



# Frameworks



Maven .....	6
O que é maven? .....	6
Instalando o Maven .....	8
Criação de um projeto .....	9
Tipos de Projeto Empocotamento .....	11
Ciclo de Vida Maven .....	12
Empocatamento JAR .....	15
Empocatamento POM.....	16
Empocatamento EJB .....	17
Preparando o Projeto para o Eclipse.....	19
Pom.xml .....	21
Estrutura.....	22
Transitividade de Dependências.....	23
Exclusão de Dependências.....	24
Dependência Opcional Alternativa para Exclusão de Dependências.....	25
POM.xml Projeto Multimodulo.....	26
Dependency Management.....	27
Apache Maven Compiler Plugin .....	28
Plugin m2E Integração do Maven com Eclipse .....	29
Primefaces.....	30
Primefaces Adicionando as Depedências no pom.xml.....	32
Primefaces Configurando o Tema no Web.xml .....	33
Primefaces.....	40
Alguns Elementos do Primefaces Tabela de Dados .....	35
Alguns elementos do Primefaces Mensagens.....	37
Alguns elementos do Primefaces Ajax .....	37
JPA Arquitetura .....	40

Persistence .....	41
Mapeamento de Anotações .....	44
Anotações .....	45
Geração Automática de Chaves Primárias IDENTITY .....	47
Geração Automática de Chaves Primárias SEQUENCE.....	48
Geração Automática de Chaves Primárias TABLE .....	49
Geração Automática de Chaves Primárias AUTO .....	50
Anotações de Relacionamento .....	51
One to One Unidirecional .....	52
One to One bidirecional .....	53
One to Many Unidirecional .....	54
One to Many ndo Join Column .....	55
One to One bidirecional .....	56
Many to Many .....	57
Tipos de Transação .....	58
JTA .....	59
Entity Manager .....	60
Estados de um Entidade .....	62
EntityManager Criar .....	63
EntityManager Recuperar .....	64
EntityManage Update .....	55
EntityManage Deletar .....	66
Cascade .....	68
OneToOne, OneToMany e ManyToMany.....	69
FetchType .....	70
JPQL Java Persistance Query Language.....	71
JPQL Recuperar .....	70

JPQL Update .....	73
JPQL Deletar .....	74
Named Queries.....	75
Native Query .....	76
CRITERIA .....	77
CRITERIA Query .....	79
<b>CDI Context and Dependency Injection.....</b>	<b>83</b>
Inversão Controle.....	85
Qualificadores - Ambiguidade de Objetos .....	86
Beans .....	87
Ponto de Injeção - No campo .....	88
Ponto de Injeção - No Construtor .....	89
Ponto de Injeção - No Método de Configuração .....	90
Qualificadores - Ambiguidades de Objetos .....	91
Qualificadores - Desambiguidade .....	92
Qualificadores - Criando Anotações Qualificadoras.....	93
Qualificadores - Usando Qualificadores .....	94
Alternatives .....	95
Injeção a partir métodos Produtores .....	96
Injeção de EJB a partir de Produtores.....	97
Desambiguação de Produtores .....	98
Alternative Usando Esterótipo.....	100
Interceptadores Anotação .....	101
Produtores e Escopo .....	102
Qualificadores - Criando Anotações Qualificadoras .....	103
Managed Beans .....	104
Escopos CDI .....	105

Interceptadores .....	107
CDI e JPA/EJB .....	108
Injeção a partir Métodos Produtores .....	109
Injeção de EJB a partir de Produtores .....	110
Decorators - Decorando seu Bean .....	111
Decorators - Formando html .....	112
CDI e JSF .....	113
Qualificadores - Desambiguidade .....	114
CDI e JPA/EJB .....	115
<b>Logging.....</b>	<b>117</b>
Log4j .....	118
Dependência do Log4j .....	119
Log4j. xml Exemplo de Configuração .....	120
Usando um Objeto Logger .....	121
Appenders .....	122
Nível de Log .....	124
SLF4J .....	126
Dependência do SLF4J com Log4J .....	127

## Maven



Maven é uma ferramenta de build open source para Java EE, projetada para fazer o trabalho mais complicado do processo de build de um projeto. O Maven deixou de ser uma simples ferramenta de construção para se tornar 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.

- O principal componente do maven é o project object model (conhecido como POM). Nele estão contidas descrições detalhadas do seu projeto, incluindo informações sobre versionamento, gerencia de configuração, dependências, recursos da aplicação e de testes, membros da equipe e estrutura, entre outras informações.
- Dependency management model - A gestão de dependências é uma parte do processo de construção de software, que muitas vezes são ignoradas 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. É por meio deste componente que plugins são gerenciados e passam a seguir um ciclo de vida bem definido.
- Plugins – Estendem as funcionalidades do Maven.

Alguns dos principais objetivos do Maven são:

- Simplificar o processo de build;
- Disponibilizar um sistema de build uniforme;
- Padronizar o fornecimento de informações do projeto;
- Criação de diretrizes para o desenvolvimento de boas práticas;
- Permitir a migração transparente de novas funcionalidades.

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, metadados ou através da criação de plugins);
  - É extensível;
  - Está em constante evolução, incorporando novas práticas e funcionalidades identificadas como comuns em comunidades de usuários.

