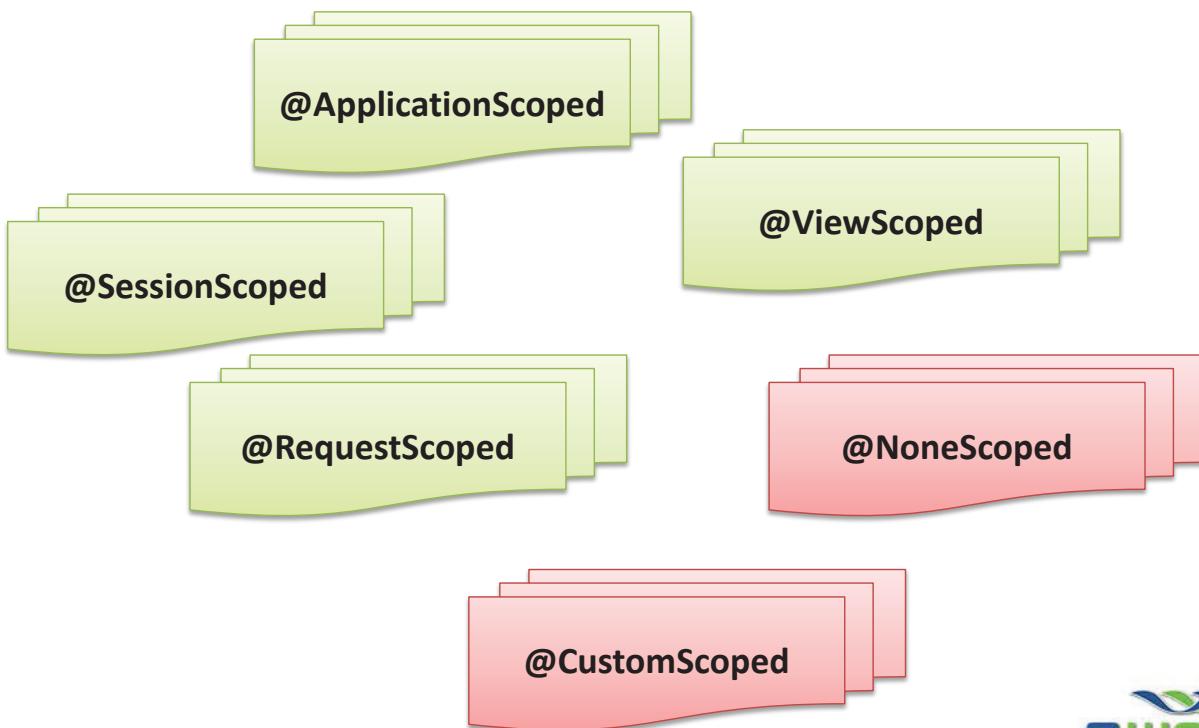
Frameworks

Managed Beans são classes Java que são configuradas para poderem interagir com páginas JSF intermediando a comunicação entre nossas páginas e o nosso model. A grande vantagem dos Managed Beans é que eles não precisam herdar nenhuma classe nem implementar alguma interface.

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

Disponível a partir da versão 2.0 do JSF a anotação **@ManagedBean** é utilizada para podermos indicar que a classe é um Managed Bean, nela temos duas propriedades, **name** que indica o nome do nosso Managed Bean (quando não informado essa propriedade o nosso Managed Bean recebe o nome da nossa classe com o primeiro caracter minúsculo) e a propriedade **eager** que quando o seu valor for **true** e o scopo do nosso ManagedBean for de **Aplicação** indica que este bean deve ser inicializado quando a aplicação for iniciada.

Escopos de Aplicação



Frameworks

Os escopos dos nossos Managed Beans também iremos declará-los via anotações, se no nosso Managed Bean não colocarmos nenhuma declaração de escopo é atribuido por default o **scopo de requisição** (RequestScoped).

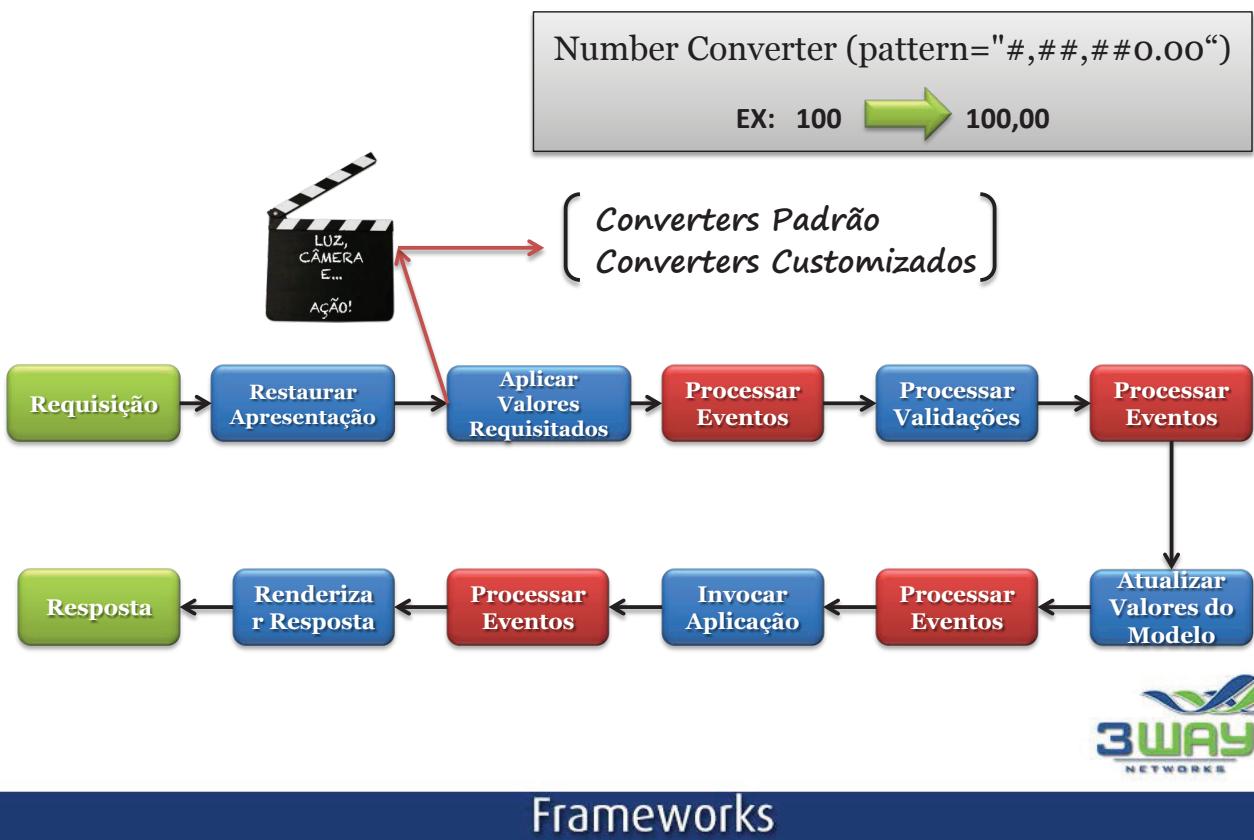
Abaixo os escopos permitidos para os nossos Managed Beans:

- **@ApplicationScoped** - Escopo de aplicação, permanece existente entre todas as interações de todos os usuários, ou seja possui vida útil durante todo o tempo em que a aplicação estiver funcionando.
- **@SessionScoped** - Escopo de sessão, permanece ativo em várias requisições HTTP enquanto durar a sessão do usuário.
- **@RequestScoped** - Escopo de requisição. Existe durante uma única solicitação/requisição HTTP.
- **@ViewScoped** - Escopo de página, persiste durante a interação do usuário com uma única página. Enquanto a página existir uma instância do Managed Bean existirá.

Outras anotações também estão disponíveis, porém são utilizadas com menor frequência, são elas:

- **@NoneScoped** - Indica que o escopo não está definido para a aplicação.
- **@CustomScoped** - Indica que o escopo do nosso Managed Bean terá um escopo personalizado, criado pelo desenvolvedor.

Conversores



Converter é o componente do JSF que é responsável por fazer a conversão de um objeto para uma String e de uma String para um objeto.

Ele atua entre a página JSF e o Managed Bean, convertendo textos da página automaticamente em objetos no Managed Bean. Desta forma, nós definimos como a conversão deve ocorrer e depois disso, trabalhamos como se estivéssemos acessando o próprio objeto na página JSF.

Esta abordagem é especialmente útil quando temos um objeto que possui uma conversão padrão. Logo, podemos implementar no componente como esta conversão ocorre, e apenas reutilizá-la. Trabalhamos mais orientado a objetos também, sem precisar acessar o conteúdo do objeto o tempo inteiro na hora de construí-lo.

Conversores Padrão

Aqui vamos ver os diferentes conversores padrão que vem com JSF Implementação junto com alguns trechos de código de amostra.

A seguir estão os conversores padrão disponíveis no JSF:

- **BigDecimalConverter** - Usado para converter os valores da sequencia de entrada do usuário em valores do tipo `java.math.BigDecimal`
- **BigIntegerConverter** - Usado para converter os valores da sequencia de entrada do usuário em valores do tipo `java.math.BigInteger`
- **BooleanConverter** - Usado para converter os valores da sequencia de entrada do usuário em valores do tipo `java.lang.Boolean`

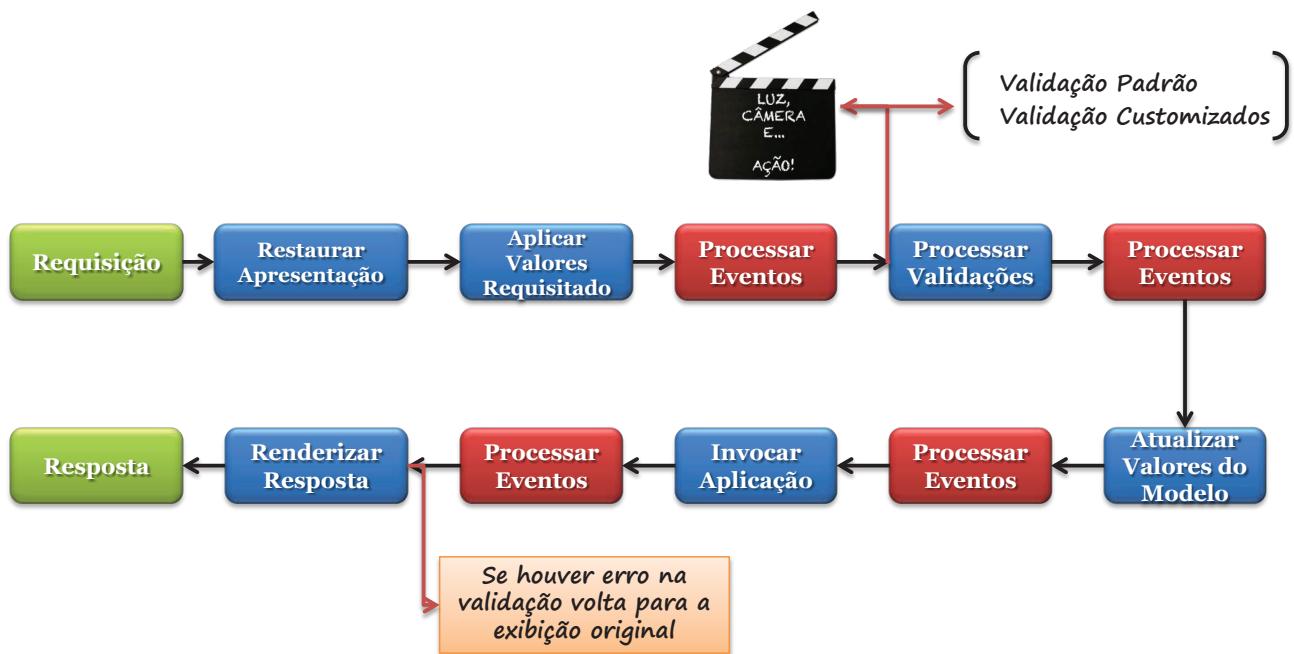
- **ByteConverter** - Usado para converter os valores da sequencia de entrada do usuário em valores do tipo java.lang.Byte
- **CharacterConverter** - Usado para converter os valores da sequencia de entrada do usuário em valores do tipo java.lang.Character
- **DateTimeConverter** - Usado para converter os valores da sequencia de entrada do usuário em valores do tipo java.util.Date com um formato padrão.
- **DoubleConverter** - Usado para converter os valores da sequencia de entrada do usuário em valores do tipo java.lang.Double
- **EnumConverter** - Usado para converter os valores da sequencia de entrada do usuário em valores do tipo java.lang.Enum
- **FloatConverter** - Usado para converter os valores da sequencia de entrada do usuário em valores do tipo java.lang.Float
- **IntegerConverter** - Usado para converter os valores da sequencia de entrada do usuário em valores do tipo java.lang.Integer
- **LongConverter** - Usado para converter os valores da sequencia de entrada do usuário em valores do tipo java.lang.Long
- **NumberConverter** - Usado para converter os valores da sequencia de entrada do usuário em valores do tipo java.lang.Number
- **ShortConverter** - Usado para converter os valores da sequencia de entrada do usuário em valores do tipo java.lang.Short

Mas para os objetos que nós mesmo criamos, como que o JSF faz esta conversão? Por default, o método `toString()` é chamado. Sobre-escrever este método é uma solução para enviar uma informação como String para a tela. Mas e se precisarmos do inverso também, String – objeto? Ai é que entra o converter. Converter é uma interface que temos que implementar e na qual dizemos como nosso Objeto é convertido para uma String e vice-versa.