## Instalando o Maven

Execute o comando para instalar a versão mais atual do maven

```
sudo apt-get install maven
```

Execute o comando `mvn --version` para verificar a versão instalada

```
Apache Maven 3.0.5
Maven home: /usr/share/maven
Java version: 1.7.0_80, vendor: Oracle Corporation
Java home: /usr/lib/jvm/java-7-oracle/jre
Default locale: pt_BR, platform encoding: UTF-8
OS name: "linux", version: "3.16.0-44-generic", arch: "amd64", family: "unix"
```

## Instalação

No debian Linux, basta executar o comando:

```
sudo apt-get install maven
```

No Windows, precisamos desempacotar o Maven em algum diretório (faça o download em <https://maven.apache.org/download.cgi>), c:\apache-maven-3.0.5 por exemplo. Feito isso, podemos invocar o maven via linhas de comando através do mvn.bat localizado no diretório bin. Para executar os comandos, c:\apache-maven-3.0.5\bin deve estar em seu PATH (variável de ambiente), assim como a J2SE SDK.

Para verificar qual versão foi instalada, execute o comando:

```
mvn –version
```

## Criação de um projeto

### GroupId

ID do grupo do projeto.  
Identificador único de  
uma organização

### Atributos Obrigatórios

```
mvn archetype:create  
-DgroupId=[group id do seu projeto]  
-DartifactId=[artifact id do seu projeto]  
-DarchetypeArtifactId=maven-archetype-webapp
```

### ArtifactID

ID do projeto. Nome  
do projeto.

Precisamos criar um diretório para seu projeto, feito isso, na linha de comando, execute o seguinte comando e os suas metas (Maven goal):

```
mvn archetype:generate -DgroupId=com.mycompany.app -DartifactId=-my-app  
-DarchetypeArtifactId=maven-archetype-quickstart
```

O primeiro parâmetro do comando indica que vamos utilizar o plugin chamado archetype. Este plugin permite que o usuários crie um projeto Maven a partir de um modelo pré-programado. Em seguida definimos o que é chamado de meta. O archetype possui quatro opções de metas:

- **archetype:create** – (deprecated) cria um projeto Maven a partir de um modelo. Compatível com a versão 1.0-alpha-7 do Archetype Plugin.
- **archetype:generate** – cria um projeto Maven a partir de um modelo, guiando o usuário através de um wizard que fará as configurações necessárias.
- **archetype:create-from-project** – cria ma archetype a partir de um projeto existente.
- **archetype:crawl** – busca no repositório por archetypes e atualiza o catálogo.

Metas (goals) são o que são executados para realizar uma ação no projeto. Por exemplo, a meta jar:jar vai compilar o projeto e produzir um jar. Cada meta existe em um plugin, e o nome da meta geralmente reflete o plugin (java:compile vem do plugin java, por exemplo). Segue a sintaxe:

```
mvn [nome-plugin]:[nome-goal]
```

Como visto acima, a criação de um projeto exige alguns atributos obrigatórios:

- **Group ID** – é um identificador único e universal de um projeto. Apesar de geralmente ser apenas o nome do projeto, ajuda se utilizarmos um nome de pacote qualificado para distinguir outros projetos com nomes parecidos.
- **Artifact ID** – um artefato é algo utilizado ou produzido por um projeto. Exemplos de artefatos produzidos pelo Maven incluem JARs, distribuições binárias e fontes, WARs. Cada artefato é unicamente identificado pelo Group ID e um Artifact ID, que é único dentro de um grupo.

-DarchetypeArtifactId se refere aos arquétipos que o Maven fornece:

- **maven-archetype-archetype** – um arquétipo para gerar um sample de um arquétipo.
  - **maven-archetype-j2ee-simple** – um arquétipo para gerar uma aplicação J2EE simplificada.
  - **maven-archetype-plugin** – um arquétipo para gerar um sample de um plugin Maven.
  - **maven-archetype-plugin-site** – um arquétipo para gerar um sample de um site de um plugin maven.
  - **maven-archetype-quickstart** – um arquétipo para gerar um sample de um projeto Maven.
  - **maven-archetype-site** – um arquétipo para gerar um sample de um site Maven, o qual demonstra um pouco dos tipos de documentos como APT, Xdoc, FML e demonstra como fazer a internacionalização (i18n) do seu site.
  - **maven-archetype-webapp** – um arquétipo para gerar um sample de um projeto Maven Web.

## Tipos de projeto empacotamento

Empacotamento para projetos libs (padrão do maven)

```
<packaging>jar</packaging>
```