Para fazermos nosso converter temos que executar alguns passos:

- Criar uma classe que implemente a interface Converter
- Registrar o converter no JSF

Validadores



Frameworks

Em JSF você podemos trabalhar com dois grupos de validadores de dados: os padrões (que são fornecidos pelo JSF) e os personalizados (que são os podemos criar). Os existentes validam algumas características de dados, tais como: tamanho de campos, limites, etc.

A Validação

Vamos imaginar que no nosso sistema exista um requisito exigindo que o campo “idade” não possa receber valores negativos. Algo natural, já que ninguém pode ter “-43” anos. Por fins didáticos, vou desconsiderar a possibilidade de usar uma máscara/validação javascript no próprio browser. Não seria nada absurdo usar javascript, mas me ajudem aqui e vamos fingir que essa seria uma validação complexa demais para usar javascript.

Nos resta então duas abordagens principais para validar a idade:

- No método “salvar” dentro do *Managed Bean*. (menos recomendado)
- Usar um *validator* JSF. (recomendado)

Mensagens de erro

JSF fornece marcas diferentes de manusear e exibir mensagens sobre a vista. Há duas tags de mensagens na implementação de biblioteca JSF HTML de referência:

- **<h:message />** - h:message exibe a mensagem anexada a um componente . Um atributo **for= “idDoAtributo”** pode ser usado para especificar o ID de um componente cujas mensagens de erro que precisa exibir. h: message é usado para exibir mensagem de erro ao lado do componente que gerou o erro Se mais de uma mensagem é anexado a esse

componente , h:message será exibida a última mensagem.

- <**h:messages** /> - h:messages é usado para exibir todas as mensagens de uma só vez . Você pode colocar h:messages tag no início do seu formulário.

Validações Padrão

Implementação de referência do JSF SUN fornece alguns componentes de validação padrão que podem ser aproveitados para implementar a validação de qualquer entrada do usuário. A biblioteca central do JSF fornece tags para validar a entrada . A seguir estão algumas marcas que podem ser utilizados para validar a entrada .

- **f:validateDoubleRange** : Esta tag verifica o valor do componente dentro da faixa especificada . O valor deve ser conversível para o tipo de ponto flutuante.
- **f:ValidateLength** : Esta tag verifica o comprimento de um valor e limitá-lo dentro de um intervalo especificado . O valor deve ser do tipo java.lang.String .
- **f:validateLongRange** : Verifica se o valor do componente está dentro de um intervalo especificado. O valor deve ser do tipo numérico ou string conversível para um long.
- **f:validateRegEx** : Verifica se o valor de local de uma entrega obedece a regra da expressão regular do java.util.regex package.
- **f:validateRequired** : Garante que o valor inserido não será vazio/nulo.
- **f:validateBean** : Registra a validação do bean no componente.

Validações Customizadas

As vezes precisamos validar algo diferente, entra ai a personalização.

Para criação de um validador basta sobrescrever o método validate(), estendendo a interface javax.faces.validator.Validator, e registrá-lo no contexto do JSF.

Navegação

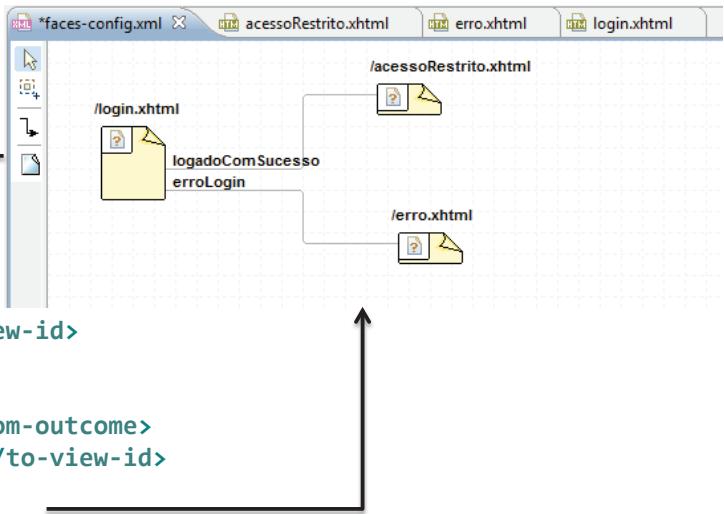
```

<navigation-rule>
  <from-view-id>/login.xhtml</from-view-id>

  <navigation-case>
    <from-outcome>logadoComSucesso</from-outcome>
    <to-view-id>/acessoRestrito.xhtml</to-view-id>
  </navigation-case>

  <navigation-case>
    <from-outcome>erroLogin</from-outcome>
    <to-view-id>/erro.xhtml</to-view-id>
  </navigation-case>

</navigation-rule>
  
```



Frameworks

O JSF tem um mecanismo de navegação melhorado, o modelo de navegação do JSF contempla as regras que definem como e quando a navegação entre páginas deve ocorrer em uma aplicação. Além de regras de navegação implícitas foi adicionado um teste que pode ser feito usando a tag **<if>** dentro do **<navigation-case>**. E para finalizar a tag **<to-view-id>** aceita EL, o que torna tudo mais dinâmico.

Navegação estática implícita

Na navegação estática implícita, quando o usuário clica em algum botão ou link, um sinal (outcome) fixo definido no próprio botão ou link é enviado para o JSF. Este sinal é uma string que será utilizada pelo tratador de navegação do JSF para definir a próxima tela que será apresentada ao usuário.

Nas navegações implícitas, os nomes dos arquivos que definem as telas de resposta são montados com as strings dos outcomes. Por exemplo, se o usuário clica em um botão ou link de uma tela definida por um arquivo chamado index.xhtml que envia o outcome “cadastro” então ele será redirecionado para a tela definida pelo arquivo cadastro.xhtml dentro do mesmo diretório que está o arquivo index.xhtml.

Navegação Estática Explícita

Na navegação implícita, os outcomes são os nomes dos arquivos que definem as telas. Para ter a liberdade de definir os nomes dos arquivos independentemente dos outcomes, po-

demos utilizar a navegação explícita. Porém, nesse tipo de navegação, devemos acrescentar algumas linhas no arquivo de configurações do JSF, o faces-config.xml.

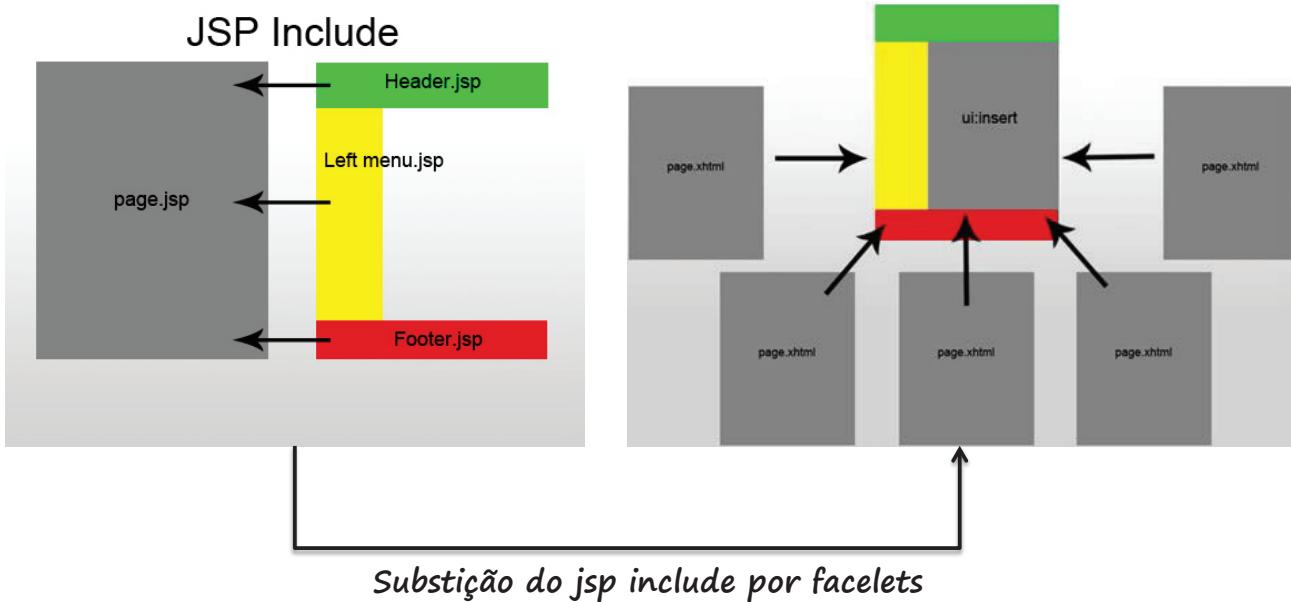
Navegação Dinâmica Implícita

Em grande parte da nossa aplicação, não vamos fixar nas telas os outcomes, iremos gerar a navegação dentro dos Managed Beans. Na navegação dinâmica, quando um usuário clica em um botão ou link, um Managed Bean é chamado para escolher um outcome e enviar para o JSF. Para isso, fazemos a ligação dos botões ou os links a métodos dos Managed Beans.

Navegação Dinâmica Explícita

Para implementar a navegação dinâmica explícita, basta seguir os passos da navegação dinâmica implícita e acrescentar as regras de navegação no arquivo de configurações do JSF.

Facelets



Frameworks

É um framework para templates Web, utilizado como view padrão do JSF 2. O Facelets requer entradas válidas de XML para e suporta todos os componentes visuais do JSF.

Facelets possui várias vantagens que vão desde a facilidade na criação e reutilização de páginas e componentes, melhor depuração de erros, AJAX nativo, uma melhor compatibilidade entre XHTML, JSTL e os componentes, ele é independente de web container, e claro, Facelets é mais rápido que JSP.

Outro ponto positivo é que Facelets tem suporte a **Unified Expression Language** (EL), incluindo o suporte para funções EL e validação EL em tempo de compilação.

Template

Ao longo do processo de desenvolvimento de uma aplicação Web, percebe-se que várias páginas têm uma estrutura semelhante, o uso de template vai permitir ao desenvolvedor web a reutilização de código das telas, utilizando-se uma página padrão com um esqueleto e que possua pontos para que sejam substituídos posteriormente. Em JSP o recurso utilizado para o reaproveitamento de código é o **include**, porém este tem várias desvantagens comparado com o **templates**.

Facelets Template

Usando templates é possível reunir alguns dos principais objetivos de desenvolvimento de aplicações web. Templates permitem a reutilização de código, reduzindo assim, o tem-

po de desenvolvimento e os custos de manutenção de uma aplicação. Além disso, templates nos auxiliam na obtenção de uma aparência comum em páginas que possuem uma estrutura semelhante. Um template define pontos onde o conteúdo pode ser substituído. O conteúdo a ser usado nesses locais, é definido pelo cliente do template (Template Client). O Template define as diferentes áreas usando a tag **<ui:insert>**, e os clientes utilizam os modelos com as tags: **<ui:component>**, **<ui:composition>**, **<ui:fragment>**, ou **<ui:decorate>**.

Internacionalização

Portugues	Inglês	
Nome:	<input type="text"/>	
Nacionalidade :	<input type="text"/>	
<input type="button" value="Salvar"/>		
Nome	Nacionalidade	Excluir
Thiago	Brasileiro	Excluir

Clique em Inglês Clique em Portuguese

Português	English	
Name:	<input type="text"/>	
Nationality:	<input type="text"/>	
<input type="button" value="Save"/>		
Name	Nationality	Delete
Thiago	Brasileiro	Delete



Frameworks

Em Java é possível fazer de forma adequada a internacionalização do código de uma aplicação desktop, com Swing utilizando recurso de formatação de **Data** e **Números**, principalmente monetário.

A internacionalização significa sua aplicação rodar em mais de um idioma, criando de forma parametrizada os dados de labels a serem mostrados em arquivos **.properties**.

A formatação baseia-se na Localização do idioma informada, seja pelo Sistema Operacional (SO) ou informado através de parametro pelo usuário. A classe **java.util.Locale** que é responsável por determinar o idioma a ser utilizado para internacionalização. Locale você define a região conforme exemplo:

```
Locale local = new Locale("pt", "BR");
```

O primeiro parâmetro define o idioma e o segundo o país, conforme **ISO-CODE**, isso porque existe países que fala dois idiomas, possibilitando então ser totalmente configurável.