Empacotamento para projetos modulares

```
<packaging>pom</packaging>
```

Empacotamento para projetos EJB

```
<packaging>ejb</packaging>
```

Empacotamento para projetos web

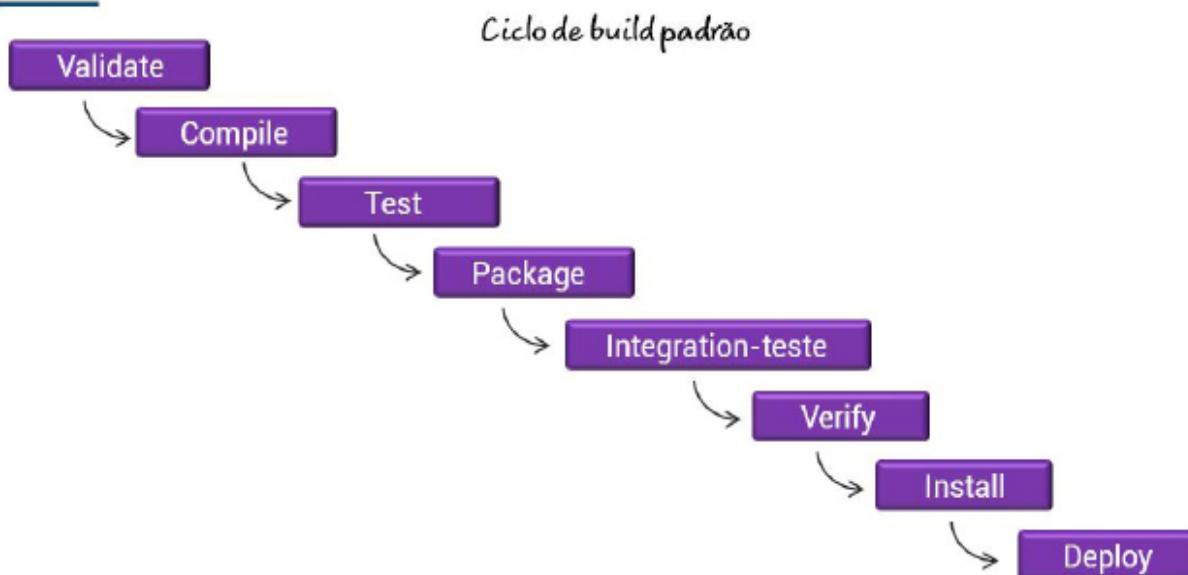
```
<packaging>war</packaging>
```

Os tipos de pacotes mais comuns em projetos Java são:

- **JAR** – empacotamento para projetos libs (padrão Maven);
- **POM** – empacotamento para projetos modulares;
- **EJB** – empacotamento para projetos EJB;
- **WAR** – empacotamento para projetos WEB.

O empacotamento escolhido para seu projeto afeta os passos requeridos para construir o projeto (build).

## Ciclo de Vida Maven



Executar `mvn test`, por exemplo, fará com que todos as fases anteriores também sejam executadas.

O Maven é baseado no conceito central de um ciclo de vida de construção. Um ciclo de vida maven consiste em uma sequência de fases. Quando você manda o Maven construir um projeto, você está dizendo para o Maven passar por um sequência de fases e executar qualquer meta que foi registrada em cada fase.

Existem três ciclos de vida no Maven: clean, default e site.

Ciclo de vida clean

É o mais simples entre os ciclos de vida, lida com a limpeza do projeto, consistindo em apenas três fases:

Fase do ciclo de vida	Descrição
Descrição	Executa os processos necessários antes da geração do projeto.
site	Gera a documentação do projeto.
post-site	Executa os processos necessários para finalizar a geração, e se preparar para a implantação.
site-deploy	Implanta a documentação do site gerado para o servidor Web especificado.

As metas associadas ao ciclo de vida site são:

- site -site:site
- site-deploy -site:deploy

Para gerar um site de um projeto Maven, basta executar:

```
mvn site
```

Ciclo de vida default

As fases do ciclo de vida de um projeto maven padrão são:

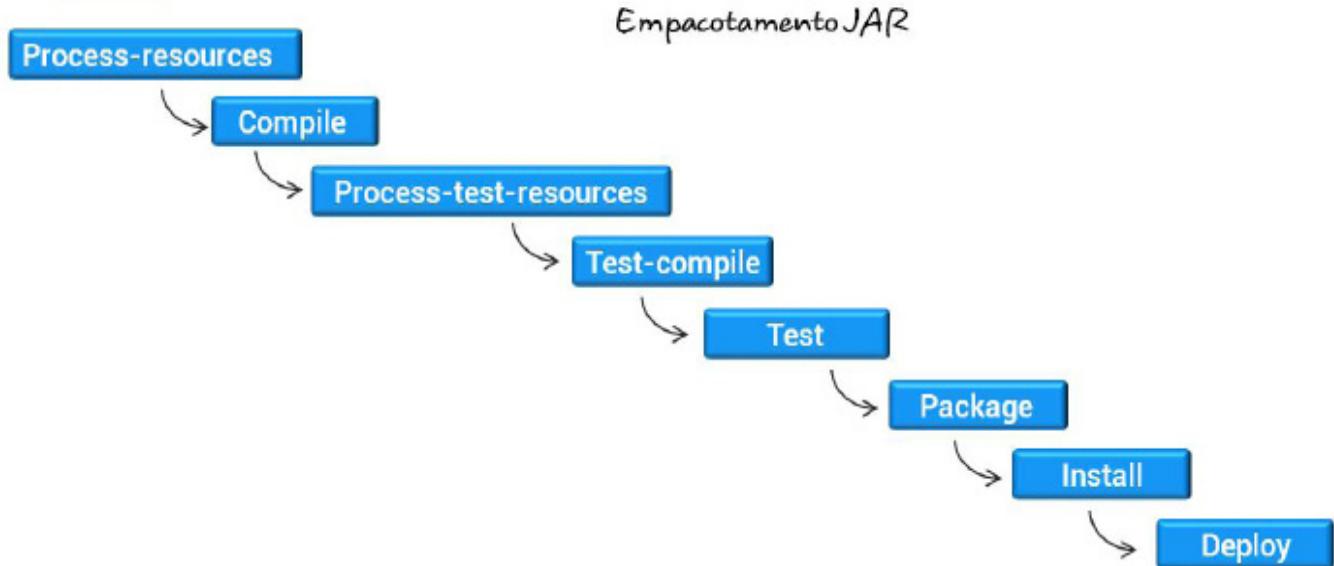
<b>Fase do ciclo de vida</b>	<b>Descrição</b>
<b>validate</b>	Valida o projeto e todas as informações necessárias para completar a construção.
<b>generate-sources</b>	Gera qualquer código fonte para inclusão na compilação.
<b>process-sources</b>	Processa o código fonte
<b>compile</b>	Compila o código fonte do projeto.
<b>process-classes</b>	Pós-processa os arquivos gerados a partir da compilação, para fazer melhorias no bytecode nas classes, por exemplo.
<b>generate-test-sources</b>	Gera qualquer código fonte de testes para inclusão na compilação.
<b>process-test-sources</b>	Processa o código fonte dos testes.
<b>generate-test-resources</b>	Criar os recursos para realizar os testes.
<b>process-test-resources</b>	Copia e processa os recursos para o diretório destinado aos testes.
<b>test-compile</b>	Compila o código fonte dos testes no diretório destinado.
<b>test</b>	Executa os testes usando um framework de testes conveniente. Esses testes não devem requerer que o código esteja empacotado ou implantado.
<b>prepare-package</b>	Realiza qualquer operação necessária para preparar um pacote antes do real empacotamento. Isso geralmente resulta em uma versão processada e desempacotada do pacote.
<b>package</b>	Pega o código compilado e empacota-o no seu formato de distribuição (JAR, WAR, EAR, etc).
<b>pre-integration-test</b>	Realiza ações necessárias antes da execução dos testes de integração. Pode envolver coisas como a configuração do ambiente necessário.

Fase do ciclo de vida	Descrição
<b>post-integration-test</b>	Realiza ações requeridas depois da execução dos testes de integração. Pode envolver a limpeza do ambiente.
<b>verify</b>	Executa qualquer verificação para verificar se o pacote é válido e encontra os critérios de qualidade.
<b>install</b>	Instala o pacote no repositório local, para uso como dependência em outros projetos localmente.
<b>deploy</b>	Copia o pacote final para o repositório remoto para que seja compartilhado com outros desenvolvedores e projetos.

Note que existem 21 fases no ciclo de vida padrão, sendo que a fase “prepare-package” não será usada em versões do maven anteriores a 2.1. Nós podemos invocar uma fase específica desse ciclo de vida, como “mvn compile”. Isso fará com que todas as fases anteriores a ela também sejam executadas. É importante destacar que, dependendo do tipo de empacotamento do projeto, diferentes metas serão limitadas a diferentes fases do ciclo de vida.

As tabelas abaixo mostram quais metas são mapeadas para cada fases do ciclo de vida, dependendo do tipo de empacotamento.

## Ciclo de Vida Maven



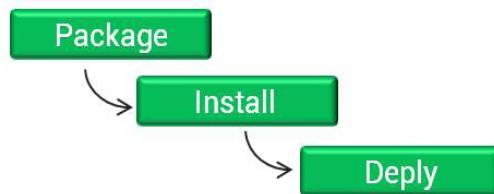
Executar `mvn test`, por exemplo, fará com que todos as fases anteriores também sejam executadas.

O empacotamento JAR é tipo padrão, seu ciclo de vida pode ser visto na tabela abaixo:

Fase do ciclo de vida	Goal
process-resources	resource:resource
compile	compiler:compile
process-test-resources	resource:testResource
test-compile	compiler:testCompile
test	surefire:test
package	jar:jar
install	install:install
deploy	deploy:deploy

## Ciclo de Vida Maven

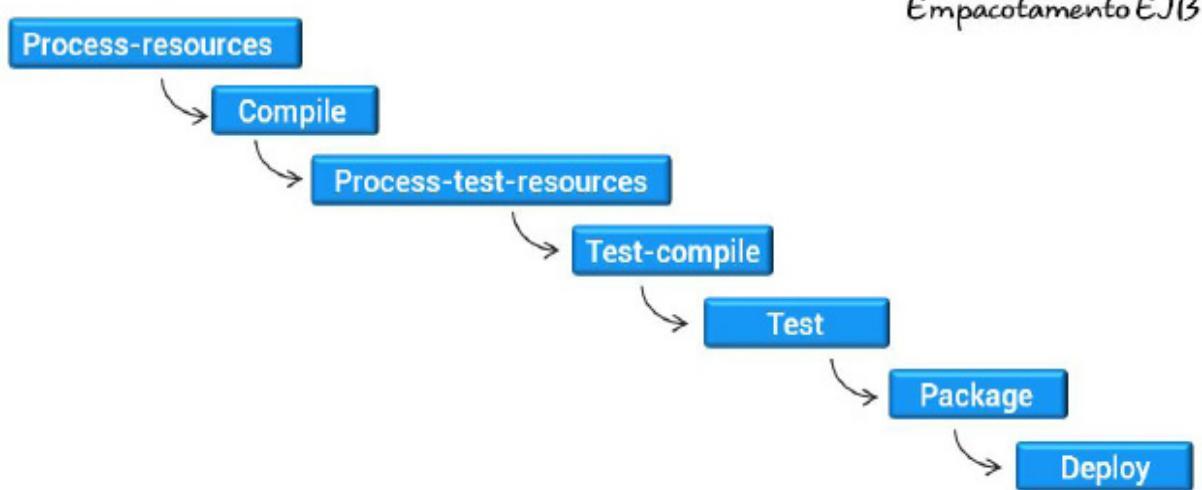
Empacotamento POM



O empacotamento do tipo POM é o mais simples. O artefato que gerado é ele mesmo, não há código para ser compilado ou testado e também não há recursos para processar. Seu ciclo de vida pode ser visto na tabela abaixo.