A classe **ResourceBundle** é a responsável por acessar o arquivo **.properties** de internacionalização baseado no Locale definido. Existe um padrão para nomear os arquivos de internacionalização, deve ser composto por **messages_“iso-code-language”_“iso-code-country”.properties** conforme exemplo abaixo para o Brasil.

messages_pt_BR.properties

O arquivo **.properties** é composto por uma lista de registros com chave e valor, onde a chave será a mesma para todos os arquivos de todos idiomas e o valor que será diferente para cada um.

```
messages_pt_BR.properties
```

```
welcome=Bem vindo!
```

```
messages_en_US.properties
```

```
welcome=Hello!
```

Para carregar o arquivo do respectivo Locale definido é necessário utilizar o método `getBundle()` da classe passando o nome inicial do arquivo e o Locale conforme abaixo:

```
Locale ptBR = new Locale("pt", "BR");
```

```
ResourceBundle bundle = ResourceBundle.getBundle("messages", ptBR);
```

A procura pelo arquivo .properties ocorre de três formas, primeiro ele procura pelo nome completo definido pelo locale, se não encontrar então procura com somente a definição do idioma e caso não encontre ele procura com somente o nome do arquivo.

```
messages_pt_BR.properties //pesquisa 1
```

```
messages_pt.properties //pesquisa 2
```

```
messages.properties //pesquisa 3
```

O JSF procura os arquivos de tradução em um local, que é dentro de resources / src/main/resources, então é lá que vamos colocar os .properties para cada idioma que a aplicação vai suportar.

Frameworks JSF



RichFaces



ICEfaces



MyFaces



Frameworks

Com o avanço dos sistemas na internet surge à necessidade de se desenvolver aplicações cada vez mais complexas, as quais envolvem fatores como conversões, validações, mecanismos de segurança, acessibilidade, suporte a Cascading Style Sheets (CSS), suporte a criação e uso de templates, entre diversas outras funcionalidades. Para auxiliar no desenvolvimento de alguns destes fatores existe o framework Java Server Faces (JSF).

Atualmente, muitos desenvolvedores web se concentram mais com as funcionalidades do que com a parte visual dos componentes, pois existem vários frameworks que implementam o JSF, o que já facilita questões como design, componentes, validação, ajax entre outros.

Abaixo será falado um pouco sobre os principais frameworks que implementam o JSF:

- **PrimeFaces** - Oferece um conjunto de componentes com versões estáveis e de código aberto para o JSF 2.0 e permite que sejam inseridos em seu conjunto, outros componentes através de especificações em JSF. O framework permite muitas possibilidades de criação de layout para aplicações web e temas gráficos que podem ser alterados facilmente, evitando a necessidade de utilizar componentes baseados em outras tecnologias.

- **RichFaces** - É um framework de código aberto que incorpora AJAX nas aplicações. Este permite integrar AJAX no desenvolvimento do negócio da aplicação e os componentes são disponibilizados como objetos, prontos para serem utilizados, a fim de criar aplicações web que permitam uma interação entre o usuário e a aplicação mais rentável e rápida. O RichFaces é totalmente integrado no ciclo de vida do JSF, incluindo, ciclos de vida, validações, conversões e gestão de recursos estáticos e dinâmicos.

- **IceFaces** - É um conjunto de componentes recentemente tornado open source, desenvolvido pela ICESoft. Tem por finalidade integrar as tecnologias JSF e Ajax de forma “nativa”. Ou seja, todos os componentes do ICEfaces são componentes JSF que dão suporte ao Ajax.

- **MyFaces** - É uma das implementações do JSF mais utilizada atualmente é o MyFaces, um projeto da Apache Software Foundation, que vem crescendo rapidamente e hoje vai muito além da especificação JSF. Além do MyFaces Core, a implementação do JSF em si, há vários subprojetos do MyFaces que fornecem componentes adicionais, trazendo suporte a Ajax e à vinculação com dados, entre outras funcionalidades importantes no desenvolvimento web.

Frameworks Primefaces



Frameworks

Existem diversos frameworks baseados na tecnologia JSF que se destinam a tornar mais simples o uso de AJAX e componentes em aplicações web, um deles é o Primefaces.

O PrimeFaces oferece um conjunto de componentes com versões estáveis e de código aberto para o JSF e permite que sejam inseridos em seu conjunto, outros componentes através de especificações em JSF.

PrimeFaces é uma biblioteca de componentes de código aberto para o JSF 2.0 com mais de 100 componentes com versões estáveis, permitindo criar interfaces ricas para aplicações web de forma simplificada e eficiente. Ele é considerado uma das melhores bibliotecas de componentes JSF.

Vantagens de se usar Primefaces:

- Possui um rico conjunto de componentes de interface (DataTable, AutoComplete, HTMLEditor, gráficos etc).
- Nenhum xml de configuração extra é necessária.
- Componentes construídos com Ajax no padrão JSF 2.0 Ajax APIs.
- Mais de 25 temas templates.
- Boa documentação com exemplos de código.
- “Fácil de usar”, os componentes são desenvolvidos com um princípio de design: “Uma boa UI esconde a complexidade dos componentes mas também é flexível quando necessário”.
- O retorno da comunidade ajuda bastante reportando bugs e no desenvolvimento com novas ideias.

- Ele é utilizado no ecossistema Java de companhia de software, bancos, instituições financeiras, universidades e muito mais.

- O Mobile UI kit para criar aplicações web móveis para dispositivos portáteis baseados em navegadores webkit

O PrimeFaces é um framework da Prime Teknoloji (empresa da Turquia) que oferece um conjunto de componentes ricos para o JSF. Seus componentes foram construídos para trabalhar com AJAX por “default”, isto é, não é necessário nenhum esforço extra por parte do desenvolvedor para realização de chamadas assíncronas ao servidor. Além disso, o PrimeFaces permite a aplicação de temas (skins) com o objetivo de mudar a aparência dos componentes de forma simples.

Os criadores do PrimeFaces faz questão de dizer que o utilizam em todos os projetos de clientes como o framework de front end. Eles dizem que com esse tipo de atitude ajuda a realizar correções nas características e rapidamente resolver os bugs. Esta postura é uma diferença significativa do PrimeFaces em relação as outras bibliotecas.

Arquitetura



Frameworks

- **User Interface (UI) Components** – comprehends the components that contain the functionalities encapsulated by AJAX, Javascript and animated graphics;
- **Optimus** – module that provides solutions to facilitate development with JSF. Also contains security extension components;
- **FacesTrace** – module responsible for functions related to the performance of JSF-based applications.

Componentes

The screenshot displays a collection of PrimeFaces components from the showcase. It includes:

- DataTable - Instant Row Selection:** Shows a table of car models with a selected row.
- Car Detail:** A modal dialog showing details of a Renault 1973 model.
- PrimeFaces Bootstrap:** A panel with a message and navigation links.
- Godfather Part I, II, III:** Three panels showing the Godfather movie poster and a summary of the plot.
- New Person:** A form with two required fields (Firstname and Surname) marked with validation errors.
- Ajax Menutem:** A menu with items for Save, Update, Delete, and Navigations (External and Internal).
- Select One:** A dropdown menu with three options: Messi - 10, Bojan - 9, and Iniesta - 8.
- Calendar:** A calendar for August 2012 with the 22nd highlighted.



Frameworks

Primefaces é um framework que contém uma gama de componentes que abrange praticamente todas as funções que uma aplicação web com interface rica pode precisar. Este Framework apresenta uma documentação bem feita com exemplos práticos que facilitam a aplicação de componentes, estes exemplos podem ser encontrados na showcase do primefaces no link: <http://www.primefaces.org/showcase/>.

Abaixo mostraremos alguns exemplos explicando estes componentes.

Calendário

The component is a date-time picker. It features a calendar for August 2011 with the 22nd highlighted. Below the calendar are time selection controls for Hour and Minute, each with a slider and a text input. At the bottom are "Now" and "Done" buttons.

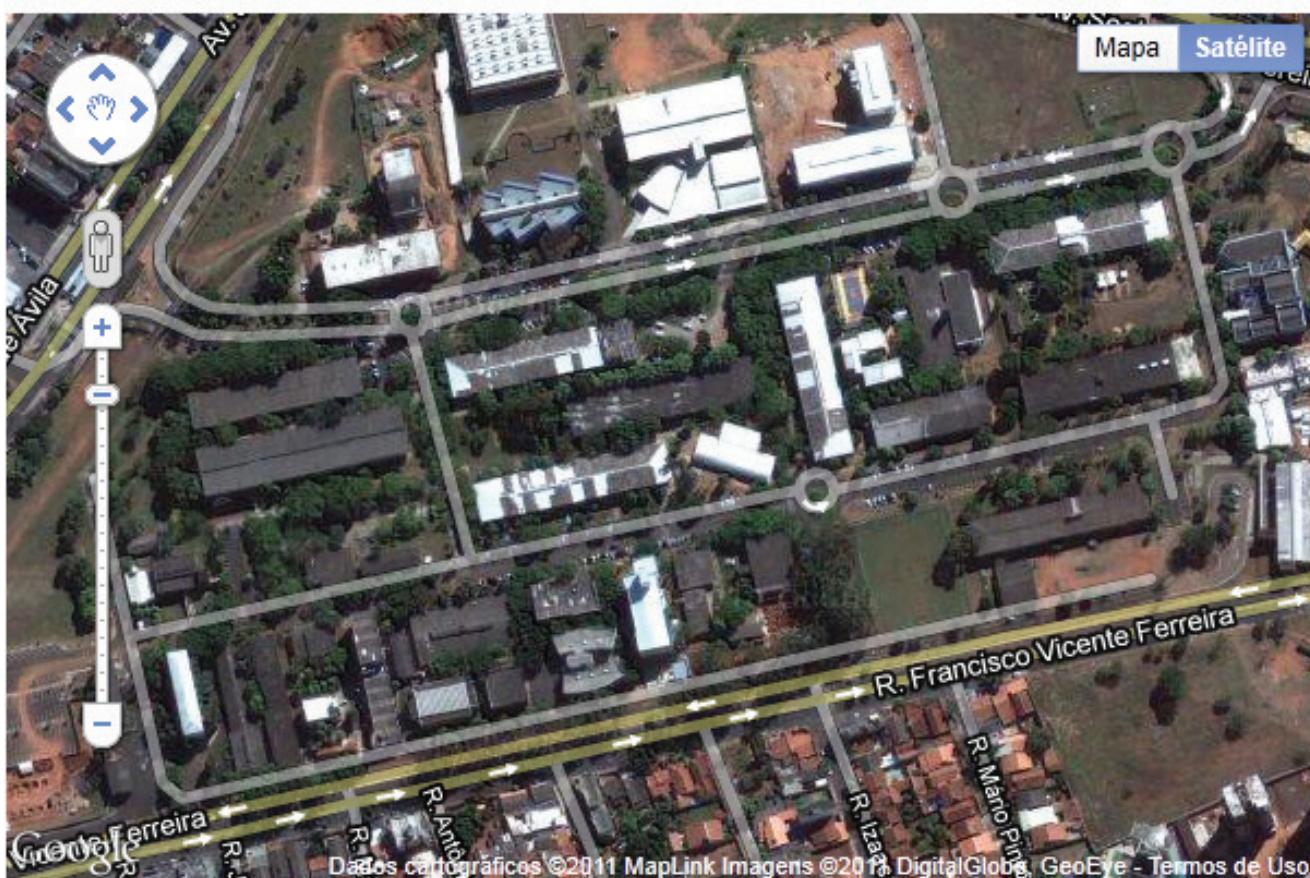
O calendário é uma interface bastante comum hoje em dia e o PrimeFaces não deixa de implementá-la. Além do calendário comum, PrimeFaces inclui um seletor de horário no componente.

Data Table

In-Cell Editing				
Model	Year	Manufacturer	Color	Options
d8f05828	1992	BMW	Brown	<input type="checkbox"/>
cef0ea87	1961	BMW	Black	<input checked="" type="checkbox"/> <input type="checkbox"/>
1731d369	1995	BMW	Maroon	<input type="checkbox"/>
081a052e	1968	Chrysler	Maroon	<input type="checkbox"/>
6692776e	1977	Ford	Red	<input type="checkbox"/>
8480eae9	1995	Mercedes	White	<input type="checkbox"/>
004591e2	1965	Opel	White	<input type="checkbox"/>
e4687cd1	1988	Ferrari	Silver	<input type="checkbox"/>
48cabc8b	1965	Volvo	Maroon	<input type="checkbox"/>

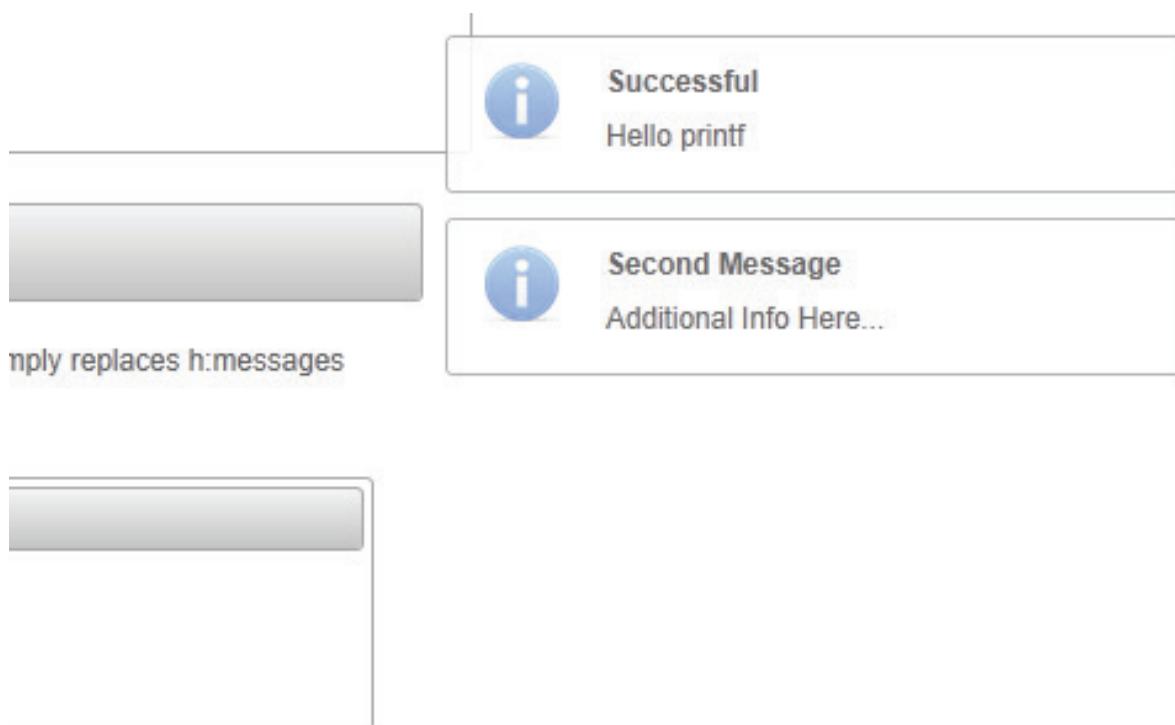
Como o próprio nome diz, esse componente apresenta uma tabela com um conjunto de dados. Essa tabela pode facilmente incluir paginação, busca e até edição em célula.

Google Maps



Com o PrimeFaces é fácil integrar sua aplicação com essa potente ferramenta do Google.

Growl



Alertar o usuário sobre o sucesso ou erro basta digitar “`<p:growl>`” para ter caixas de texto flutuando de forma elegante em sua aplicação.

Primefaces - Mobile



Frameworks

Aplicações móveis na maioria dos casos são construídas utilizando a API nativa da plataforma, limitando a execução das aplicações a dispositivos da plataforma para a qual elas foram desenvolvidas. Assim, para que uma aplicação desenvolvida para uma plataforma X seja executada na plataforma Y, ela deverá ser portada para essa nova plataforma.

Praticamente todos dispositivos móveis, possuem um navegador web que possibilita o acesso à internet. Assim outra maneira de construir aplicações para dispositivos móveis é desenvolvendo sistemas web otimizados para estes aparelhos. Essa otimização é focada principalmente na interface gráfica da aplicação, que deve ser compatível com diversas resoluções de telas e voltada para a utilização de touchscreen no lugar dos tradicionais teclado e mouse.

O PrimeFaces Mobile é um conjunto de componentes de interface gráfica desenvolvido sobre o framework jQuery Mobile. Ele permite a implementação de páginas JSF otimizadas para diversas plataformas móveis, como Android, iOS, BlackBerry, Windows Mobile, Palm e Symbian. Projetado para ser utilizado de forma simples, o PrimeFaces Mobile praticamente não exige configurações extras se comparado a outros sistemas JSF.

Podemos encontrar documentação e exemplos no site do primefaces no link <http://www.primefaces.org/showcase/mobile/>.

Quem usa?



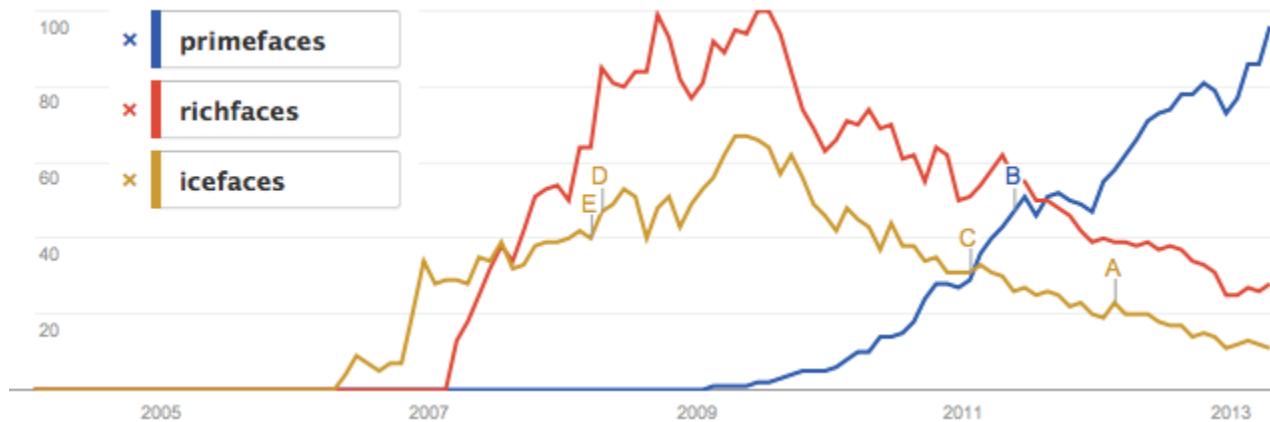
Frameworks

PrimeFaces é uma das bibliotecas de interface do usuário mais populares em Java, amplamente utilizado por empresas de software, marcas de renome mundial, bancos, instituições financeiras, companhias de seguros, universidades e muito mais. Acima estão alguns dos usuários que utiliza esse framework.

De acordo com DevRates.com , PrimeFaces está em segundo na classificação geral 9.0 (no momento da escrita), como quadro favorito dos desenvolvedores para criar interfaces de usuário com java .



Milhares de aplicativos foram criados com PrimeFaces , mesmo os produtos concorrentes como IceFaces é alimentado por PrimeFaces . A seguir é um gráfico gerado pelo Google Trends comparando os PrimeFaces popularidade com os competidores. Para muitos, PrimeFaces é o padrão de Bibliotecas de componentes JSF .



Frameworks

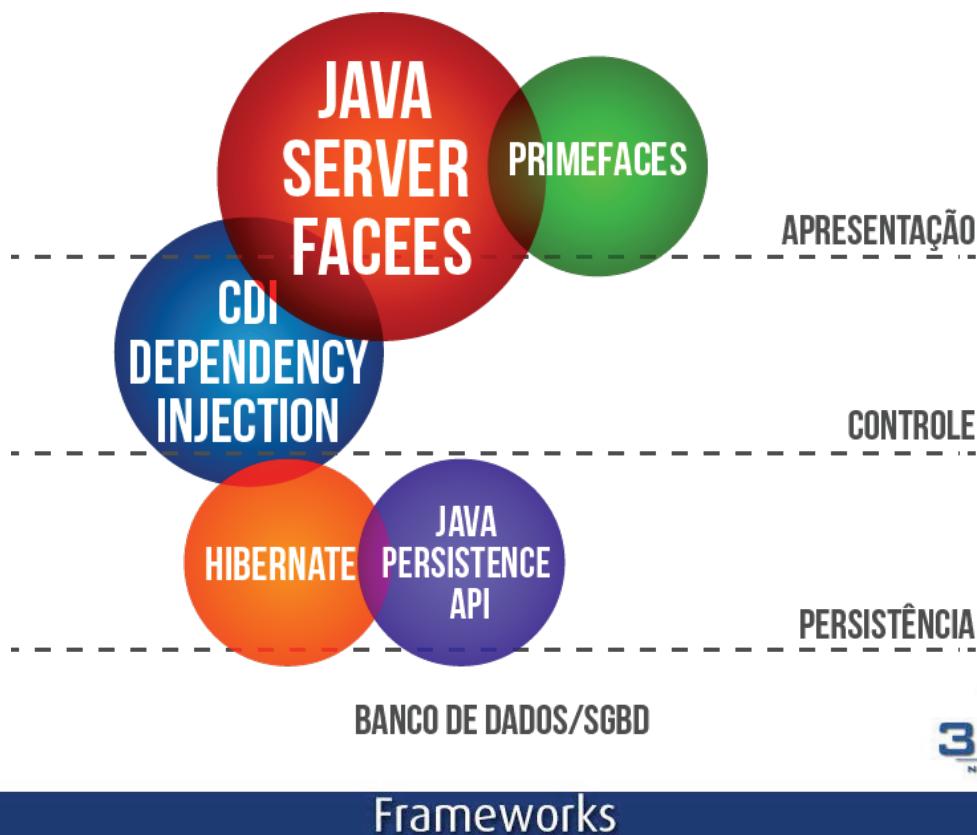
Injeção de Dependência e Contextos - CDI



A CDI é uma especificação Java, cujo nome completo é “Context and Dependency Injection for Java EE” [?], mas como podemos perceber o “for Java EE” não entra na sigla. De fato, conforme formos nos aprofundando no assunto, vamos perceber que a CDI não é apenas para o Java EE, mas também para o Java SE.

A CDI — Context and Dependency Injection for Java EE — é a especificação que rege como deve funcionar os frameworks de injeção de dependências para a plataforma Java. Dessa forma, agora temos uma maior liberdade de escolha na implementação, e mesmo frameworks que não a implementam acabam tendo um comportamento parecido com o da especificação, para que seja possível reutilizar o conhecimento e até mesmo código vindo de aplicações feitas com implementações de CDI.

O que é e para que serve a CDI?



CDI realiza injeção de dependências entre classes de uma aplicação Java EE 6 e permite, ainda, que páginas JSF e JSP façam referência a estas classes por meio da EL (linguagem de expressões unificada). As classes gerenciadas pelo CDI, as quais chamamos de beans, são associadas a determinados contextos para gerenciamento automático do seu ciclo de vida. O CDI oferece, além disso, uma série de funcionalidades como qualificadores, alternativas, decoradores, interceptadores e eventos que permitem uma grande flexibilidade no desenvolvimento da aplicação.

A CDI é só para Java EE?

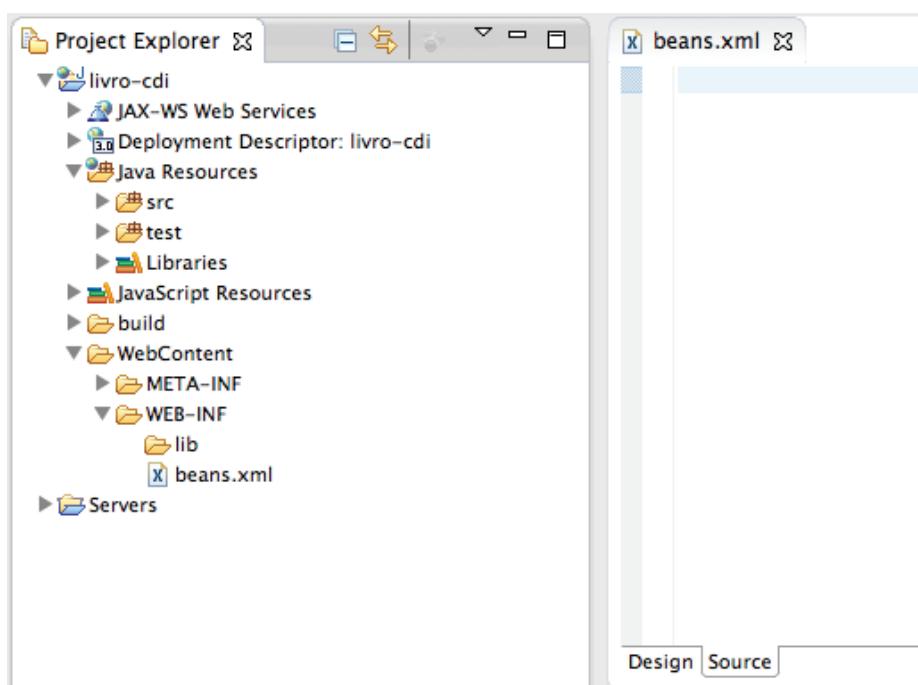
Apesar de seu nome, Context and Dependency Injection for Java EE, a CDI é uma especificação que pode ser utilizada em ambiente Enterprise ou em ambiente Standard, como dentro de um Tomcat, Jetty, ou até mesmo em uma aplicação Desktop. Esse nome não quer dizer onde a CDI roda, mas sim onde ela vem pronta para uso, que no caso é no ambiente Java EE, tanto no profile Full quanto noWeb.

Ate a versão 5, o Java EE era uma coleção enorme de especificações que, juntas, nos permitem desenvolver os mais variados tipos de aplicação, mas tinha muito mais funcionalidades do que a grande maioria das aplicações costuma usar. Por isso, a partir da versão 6, o Java EE passou a ter diferentes perfis: o Full e o Web. O Full, ou completo, é o equivalente à coleção do Java EE 5 e mais algumas especificações novas, dentre elas a CDI. Já o perfil Web é um subconjunto dessas especificações, contendo apenas as especificações mais úteis para as aplicações mais comuns.