Fase do ciclo de vida	Goal
package	site:attach-descriptor
install	install:install
deploy	deploy:deploy

## Ciclo de Vida Maven

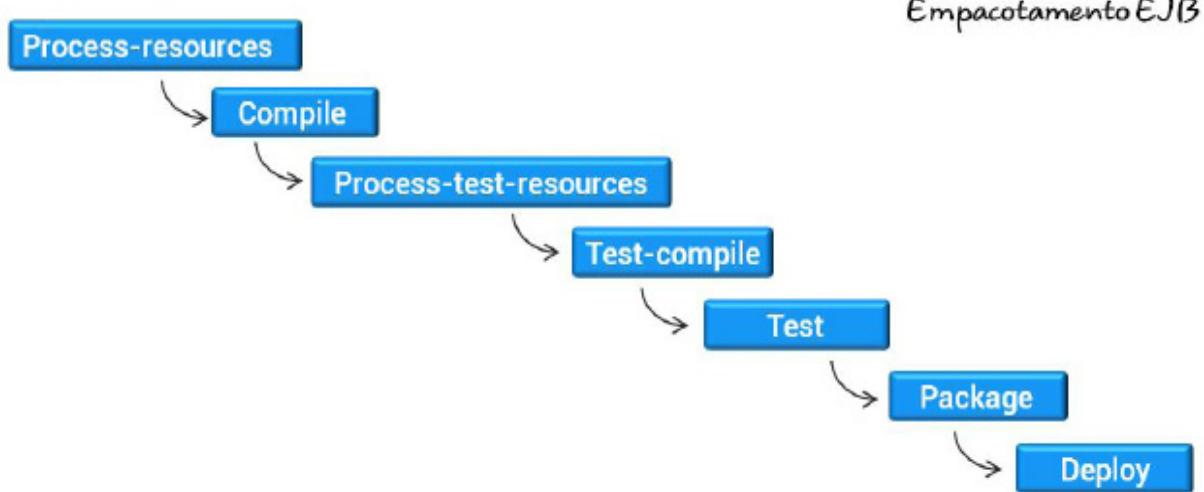


Executar `mvn test`, por exemplo, fará com que todas as fases anteriores também sejam executadas.

EJBs, ou Enterprise Java Beans, são um mecanismo de acesso de dados bastante comum para desenvolvimento orientado a modelos em Enterprise Java. O Maven fornece suporte para EJB 2 e 3. Seu ciclo de vida pode ser visto na tabela abaixo.

Fase do ciclo de vida	Goal
process-resources	resources:resources
compile	compiler:compile
process-test-resource	resource:testResource
test-compile	compiler:testCompile
test	surefire:test
package	ejb:ejb
deploy	deploy:deploy

## Ciclo de Vida Maven



Executar `mvn test`, por exemplo, fará com que todas as fases anteriores também sejam executadas.

O empacotamento WAR é similar com os tipos EJB e JAR. A diferença está na meta da fase package. A meta war:war necessita de uma configuração web.xml no diretório WEB-INF.

Fase do ciclo de vida	Goal
process-resources	resources:resources
compile	compiler:compile
process-test-resources	resources:testResources
test-compile	compiler:testCompile
test	surefire:test
package	war:war
install	install:install
deploy	deplo:deploy

## Preparando o projeto para o Eclipse

Comando

```
mvn eclipse:eclipse
```

Caso seja um projeto web

```
mvn eclipse:eclipse -Dwtpversion=2.0
```

No Eclipse:

Arquivo > Import... > Maven > Existing Maven Projects



Ao criar um projeto maven, um arquivo chamado pom.xml será gerado.

O Maven Eclipse Plugin é usado para gerar arquivos da IDE do Eclipse (\*.classpath, \*.project, \*.wtpmodules e a pasta .settings).

Para gerar esses arquivos, devemos executar o seguinte comando:

```
mvn eclipse:eclipse
```

```
mvn eclipse:eclipse -Dwtpversion=2.0 (para projetos web)
```

Feito isso, importe o projeto de dentro do eclipse.

O Maven Eclipse Plugin possui algumas metas:

eclipse:configure-workspaces – é usado para adicionar a variável de classpath M2\_REPO no Eclipse, que aponta para o seu repositório local e opcionalmente configura características de outros workspaces.

Eclipse:eclipse – gera os arquivos de configuração do Eclipse.

Eclipse:resolve-workspace-dependencies – é usado para fazer o download todos os elementos da variáveis de classpath M2\_REPO para todos os projetos em um workspace. É usado se os arquivos de configuração do projeto Eclipse são enviados para o controle de versão e outros usuários precisam resolver novos artefatos depois de uma atualização.

Eclipse:clean – é usado para deletar os arquivos usados pela IDE Eclipse.

O Eclipse precisa saber o caminho para o repositório maven local. Portanto o classpath da variável M2\_REPO precisa estar configurado. Execute o seguinte comando:

```
mvn -Dworkspace=<caminho-para-workspace> eclipse:add-maven-repo
```

Você pode definir uma nova variável de classpath de dentro do Eclipse:  
window > preferences>java>Build Path>Classpath Variables.

Atualmente os usuários são recomendados a usar o m2e, Eclipse Maven Integration, ao invés do Maven Eclipse Plugin, já que ele pode se aproximar mais do tempo de compilação e tempo de execução dos classpaths reais, conforme descrito no pom.xml do projeto. Entretanto, existem algumas configurações de projeto e workflows que ainda funcionam de forma mais eficiente com metadata do eclipse gerada estaticamente.

## Pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>threeway.projeto.maven</groupId>
  <artifactId>Exemplo-Maven</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>Exemplo Maven</name>
  <description>Descrição do projeto</description>

  <dependencies>
    <!-- Aqui ficam as dependências do projeto -->
  </dependencies>
</project>
```

Ainda podemos ter mais propriedades além das dependências, como configurações de build, informações do projeto (nome, url, desenvolvedores, etc.) e configurações do ambiente (repositórios, perfis, etc.).

Mais informações sobre o POM  
<https://maven.apache.org/pom.html>

## Project Object Model – pom.xml

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 POM 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 de seu pai. O próprio Maven provê um POM que está no topo da hierarquia e, portanto define valores default para todos os projetos.

Uma das características do Maven que mais se destaca é o gerenciamento de dependências. Gerenciar dependências de um único projeto não é muito trabalhoso, mas quando possuímos um projeto com vários módulos, o Maven pode ajudar a manter um alto nível de controle e estabilidade sobre as dependências.

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

Artefatos encontrados em repositórios globais 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)

## Estrutura

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <!-- Relações do POM -->
  <groupId>threeway.maven</groupId>
  <artifactId>ExemploMaven</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>

  <!-- Informações do Projeto -->
  <name>ExemploMaven</name>
  <url>http://maven.apache.org</url>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>

    <!-- Outras dependências -->
  </dependencies>

  <modules> </modules>
```