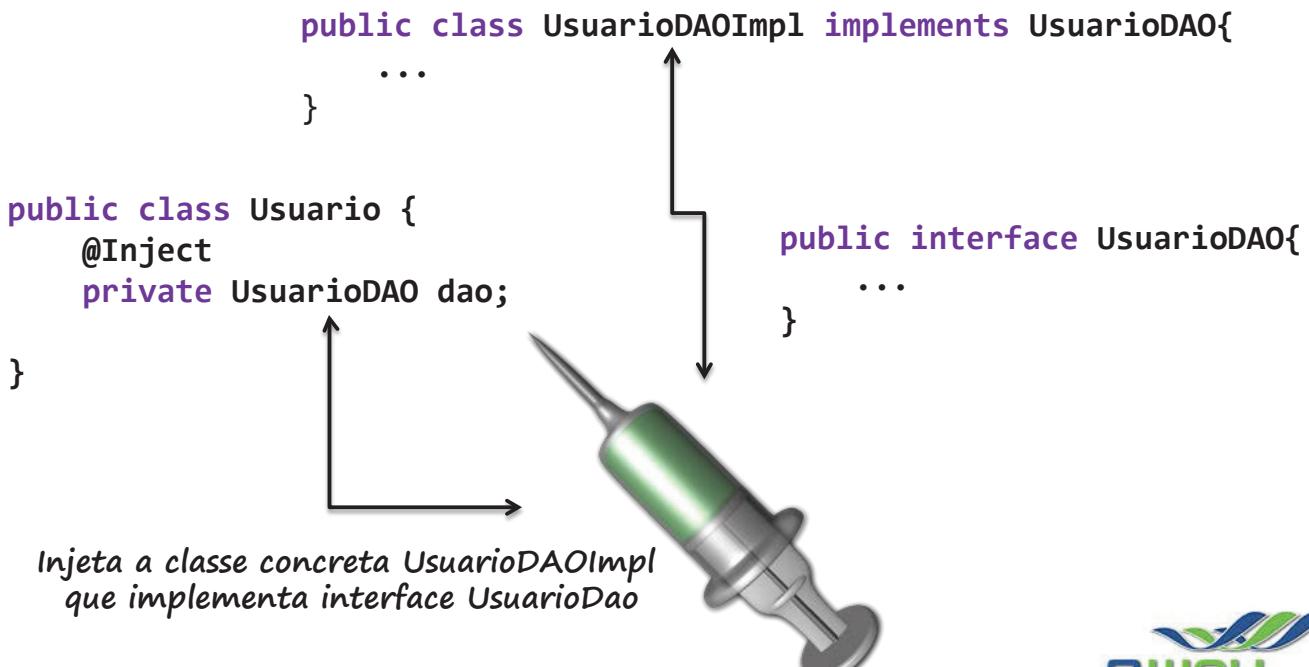
O que é um pacote CDI?

Como estamos trabalhando com uma aplicação Web (war), basta colocarmos um arquivo chamado beans.xml dentro da pasta WEB-INF da nossa aplicação. Nesse ponto podemos nos perguntar: mas e as configuração da CDI, onde colocamos? A resposta esta novamente em defaults inteligentes. Como ate aqui não precisamos fazer configuração alguma, não precisamos sequer definir uma estrutura xml mínima dentro desse arquivo. Basta que ele exista com esse nome, mesmo estando vazio, que a CDI já começa a funcionar. Somente quando precisarmos configurar algo específico, e que nos preocuparemos em por algo dentro do arquivo.

Ate agora a estrutura da nossa aplicação esta parecida coma da imagem a seguir:



Injeção de Dependências



Frameworks

A injeção de dependências é muito importante para termos um sistema mais simples de manter. Apesar de parecer que sua principal vantagem é ajudar nos testes, isso na verdade é só uma forma de percebermos sua real vantagem, que é a facilidade em trocar peças do nosso sistema diminuindo muito a possibilidade de quebra. Vamos voltar ao nosso exemplo para entendermos melhor como isso funciona.

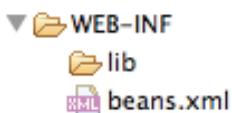
O CDI introduziu uma abordagem para melhor interação entre seus componentes (Java EE), com um avançado modelo de ciclo de vida. Entre os principais recursos adicionados pelo CDI ao Java EE:

- **Integração unificada de Expression Language (EL):** Permite que qualquer Bean do CDI possa ser exposto em um componente JSF;
- **Eventos:** CDI permite um mecanismo de troca de notificações (Eventos) Type safe;
- **Injeção de Dependência:** CDI possibilita injeção de Dependência Type safe de qualquer componente Java EE;
- **Métodos Produtores:** Traz uma abordagem para criar injeções polimórficas durante a execução;
- **Decorators:** CDI utiliza o padrão de projeto Decorator para desacoplar questões técnicas da lógica de negócios;
- **Interceptors:** Definido na especificação Java Interceptors, recurso que cria um meio para interceptar métodos e fazer algum tipo de processamento no mesmo.
- **Conversation Scope:** CDI adiciona um novo escopo além dos já definidos Request, Application e Session;

Para solicitar a injeção de uma dependência basta utilizarmos a anotação `@javax.inject.Inject`. Dessa forma a CDI procura em seu contexto uma classe candidata a suprir essa dependência, e como por padrão toda classe dentro de um pacote CDI é candidata a ser injetada, não precisamos fazer nenhuma configuração nas classes. Veremos que em alguns casos é interessante especificarmos algumas características adicionais, mas isso é a exceção, pois o default já nos serve para a maioria dos casos. Defaults inteligentes são uma das marcas da CDI.

O que é um “bean”?

Managed Beans
Session Beans



```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
  <alternatives></alternatives>
  <decorators></decorators>
  <interceptors></interceptors>
</beans>
```



Frameworks

Em um projeto que usa CDI, **quase** todas as classes são consideradas como beans CDI, também conhecidas como CDI managed beans. Beans CDI podem ser injetados em outros beans.

Beans são definidos como objetos gerenciados pelo container com mínimas restrições de programação, também conhecidos pelo acrônimo **POJO** (Plain Old Java Object). Eles suportam um pequeno conjunto de serviços básicos, como injeção de recurso, callbacks e interceptadores do ciclo de vida.

Com pouquíssimas exceções, quase toda classe Java concreta que possui um construtor com nenhum parâmetro (ou um construtor designado com a anotação @Inject) é um bean.

Um bean tem a responsabilidade de definir como outros beans dos quais ele depende ou necessita em seu modelo, serão interpretados pelo container, que por sua vez, irá gerenciar o ciclo de vida destes objetos. Ou seja, um bean pode requerer instâncias de outros tipos de beans.

O Ciclo de vida de um bean dependerá do escopo em que ele foi inserido. Por sua vez, os diferentes escopos definem ao container como suas informações serão geridas.

O ciclo de vida de beans é completamente desacoplado no CDI, pois permite que diferentes beans que implementam a mesma interface sejam substituídos uns aos outros mesmo quando possuem escopos diferentes.

O uso do @PostConstruct

Quando usamos CDI, ou mesmo outro framework que gerencia as dependências das nossas classes, precisamos ficar atentos sobre o fato de que o método que representa a instanciação em si não é a chamada do construtor, e sim, o método anotado com `@PostConstruct`, pois na invocação desse método sim temos certeza que o objeto está pronto.

Então, se pretendíamos fazer alguma programação no construtor, pode ser interessante fazer no `@PostConstruct`, pois nele, independentemente se a injeção de dependência foi via construtor, método inicializador ou diretamente nas propriedades, saberemos que o objeto está pronto. É uma espécie de construtor lógico do objeto.

Resolvendo dependências com tipagem forte

Talvez a principal funcionalidade da CDI seja a resolução de dependências de forma tipada. Enquanto as primeiras ferramentas de injeção de dependências eram baseadas em XML ou em anotações que usavam Strings, a CDI usa a mesma tipagem forte que estamos habituados a usar no Java.

Não injetamos uma dependência baseado no nome de um bean, e sim no seu tipo. A utilização de nomes é feita em casos excepcionais, por exemplo, quando queremos injetar um objeto de um tipo muito simples, como uma String.

Escopos do CDI

@ApplicationScoped

@SessionScoped

@ConversationScoped

@RequestScoped



Frameworks

CDI Bean é muito mais poderoso em comparação a um JSF Bean. Então se não existir nenhum problema com o cenário da aplicação web em usar CDI Bean devemos usa-lo.

Comparação entre escopos do JSF com CDI:

	JSF Bean	CDI Bean
Session Scope	Sim	Sim
Request Scope	Sim	Sim
Application Scope	Sim	Sim
Conversation Scope	Não	Sim
View Scope	Sim	Não
Custom Scope	Sim	Não

Breve explicação dos Escopos do CDI

- Session Scope persiste desde o inicio de uma sessão até o termino dela. (Quando um usuário começa o acesso da aplicação até terminar o acesso da aplicação)
- Request Scope inicia quando um HTTP request é submetido até que a resposta é enviada de volta ao cliente.

- Application Scope persiste durante toda a vida da aplicação, a partir da hora em que ela for instanciada.
- Conversation Scope persiste até que um objetivo é alcançado.

Session, Application e Request, funcionam exatamente como são definidos nas aplicações Java EE, porém, estas aplicações não possuem um novo escopo, definido pelo CDI, o ConversationScope.

Conversation Scope

Este escopo é bem parecido com o Session, pois também mantém o estado associado a um usuário do sistema. Porém, neste escopo, os dados só serão guardados durante a comunicação entre cliente/servidor.

Para entender melhor este escopo pode-se pensar em formulários do tipo Wizard, onde, cede-se informações à aplicação durante várias etapas e as informações cedidas em cada etapa devem ser guardadas enquanto aquele formulário estiver ativo, ou seja, durante aquela tarefa.

Este escopo define uma melhor maneira de guardar dados de um usuário específico, quando aqueles dados só precisam ser mantidos durante uma determinada tarefa.

Qualificadores

```

public interface Relatorio {
}

@Qualifier
@Target({ TYPE, METHOD, PARAMETER, FIELD })
@Retention(RUNTIME)
@Documented
public @interface Analitico {

}

@Analitico
public class RelatorioAnalitico implements Relatorio{
}

@Qualifier
@Target({ TYPE, METHOD, PARAMETER, FIELD })
@Retention(RUNTIME)
@Documented
public @interface Sintetico {

}

@Sintetico
public class RelatorioSintetico implements Relatorio{
}

```



Frameworks

A forma mais comum de eliminar a ambiguidade é através do uso de qualificadores. Esse nome, por sinal, é praticamente autoexplicativo. Com qualificadores conseguimos distinguir um candidato a suprir uma dependência de outro.

CDI define alguns qualificadores chamados de Built-in Qualifiers, ou seja, qualificadores já criados e definidos. São Eles:

- **@New**
- **@Named**
- **@Default**
- **@Any**

Todo bean é anotado como **@Any**, já a anotação **@Default** é usada para todo aquele que não possui um qualificador diferente de **@Named**. Tanto **@Any** quanto **@Default** são declaradas implicitamente. Anotando um bean como **@Named** pode-se definir um nome, o qual definirá este bean em um EL, como no JSF, por exemplo.

A anotação **@New** permite criar uma nova instância de um bean que deve ser obrigatoriamente um ManagedBean. Esta nova instância não dependerá do seu próprio escopo, e sim, do escopo declarado naquela injeção específica.

Interceptadores

```

@Retention(RetentionPolicy.RUNTIME)
@InterceptorBinding
public @interface Logged {
}

@Interceptor
@Logged
public class LoggerInterceptor {

    @TreinamentoLogger
    private Logger logger;

    @AroundInvoke
    public Object aroundInvoke(InvocationContext context) throws Exception {
        logger.debug("Executando método: " + context.getMethod());
        return context.proceed();
    }
}

@Logged
public class RelatorioBean {
}
  
```



Frameworks

Um interceptador é uma classe usada para intervir em chamadas de métodos de classes. Podemos usar interceptadores para várias coisas, como registrar logs ou executar tarefas repetitivas e que não fazem parte da regra de negócio do sistema.

Segundo a especificação Java Interceptors, estas classes têm como funcionalidade receber requisições de métodos de uma classe alvo. CDI traz um conjunto de melhorias para a especificação com uma melhor abordagem, baseada em anotações e semanticamente melhor.

As anotações são usadas para associar um interceptador a um ou mais beans, podendo então, executar diferentes tarefas antes de chamar um método alvo, ou seja, o que foi interceptado. Entre os tipos de interceptação, pode-se dividir entre as seguintes categorias:

- **Ciclo de Vida**
- **Métodos de negócio**
- **Timeout**

Uma interceptação de ciclo de vida é toda aquela que será acionada após algum evento relacionado ao ciclo de vida do bean alvo, como após a construção do mesmo.

As interceptações de métodos de negócio são em nível de método, obviamente, ou seja, um método será marcado para ser interceptado, não um bean. O CDI também permite que uma classe possa interceptar tanto ciclo de vida quanto métodos de negócio.

Interceptação Timeout refere-se a uma especificação do EJB que define temporização para certas chamadas de um EJB, este tipo de interceptação ocorrerá após o estouro deste temporizador.

Decorators

Decorators em CDI nada mais são do que uma forma de implementarmos o padrão de projeto de mesmo nome. Logo, a definição de um se aplica ao outro.

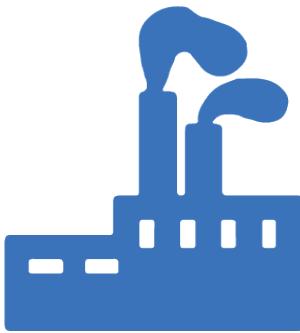
Um decorator, ou decorador, é um objeto que estende as funcionalidades de um objeto já existente, para isso ele precisa ter a mesma tipagem do objeto decorado. A grande sacada desse padrão e que em vez de estender o objeto original e sobrescrever os métodos que desejamos alterar ou criar novos métodos, fazemos isso de forma dinâmica, em tempo de execução.

Produtores

```
public class EntityManagerProducer{

    private EntityManagerFactory emf = Persistence.createEntityManagerFactory("persistenceUnit");

    @Produces
    public EntityManager create(){
        return factory.createEntityManager();
    }
}
```



Frameworks

Sim, você pode injetar qualquer coisa em qualquer lugar, a única coisa que você precisa fazer é produzir a coisa que deseja injetar. Para isto, CDI possui produtores (**producers**) (uma boa implementação do Padrão de Fábricas – Factory pattern). Um produtor expõe qualquer tipo de:

- **Classe:** conjunto sem restrições dos tipos de beans, superclasses, e todas as interfaces implementadas diretamente ou indiretamente
- **Interface:** conjunto sem restrições dos tipos de beans, interfaces implementadas diretamente ou indiretamente

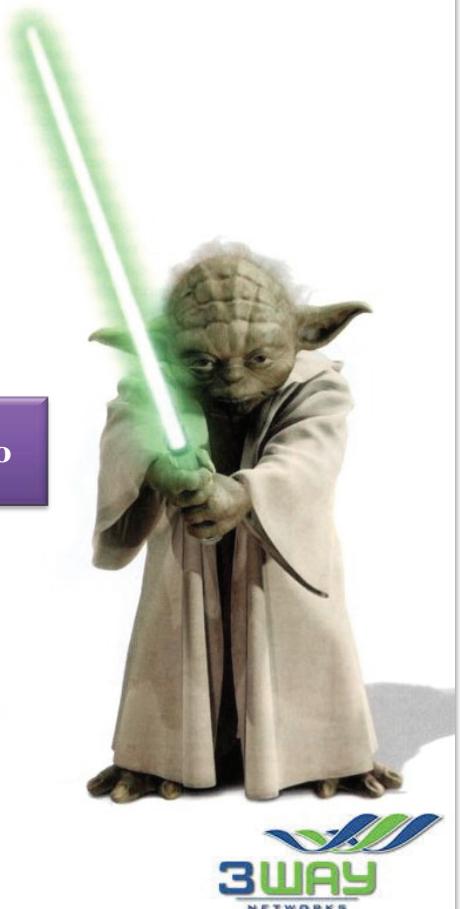
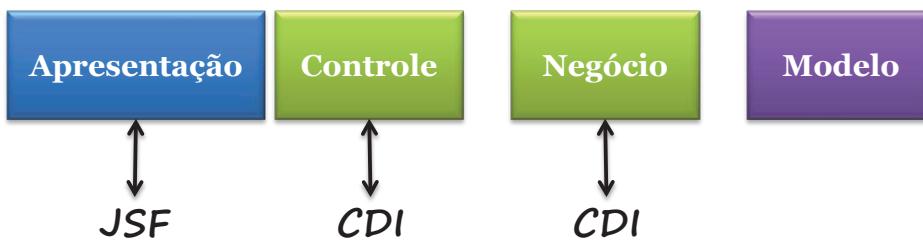
Primitivos e tipos de Array do Java

Então com isso você pode injetar `java.util.Date`, `java.lang.String` ou ainda um `int`. Vamos iniciar produzindo e injetando alguns tipos de dados e primitivos.

Bem simples. Basta criarmos um método anotado com `@Produces` que retorne o tipo que queremos produzir. Dessa maneira, quando o ponto de injeção solicitar a dependência, a CDI irá chamar o método produtor para gerar o objeto esperado.

Quando o contexto CDI sobe, as dependências são todas checadas para ver se tem beans que as satisfaçam. E nesse momento que a CDI lança exceções caso a dependência não possa ser satisfeita ou em caso de ambiguidade, que é quando encontra mais de um candidato compatível.

CDI e JSF



Frameworks

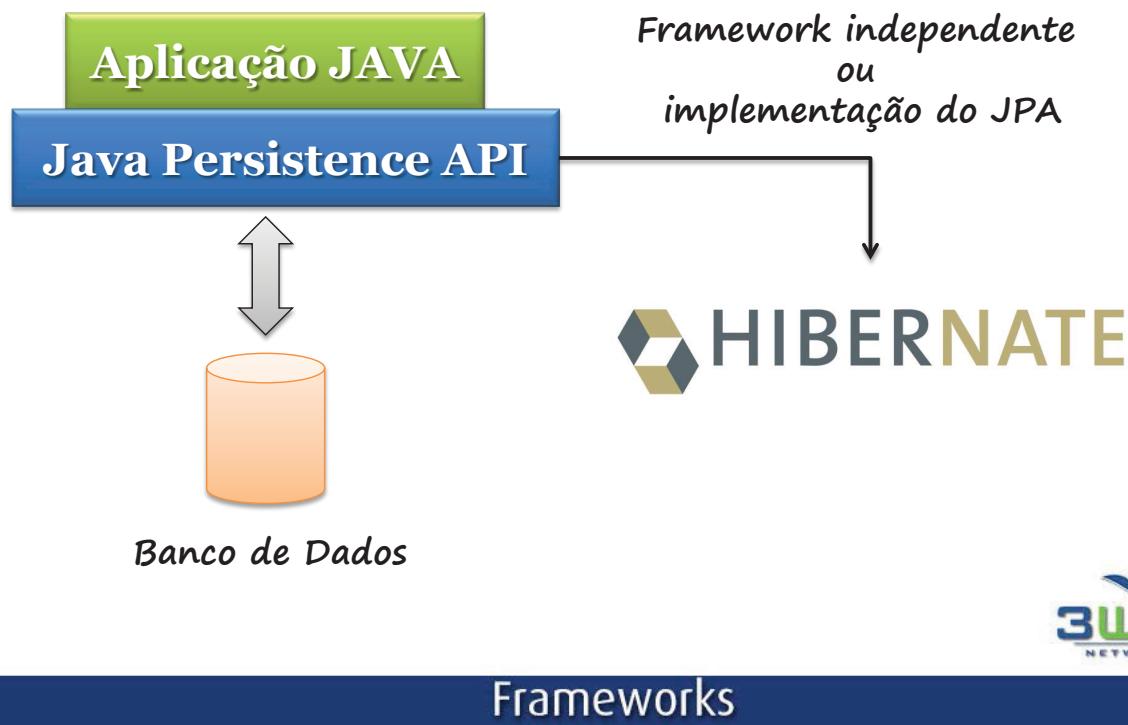
A Injeção de Dependência e Contextos (CDI), especificada por JSR-299, é parte integrante do Java EE 6 e fornece uma arquitetura que permite aos componentes do Java EE, como os servlets, enterprise beans e JavaBeans, existirem dentro do ciclo de vida de uma aplicação com escopos bem definidos. Além disso, os serviços CDI permitem que os componentes do Java EE como beans gerenciados do JavaServer Faces (JSF), sejam injetados e interajam de maneira acoplada flexível, disparando e observando eventos.

A prática mais comum de modelar uma solução Web para este tipo de problema é trabalhar com **5 camadas lógicas** : de apresentação, de controle, de negócio , de acesso a dados e de domínio . Essa prática , na realidade , é o resultado de uma evolução do padrão de projeto **MVC - Model , View , Control** , com a divisão da camada de modelo em **3 camadas** : negocio , acesso a dados e a camada das entidades de domínio. Dependendo do campo do problema pode ser até maior o numero de camadas , mas de modo geral são essas 5 as utilizadas.

Essa arquitetura **lógica** proporciona a separação de responsabilidades na automação da solução , e aliando-se ao CDI possibilita o desacoplamento dos objetos associados à interface do usuário (apresentação) com os objetos de negocio e destes com os objetos de domínio.

Frameworks

JPA – Java Persistence API | Hibernate



Após algum tempo de estouro do Hibernate, surge o Java Persistence API, especificação criada com o objetivo de padronizar as ferramentas de mapeamento objeto relacional para aplicações JAVA, diminuindo sua complexidade de desenvolvimento, sendo muito utilizado na comunidade para a camada de Persistência.

JPA não é uma nova tecnologia, mas sim uma especificação padronizada que ajuda a construir uma camada de persistência que é independente de qualquer provedor de persistência particular. Java Persistence API baseia-se nas principais ideias dos frameworks de persistência principais e APIs como Hibernate e TopLink da Oracle.

A *Java Persistence API* (JPA) é um framework para persistência em Java, que oferece uma API de mapeamento objeto-relacional e soluções para integrar persistência com sistemas corporativos escaláveis.

Com JPA, os objetos são POJO (*Plain Old Java Objects*), ou seja, não é necessário nada de especial para tornar os objetos persistentes. Basta adicionar algumas anotações nas classes que representam as entidades do sistema e começar a persistir ou consultar objetos.

JPA é uma especificação, e não um produto. Para trabalhar com JPA, precisamos de uma implementação.

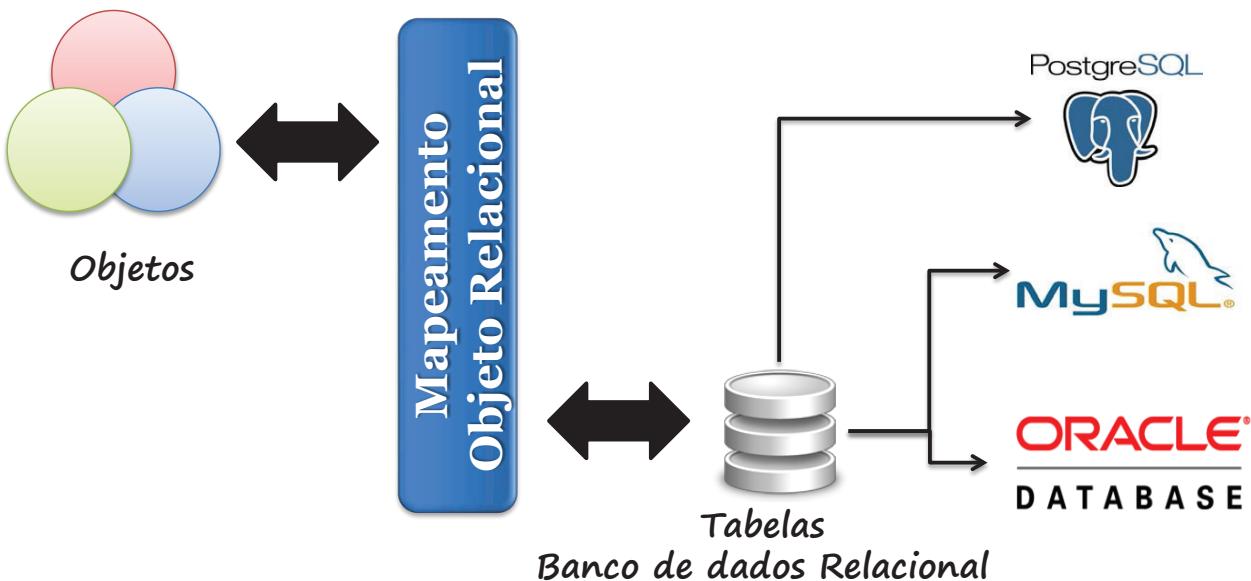
O Hibernate é um framework de mapeamento objeto relacional para aplicações Java, ou seja, é uma ferramenta para mapear classes Java em tabelas do banco de dados e vice-versa. É bastante poderoso e dá suporte ao mapeamento de associações entre objetos, herança, polimorfismo, composição e coleções.

O Hibernate não apresenta apenas a função de realizar o mapeamento objeto relacional. Também disponibiliza um poderoso mecanismo de consulta de dados, permitindo uma redução considerável no tempo de desenvolvimento da aplicação.

O projeto do Hibernate ORM possui alguns módulos, sendo que o **Hibernate EntityManager** é a implementação da JPA que encapsula o Hibernate Core.

O **Hibernate Core** é a base para o funcionamento da persistência, com APIs nativas e metadados de mapeamentos em arquivos XML. Possui uma linguagem de consultas chamada HQL (parecido com SQL), um conjunto de interfaces para consultas usando critérios (Criteria API), etc.

Mapeamento Objeto Relacional



Frameworks

Na década de 80, foram criados os bancos de dados relacionais que substituíram as bases de dados de arquivos. Para esse tipo de base de dados foi criada uma linguagem, a SQL. Essa linguagem foi toda baseada na lógica relacional e por isso contava com diversas otimizações em suas tarefas se comparadas com as outras tecnologias existentes. A partir de então foi diminuído o tempo gasto para as operações de persistência, mesmo com um grande volume de dados. Entretanto, essa linguagem não propiciava aos desenvolvedores uma facilidade para que a produtividade fosse aumentada.