## Estrutura

```
<!-- Informações do Projeto -->
<description> </description>
<inceptionYear> </inceptionYear>
<licenses> </licenses>
<developers> </developers>
<contributors> </contributors>
<organization> </organization>

<!-- Configurações de Build -->
<build> </build>
<reporting> </reporting>

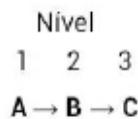
<!-- Informações de Ambiente -->
<issueManagement> </issueManagement>
<ciManagement> </ciManagement>
<mailingLists> </mailingLists>
<scm> </scm>

<!-- Ambiente do Maven -->
<prerequisites> </prerequisites>
<repositories> </repositories>
<pluginRepositories> </pluginRepositories>
<distributionManagement> </distributionManagement>
<profiles> </profiles>

</project>
```

O POM 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 da vida ao projeto. De fato, para o Maven, um projeto não precisa ter mais do que um simples arquivo pom.xml.

## **Transitividade de Dependências**



C é dependência transitiva de A

### **Problema: mesmo Artifact-Id e Group-Id**

- Primeiro mais Próximo**

Dependências de um nível mais alto tem prioridade sobre as de níveis mais baixos.  
Dependência direta tem prioridade sobre uma transitiva.

- Primeiro Encontrado**

Em um mesmo nível, a primeira dependência encontrada é usada  
(mesmo group-id/artifact-id e diferentes versões).

Dependências transitivas estão presentes desde a versão 2.0 do Maven. Isso permite que você evite ter que descobrir e especificar bibliotecas que suas próprias dependências precisam, então essas bibliotecas serão inclusas automaticamente.

Essa característica é facilitada com a leitura dos arquivos de projeto de suas dependências de repositórios remotos especificados. No geral, todas as dependências desses projetos são usadas em seu projeto. Não há limite para o número de níveis que as dependências podem ser agrupadas, causando problemas apenas se houver alguma dependência cíclica.

Mecanismos para limitar quais dependências são incluídas.

**Mediação de Dependências** – determina qual versão de um dependência será usada quando multiplas versões de um artefato são encontradas.

**Gerenciamento de Dependências** – permite que o autor do projeto especifique diretamente qual versão de um determinado artefato será usado.

**Escopo de Dependências** – permite que você apenas inclua dependências apropriadas para a fase atual da build.

**Exclusão de Dependências** – permite excluir dependências explicitamente.

**Dependências Opcionais** – permite que você defina um dependência como opcional ao invés de excluí-la.

## Exclusão de Dependências

```
<exclusions>
  <exclusion>
    <artifactId> </artifactId>
    <groupId> </groupId>
  </exclusion>
</exclusions>
```

- ✓ Conflitos entre versões de um mesmo artefato.
- ✓ Conflito entre artefatos do seu projeto e artefatos da plataforma de implantação.
- ✓ Excluir dependências transitivas não desejadas.

### Exclusão de Dependências

A exclusão de dependências tem o objetivo de eliminar dependências transitivas indesejáveis. Na maioria das vezes você irá utilizar a exclusão para tratar conflitos entre versões de um mesmo artefato ou conflitos entre artefatos do seu projeto e artefatos da plataforma de implantação.

Funciona da seguinte forma:

Supondo um projeto A que depende de B, e B depende de C, o dono do projeto A pode explicitamente excluir o projeto C das suas dependências com a tag `<exclusion>`.

## Dependência Opcional Alternativa para exclusão de dependências

Projeto A depende de B

```
<dependencies>
    <dependency>
        <groupId>threeway.opcional</groupId>
        <artifactId>ProjetoB</artifactId>
        <version>0.0.1-SNAPSHOT</version>
        <optional>false</optional>
    </dependency>
</dependencies>
```

Tag `optional`

Projeto B depende de C

```
<dependencies>
    <dependency>
        <groupId>threeway.opcional</groupId>
        <artifactId>ProjetoC</artifactId>
        <version>0.0.1-SNAPSHOT</version>
        <optional>true</optional>
    </dependency>
</dependencies>
```

Impede que as dependências de C se propaguem  
transitivamente para projetos dependentes de B (projeto A)

### Dependências Opcional

As dependências opcional são usadas quando não podemos separar um projeto (qualquer seja o motivo) em submódulos.

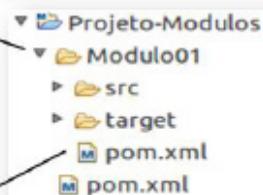
Um outro meio de excluir dependências transitivas é através da tag optional. Supondo que tenhamos os projetos A, B e C, e que B depende de C, podemos fazer com que B marque o projeto C como uma dependência opcional, isso não afetará uma relação direta entre B e C, mas quando A adicionar B como sua dependência, a dependência opcional terá efeito. C não será incluso no classpath de A, a não ser que você declare C diretamente no POM do projeto A.

Declarar dependências opcionais pode ser importante para salvar memória/espaço. É importante controlar a lista de dependências que são realmente necessárias para usar em um projeto, já que esses jars serão empacotados de acordo com o tipo do projeto e a inclusão de jars errados podem causar problemas.