Uma solução que surgiu no início da década de 90 foi à criação de um modelo de banco de dados baseado no conceito de orientação a objetos. Este modelo visava facilitar, para os desenvolvedores, a implementação da camada de persistência da aplicação, pois eles já estavam familiarizados com o paradigma de orientação a objetos, consequentemente, a produtividade certamente aumentaria. Na prática, esse modelo de dados não foi utilizado em grandes aplicações, visto que elas tinham um volume de dados muito grande e esse modelo era ineficiente em termos de tempo de resposta, pois ao contrário dos bancos de dados relacionais, eles não tinham um modelo matemático que facilitasse as suas operações de persistências.

Então a solução foi usar os BDR e desenvolver ferramentas para que o seu uso seja facilitado. Uma dessas ferramentas é o framework Hibernate que usa o conceito de mapeamento objeto relacional (MOR).

Um dos principais objetivos dos frameworks ORM é estabelecer o mapeamento entre os conceitos do modelo orientado a objetos e os conceitos do modelo entidade relacionamento. Este mapeamento pode ser definido através de xml ou de maneira mais prática com anotações

Java. Quando utilizamos anotações, evitamos a criação de extensos arquivos em xml.

Mapeamento objeto relacional (*object-relational mapping*, ORM, O/RM ou O/R mapping) é uma técnica de programação para conversão de dados entre banco de dados relacionais e linguagens de programação orientada a objetos.

Em banco de dados, entidades são representadas por tabelas, que possuem colunas que armazenam propriedades de diversos tipos. Uma tabela pode se associar com outras e criar relacionamentos diversos.

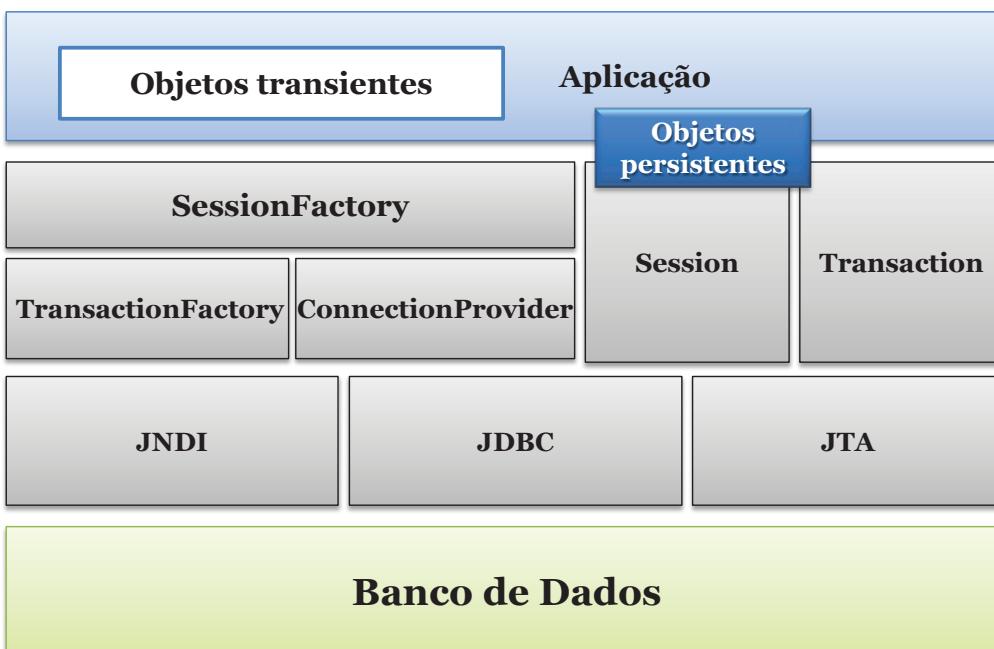
Em uma linguagem orientada a objetos, como Java, entidades são classes, e objetos dessas classes representam elementos que existem no mundo real. Por exemplo, um sistema de faturamento possui a classe NotaFiscal, que no mundo real existe e todo mundo já viu alguma pelo menos uma vez, além de possuir uma classe que pode se chamar Imposto, que infelizmente todo mundo sente no bolso. Essas classes são chamadas de classes de domínio do sistema, pois fazem parte do negócio que está sendo desenvolvido.

Em banco de dados, podemos ter as tabelas *nota_fiscal* e também *imposto*, mas a estrutura de banco de dados relacional está longe de ser orientado a objetos, e por isso a ORM foi inventada para suprir a necessidade que os desenvolvedores têm de visualizar tudo como objetos para programarem com mais facilidade.

Podemos comparar o modelo relacional com o modelo orientado a objetos conforme a tabela abaixo:

Modelo relacional	Modelo OO
Tabela	Classe
Linha	Objeto
Coluna	Atributo
-	Método
Chave estrangeira	Associação

Arquitetura do Hibernate



Frameworks

A arquitetura do Hibernate é formada basicamente por um conjunto de interfaces. A Figura Acima apresenta as interfaces mais importantes nas camadas de negócio e persistência. A camada de negócio aparece acima da camada de persistência por atuar como uma cliente da camada de persistência.

As interfaces são classificadas como:

- Interfaces responsáveis por executar operações de criação, deleção, consulta e atualização no banco de dados: Session, Transaction e Query;
- Interface utilizada pela aplicação para configurar o Hibernate: Configuration;
- Interfaces responsáveis por realizar a interação entre os eventos do Hibernate e a aplicação: Interceptor, Lifecycle e Validatable.
- Interfaces que permitem a extensão das funcionalidades de mapeamento do Hibernate: UserType, CompositeUserType, IdentifierGenerator.

O Hibernate também interage com APIs já existentes do Java: JTA, JNDI e JDBC.

Session (org.hibernate.Session)

O objeto Session possibilita a comunicação entre a aplicação e o banco de dados, através de uma conexão JDBC. É um objeto leve de ser criado, não deve ter tempo de vida por toda a aplicação. Um objeto Session possui um cache local de objetos recuperados na sessão. Com ele é possível criar, remover, atualizar e recuperar objetos persistentes.

SessionFactory (org.hibernate.SessionFactory)

O objeto SessionFactory é aquele que mantém o mapeamento objeto relacional em memória. Permite a criação de objetos Session, a partir dos quais os dados são acessados, também denominado como fábrica de objetos Sessions.

Um objeto SessionFactory é threadsafe, porém deve existir apenas uma instância dele na aplicação, pois é um objeto muito pesado para ser criado várias vezes.

Configuration (org.hibernate.Configuration)

Um objeto Configuration é utilizado para realizar as configurações de inicialização do Hibernate. Com ele, define-se diversas configurações do Hibernate, como por exemplo: o driver do banco de dados a ser utilizado, o dialeto, o usuário e senha do banco, entre outras. É a partir de uma instância desse objeto que se indica como os mapeamentos entre classes e tabelas de banco de dados devem ser feitos.

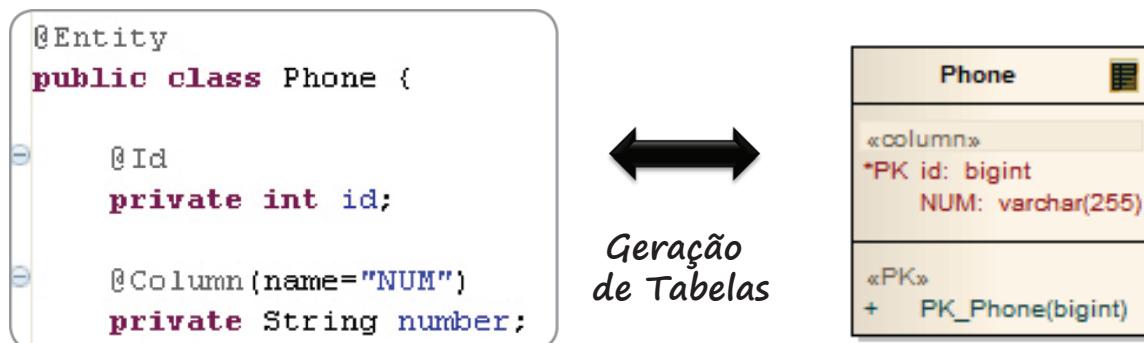
Transaction (org.hibernate.Transaction)

A interface Transaction é utilizada para representar uma unidade indivisível de uma operação de manipulação de dados. O uso dessa interface em aplicações que usam Hibernate é opcional. Essa interface abstrai a aplicação dos detalhes das transações JDBC, JTA ou CORBA.

Interfaces Criteria e Query

As interfaces Criteria e Query são utilizadas para realizar consultas ao banco de dados.

Mapeamento utilizando Anotações



Frameworks

A JPA possui várias anotações para o mapeamento de classes em tabelas. A seguir veremos algumas anotações que iremos utilizar no projeto.

1) @Entity – toda classe que represente uma tabela no banco de dados deve ser anotada com essa anotação;

2) @Table(name) – anotação responsável por dizer qual tabela no banco de dados a classe irá representar. O parâmetro **name** deve ser uma **String** contendo o nome da tabela;

3) @Column(name, nullable, length) – cada atributo da classe que represente uma coluna da tabela no banco de dados deve ser anotado com essa anotação. Segue detalhes sobre os parâmetros:

- **name** – nome da coluna (**String**);
- **nullable** – valor que representa se o atributo poderá ter valor nulo ou não (**Boolean**);
- **length** – tamanho máximo do atributo (**Integer**).

4) @Enumerated(EnumType) – utilizada para atributos que serão representados por enumeradores (ex: **Sexo.Masculino** e **Sexo.Feminino**). Recebe como parâmetro a forma como o enumerador será persistido no banco de dados:

- **EnumType.STRING** – o valor textual da opção será armazenado no banco (caso a opção escolhida seja **Sexo.Masculino** será armazenada a String “Masculino”);
- **EnumType.ORDINAL** – o número que representa a posição da opção será armazenado no banco (caso seja escolhido **Sexo.Masculino** será armazenado 0, **Sexo.Feminino** será 1).

5) @Temporal(TemporalType) – utilizada para atributos que representam data.

- **TemporalType.DATE** – armazena somente a data (dia, mês e ano) do atributo no banco;
- **TemporalType.TIME** – armazena somente o horário (hora, minuto e segundo) do atributo no banco;
- **TemporalType.TIMESTAMP** – armazena data e horário do atributo no banco.

6) @Id – informa qual atributo é a chave primária da tabela. Só um atributo da classe pode estar anotado com essa anotação.

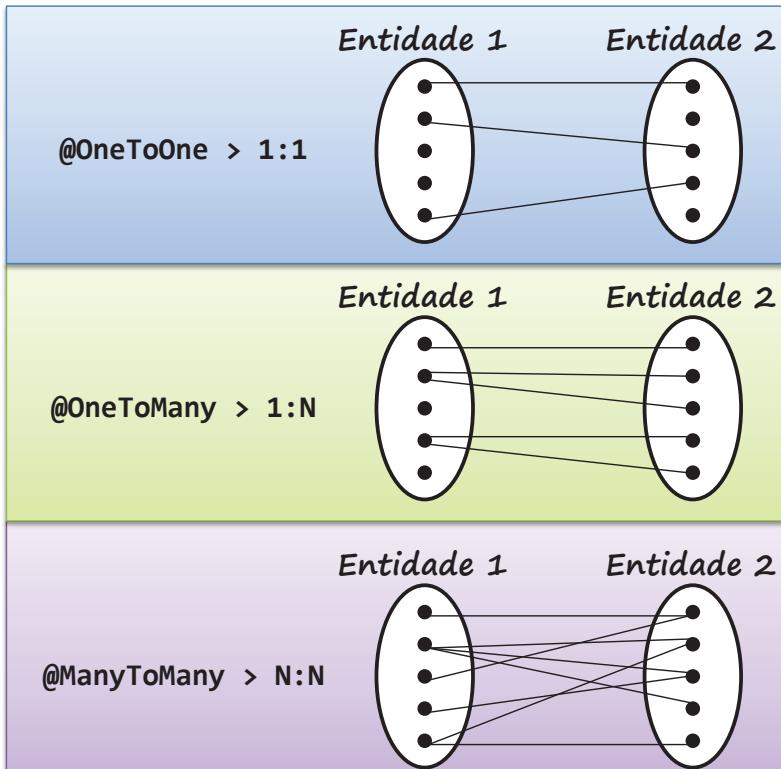
7) @SequenceGenerator(name, sequenceName) – pode anotar tanto um atributo quanto a classe. É utilizado para mapear um gerador de sequência do banco. Seus parâmetros são:

- **name** – nome que as outras anotações deverão utilizar para referenciar o gerador;
- **sequenceName** – nome do gerador no banco de dados.

8) @GeneratedValue(strategy, generator) – utilizada para anotar atributos cujo valor seja gerado por um gerador de sequência (**id**, por exemplo).

9) @Transient - Informa que o atributo não representa uma coluna da tabela.

Associações



Frameworks

Não se deve mapear somente os objetos no banco de dados, como também mapear as relações que o objeto está envolvido como possam ser restaurados. Existem quatro tipos de relações que um objeto pode ser envolvidos: Herança, Associação, Agregação e Composição. Para mapear essas relações de forma eficaz, devemos entender as diferenças entre elas, como implementar relacionamentos em geral, e como implementar especificamente os relacionamentos “muitos para muitos”.