## POM.xml Projeto multimodulo

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>threeway.projetomaven</groupId>
  <artifactId>Projeto-Modulos</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>pom</packaging>
  <modules>
    <module>Modulo01</module>
  </modules>
</project>
```

Empacotamento deve ser do tipo pom



Projeto pai passa para seus módulos: Group id, Version, Dependencias, Plugins

### Projeto Multimódulo

Um projeto multimódulo é definido por um POM pai que referencia um ou mais subprojetos (módulos). Note que o empacotamento do projeto pai deve ser do tipo pom. O projeto pai possui o que chamamos de super pom, o qual possui a característica de herança, ou seja, os módulos dependentes desse super pom herdam algumas informações como:

- Dependências
- Propriedades
- Configurações
- Recursos

## Dependency Management

```
<modules>
  <module>Modulo01</module>
</modules>

<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
    </dependency>
  </dependencies>
</dependencyManagement>
```

Dependência no Modulo01

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
  </dependency>
</dependencies>
```

A versão é especificada  
no pom pai

Permite melhor controle de versão de dependências em projetos multimodulos

A tag dependency-management é um mecanismo para centralizar as dependências. Permite que os autores do projeto especifiquem diretamente as versões dos artefatos a serem usados quando são encontradas dependências transitivas ou em dependências onde nenhuma versão foi especificada.

Em projetos modulares, as dependências podem ser definidas no POM pai, sendo herdadas pelos POM filhos quando for necessário. Tendo uma única fonte para todas as dependências torna o controle de versão das mesmas muito mais simples.

## Apache Maven Compiler Plugin

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.3</version>
      <configuration>
        <source>1.8</source>
        <target>1.8</target>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Indica qual versão da JDK o projeto será compilado.

Esse plugin é usado para compilar os fontes de seu projeto. Por padrão, ele é chamado implicitamente pelo ciclo de vida do Maven na fase apropriada, portanto, não precisamos necessariamente defini-lo no pom.xml.

Apesar disso, ainda podemos defini-lo no pom com finalidade de configurar a forma como o maven deve compilar nossas classes.

Podemos configurar uma versão diferente da JDK para compilar as classes.

Podemos configurar para que as classes compiladas sejam compatíveis com uma JDK específica.

## Plugin m2E Integração do maven com Eclipse



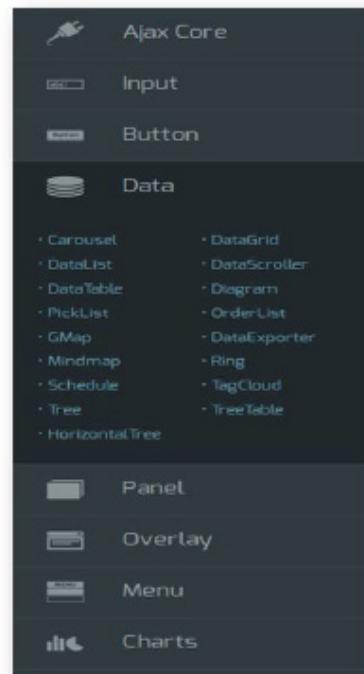
O objetivo do m2e é fornecer um suporte ao Maven Apache na Eclipse IDE, facilitando a edição do pom.xml, executar uma build a partir da IDE entre outras coisas:

- Criar e importar projetos Maven;
- Gerenciamento de dependências e integração com o classpath do Eclipse
- Download e update automático de dependências
- Resolução de artefatos Javadoc e fontes
- Buscar e navegar repositórios maven remotos
- Gerenciamento do POM atualização automática da lista de dependências
- Materializar um projeto a partir do Maven POM
- Adaptar projetos Maven modulares aninhados para a Eclipse IDE
- Algumas versões da IDE Eclipse já vem com esse plugin integrado.

## Primefaces

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

*Primefaces oferece uma documentação de fácil acesso e com exemplos de como usar cada um de seus componentes*



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 2, e permite que sejam inseridos em seu conjunto, outros componentes através da 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.

PrimeFaces é uma biblioteca de componentes de código aberto para o JSF 2 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 usar PrimeFaces:

- Possui um rico conjunto de componentes de interface (DataTable, AutoComplete, HTMLEditor, gráficos, etc).
  - Nenhum xml de configuração extra é necessário.
  - Componentes construídos com Ajax no padrão JSF 2 Ajax APIs.
  - Mais de 25 temas templates.
  - Boa documentação com exemplos de código.
  - “Fácil de usar”, os componentes são desenvolvidos com 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 empresas de software, bancos, instituições financeiras, universidades e muito mais.

- Oferece o Mobile UI Kit para criar aplicações web móveis para dispositivos portáteis baseados em navegadores webkit.

## Primefaces Adicionando as dependências no pom.xml

```
<repositories>
    <repository>
        <id>prime-repo</id>
        <name>PrimeFaces Maven Repository</name>
        <url>http://repository.primefaces.org</url>
        <layout>default</layout>
    </repository>
</repositories>

<dependency>
    <groupId>org.primefaces</groupId>
    <artifactId>primefaces</artifactId>
    <version>4.0</version>
</dependency>

<dependency>
    <groupId>org.primefaces.themes</groupId>
    <artifactId>all-themes</artifactId>
    <version>1.0.10</version>
</dependency>
```

Para utilizar o PrimeFaces em seu projeto, basta configurar o seu pom.xml com as dependências referentes ao PrimeFaces e adicionar a taglib JSF em sua página, usando xml namespace.

**xmlns:p="http://primefaces.org/ui"**

Para usar o PrimeFaces Mobile use o namespace

**xmlns:p="http://primefaces.org/mobile"**

A partir da versão 4.0, o primefaces pode ser encontrado no repositório Maven Central, para as versões anteriores e para os pacotes de temas é necessário adicionar o repositório do primefaces dentro do pom.xml de seu projeto ou no arquivos .m2/settings.xml.