Herança entre entidades com JPA

Muitas vezes temos várias entidades em um projeto que possuem propriedades em comum, como código ou nome, por exemplo. Para facilitar a criação dessas entidades pode-se criar uma superclasse, que não é uma entidade e que contenha as propriedades que serão herdadas pelas entidades. Uma entidade pode herdar algo de uma superclasse sendo que esta superclasse é uma classe em seu modelo de domínio que não será transformada em uma entidade. Para resolver este problema temos a anotação **@javax.persistence.MappedSuperclass**.

Relacionamentos entre entidades com JPA

- **Relacionamento @OneToOne 1:1** - quando uma tabela é dependente de outra. Veja um relacionamento entre Endereço e Cliente. A tabela Endereço é dependente da tabela Cliente. Note que a chave estrangeira originada da tabela Endereço identifica a tabela Cliente.
- **Relacionamento @OneToMany 1:N**: - quando uma linha de uma tabela se relaciona com várias linhas de outra. Veja o relacionamento entre Cidade e Bairro, isto pode

ser traduzido pela seguinte frase: “Uma cidade possui vários bairros”. Note que do ponto de vista da cidade, a relação é 1:N, enquanto que do ponto de vista do Bairro, a relação é N:1.

• **Relacionamento @ManyToMany N:N:** Em um relacionamento many-to-many um registro da tabela ‘A’ pode referenciar vários registros na tabela ‘B’, e um Registro da tabela ‘B’ pode referenciar vários registros da tabela ‘A’. Por exemplo, um livro possui muitos autores e um autor possui muitos livros. Para informar a cardinalidade do relacionamento entre livros e autores, devemos utilizar a anotação @ManyToMany na coleção.

FetchType

Com o FetchType podemos definir a forma como serão trazidos os relacionamentos, podemos fazer de duas formas:

• **FetchType.EAGER:** sempre que o objeto “pai” for trazido da base de dados, o atributo mapeado com fetch=FetchType.EAGER fará com que o seu conteúdo também seja trazido;

• **FetchType.LAZY:** sempre que o objeto “pai” for trazido da base de dados, o atributo mapeado com fetch=FetchType.LAZY fará com que o seu conteúdo somente seja trazido quando acessado pela primeira vez.

Relacionamentos OneToMany e ManyToMany têm por default o tipo de recuperação Lazy, enquanto os relacionamentos OneToOne e ManyToOne tem o tipo de recuperação Eager.

Cascade

A anotação @Cascade, também utilizada no mapeamento da coleção centros, serve para indicar com que ação em cascata o relacionamento será tratado, ou seja, especifica quais operações deverão ser em cascata do objeto pai para o objeto associado. Por exemplo, pode assumir alguns dos valores abaixo:

• **CascadeType.PERSIST:** os objetos associados vão ser inseridos automaticamente quando o objeto “pai” for inserido;

• **CascadeType.SAVE_UPDATE:** os objetos associados vão ser inseridos ou atualizados automaticamente quando o objeto “pai” for inserido ou atualizado;

• **CascadeType.REMOVE:** os objetos associados ao objeto “pai” vão ser removidos, quando o mesmo for removido;

• **CascadeType.REPLICATE:** Se o objeto for replicado para outra base de dados, os filhos também serão;

• **CascadeType.LOCK:** Se o objeto for reassociado com a sessão persistente, os filhos também serão;

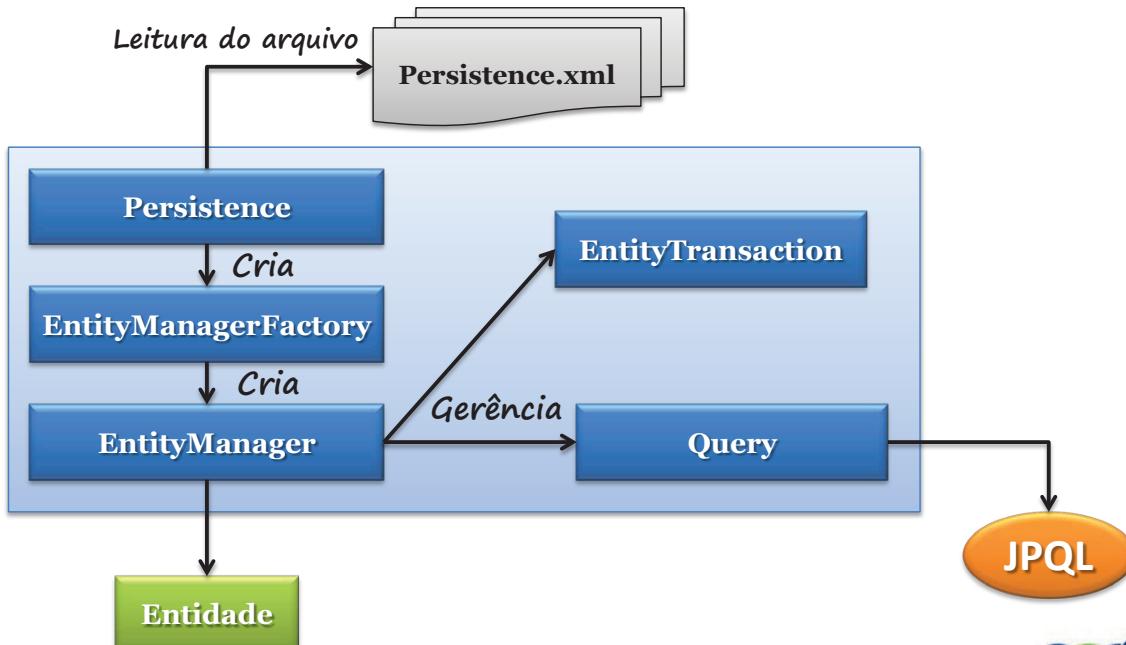
• **CascadeType.REFRESH:** Se o objeto for sincronizado com a base de dados, os filhos também serão;

• **CascadeType.MERGE:** Se um objeto tiver suas modificações mescladas em uma sessão, os filhos também terão;

• **CascadeType.EVICT:** Se o objeto for removido do cache de primeira nível, os filhos também serão;

• **CascadeType.ALL:** junção de todos os tipos de cascade.

Entity Manager



Frameworks

EntityManager é o serviço central para todas as ações de persistência. Entidades são objetos de Java, eles não ficam persistentes explicitamente até seu código interagir com o EntityManager para os fazer persistente. O EntityManager administra o mapeamento entre uma classe de entidade e uma fonte de dados subjacente. Os Entity Managers são configurados para serem capazes de persistir ou gerenciar tipos específicos de objetos, ler e escrever-los numa base de dados e ser implementado por um Provedor de Persistência (persistence provider) como Hibernate, TopLink, JDO, entre outros.

Quando um Entity Manager obtém uma referência de uma entidade, seja através de um argumento passado numa chamada de método ou porque essa entidade foi lida de uma base de dados, este objeto é dito como gerenciado (managed) por um Entity Manager. O conjunto de instâncias de entidades gerenciadas dentro de um Entity Manager em um dado momento é chamado como contexto de persistência (persistence context). Apenas uma instância Java com a mesma identidade de persistência pode existir em um contexto de persistência em qualquer momento.

Todos Entity Managers vem de fábricas do tipo EntityManagerFactory. A configuração de um Entity Manager é modelado por um EntityManagerFactory que criou-o, mas ele é definido separadamente como uma unidade de persistência (persistence unit). A unidade de persistência ordena implicitamente ou explicitamente as configurações e classes de entidades usada por todos Entity Managers obtidos por uma única instância de um EntityManagerFactory ligada a uma unidade de persistência.

Portanto, existe uma correspondência um-para-um (one-to-one) entre uma unidade de persistência e a sua EntityManagerFactory concreta.

Unidade de Persistência são nomeadas para permitir uma diferenciação de uma EntityManagerFactory para outra. Isto dá controle para aplicação sobre quais configurações ou unidade de persistência é para ser usada por uma operação em uma entidade particular.

Persistence Contexts

Uma unidade de persistência ou persistence-unit é uma configuração nomeada de classes de entidade. Um contexto de persistência é um gerenciamento de entidades onde, todo contexto de persistência é associado com uma unidade de persistência.

Persistence Unit

Um EntityManager mapea um conjunto de classes a um banco de dados particular. Este conjunto de classes é chamado de *persistence unit* (unidade de persistência). Uma unidade de persistência está definida em um arquivo chamado *persistence.xml*. Este arquivo é um descriptor de desenvolvimento exigido no JPA. Um arquivo de *persistence.xml* pode definir um ou mais unidades de persistência. Este arquivo fica situado no diretório META-INF.

Obtendo um Entity Manager

Um Entity Manager é sempre obtido através de um EntityManagerFactory que determina os parâmetros de configuração que vão dizer como será o funcionamento do Entity Manager. O método estático `createEntityManagerFactory()` na classe Persistence retorna o EntityManagerFactory para um nome de unidade de persistência específico.

O nome da unidade de persistência especificada que é passada no método `createEntityManagerFactory()` identifica a unidade de persistência que tem o objetivo de determinar configurações como, por exemplo, os parâmetros de conexão que os Entity Managers gerados nesta fábrica usarão quando se conectar com a base de dados.

Transações

As mudanças nas entidades devem ser feitas persistentes utilizando uma transação. Nas operações de inserção, atualização e deleção devemos utilizar transações, exceto no método `find()` que apenas busca dados. Quando transações não são utilizadas temos como retorno uma exceção ou então nada é feito na base de dados.

JPQL

```

EntityManagerFactory emf = Persistence.createEntityManagerFactory("PersistenceDemoPU");
EntityManager em = emf.createEntityManager();
Query query = em.createQuery("SELECT p FROM Player p", Player.class);
List<Player> playerList = query.getResultList();
System.out.println("Player List:");
for (Player p : playerList) {
    System.out.println(p.toString());
}
em.close();
emf.close();

```



Frameworks

Java Persistence Query Language (JPQL) é uma plataforma independente de linguagem de consulta orientada a objeto, foi definida como parte da especificação Java Persistence API. JPQL é usado para fazer consultas com entidades armazenados em um banco de dados relacional. É fortemente inspirada pelo SQL e suas consultas assemelham as consultas SQL na sintaxe, mas operam em objetos de entidade JPA ao invés de diretamente com as tabelas de banco de dados.

JPQL é usado para definir as pesquisas nas entidades persistentes e são independentes do mecanismo usado para armazenar essas entidades. Como tal, JPQL é “portátil”, e não restrito a nenhum armazenamento de dados particular.

Estrutura de consulta JPQL

A sintaxe da linguagem JPQL é muito semelhante à sintaxe SQL. Ter um SQL como sintaxe em consultas JPA é uma vantagem importante, pois o SQL é uma linguagem de consulta muito poderosa e muitos desenvolvedores já estão familiarizados.

A principal diferença entre SQL e JPQL é que o resultado de uma consulta SQL são registros e campos, ao passo que o resultado de uma consulta JPQL classes e objetos. Por exemplo, uma consulta JPQL pode recuperar e retornar objetos de entidade, em vez de apenas os valores de campo a partir de tabelas de banco de dados, como acontece com SQL. Isso faz objecto JPQL mais orientado amigável e fácil de usar em Java.

Criteria

Escrevendo SQL com JAVA

```
public List<Fornecedor> recuperarFornecedores(String pNome) {
    Session session = (Session) entityManager.getDelegate();

    Criteria criteria = session.createCriteria(Fornecedor.class);
    criteria.add(Restrictions.ilike("nome", pNome, MatchMode.ANYWHERE));
    return criteria.list();
}
```

*Resultado > SELECT * FROM nome like '%pNome%'*



Frameworks

A especificação da JPA define uma nova API para as consultas dinâmicas através da construção de uma instância da classe javax.persistence.CriteriaQuery baseada em objeto, em vez de uma abordagem baseada em strings usado em JPQL (Java PersistenceQuery Language). Esta capacidade de definição de consulta dinâmica é referida como Criteria API, e baseia-se no esquema persistente das entidades, seus objetos incorporados e suas relações.

A sintaxe é projetada para construir uma árvore de comando cujos os nós representam os elementos de consulta semântica, tais como projeções, predicados condicionais de cláusula WHERE ou GROUP BY elementos.

Quando utilizamos o Hibernate para realizar consultas no banco de dados, temos a oportunidade de trabalhar com SQL, com HQL e também com **Criteria Query API**. A API (Application Programming Interface) Criteria do Hibernate fornece uma maneira elegante de construção de query dinâmica para consultas no banco de dados. A API é uma alternativa as consultas HQL (Hibernate Query Language) e também ao SQL tradicional.

Já a API Criteria une as vantagens do HQL com a praticidade de escrever consultas com código Java através da interface **org.hibernate.Criteria**, abolindo assim o uso de longas strings. A interface **Criteria** define os métodos necessários para trabalhar com **Criteria** e por ser um consulta de forma programática, erros no código podem ser sinalizados ainda em tempo de compilação, coisa que com HQL ou SQL não é possível observar.