Isso é tudo que precisa ser feito. Note que o prefixo p é apenas um link simbólico, podendo ser utilizado qualquer outro caractere para referenciar os componentes do PrimeFaces. Você já pode criar sua primeira página com componentes PrimeFaces:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:p="http://primefaces.org/ui">
<h:head></h:head>
<body>
<p:outputLabel value="JSF22VV/Primefaces está trabalhando"/>
</body>
</html>
```

## Primefaces Configurando o tema no web.xml

```
<context-param>
    <param-name>primefaces.THEME</param-name>
    <param-value>bootstrap</param-value>
</context-param>
```

Tema escolhido

Tag Library do primefaces

```
xmlns:p="http://primefaces.org/ui"
```

Como citado anteriormente, o PrimeFaces oferece alguns temas que devem ser configurados no web.xml, passando o parâmetro primefaces.THEME para contexto.

Cada componente é estilizado com CSS e contains duas camadas: estilos específicos do componente e skinning ou estilos independentes de componente.

Para customizar alguma classes CSS do PrimeFaces você deve garantir que seu CSS seja carregado depois do PrimeFaces. Você consegue isso colocando `<h:outputStylesheet>` para referenciar seu arquivo CSS dentro `<h:body>` ao invés do `<h:head>`:

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

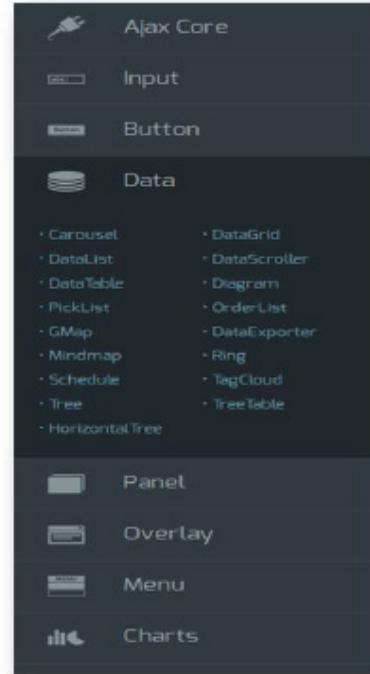
Além disso você deve localizar as classes a serem sobrepostas com ajuda de algum web browser debug com o FireBug, por exemplo :

```
.ui-message-info, .ui-message-error, .ui-message-warn, .ui-message-fatal {
    border: 1px solid;
    margin: 0px 5px;
    padding: 2px 5px;
}
.ui-messages-error, .ui-message-error {
    color: #D8000C;
    background-color: #FFBABA;
}
```

## Primefaces

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

*Primefaces oferece uma documentação de fácil acesso e com exemplos de como usar cada um de seus componentes*



O PrimeFaces tem uma documentação com muitos exemplos práticos que facilitam a aplicação de componentes, estes exemplos podem ser encontrados na showcase do PrimeFaces no link: <http://www.primefaces.org/showcase>.

Os componentes PrimeFaces fornece recurso de Renderização Parcial da Página e view-processing que te permitem escolher o que será processado no ciclo de vida do JSF e o que será renderizado ao fim de uma requisição AJAX. O framework AJAX do PrimeFaces é baseado na API padrão do JSF 2. No entanto, no lado cliente, os JavaScripts do PrimeFaces são implementados na biblioteca JQuery.

Como isso funciona, veja o exemplo:

```
<p:commandButton update="display" action="#{mb.updateValue}" value="Update"/>
<h:outputText id="display" value="#{mb.value}" />
```

O atributo update informa que somente o componente cujo id="display" deve ser renderizado, se você quiser atualizar mas componentes basta referencia-los na atributo update.

```
<p:commandButton update="display1, display2, display3" ... />
```

Além disso tem palavras reservadas que podem ser usada no lugar do ID para atualização parcial (AP):

**@this**: componente que dispara a AP é atualizado;

**@parent**: o pai que dispara AP atualização é atualizado

**@form**: o form que encapsula o componente que dispara a AP é atualizado

**@none**: a AP não altera o DOM com a resposta AJAX

**@all**: todo o documento é atualizado como nas requisições sem AJAX.

## Alguns elementos do Primefaces Tabela de Dados

```
<p: dataTable id="primeTable" value="#{clienteData.clienteList}">
    <var="cliente" style="width: 300px">
        <p:column headerText="Cliente" style="text-align: center">
            <h:outputText value="#{cliente.nome}" />
        </p:column>
        <p:column headerText="Código" style="text-align: center">
            <h:outputText value="#{cliente.codigo}" />
        </p:column>
    </p: dataTable>
```

Cliente	Código
Cliente 01	1
Cliente 02	2
Cliente 03	3

A tabela de dados é um componente bastante comum. Podemos utilizá-la com a tag <p: dataTable> e ter uma lista com os dados vinda de um bean, passada pelo atributo value. Para configurar as colunas da tabela, utilizamos a tag <p: column> e os atributos headerText para dar um título e style para inserir alguma configuração de CSS. Dentro da tag <p: column> temos um <h: outputText> que irá exibir o dado desejado, configurado no atributo value.

Um dos recursos mais interessantes deste componente paginação, permitindo que você visualize toneladas de dados fazendo “carga preguiçosa”, para tanto o atributo lazy deve ser true, e atributo value deve ser vinculado a uma herança da classe org.primefaces.model.LazyDataModel, o Grow chama o método sobrecarregado load()

### <h:form>

```
<h:form>
    <p: dataTable var="car" value="#{dtPaginatorView.lazyModel}">
        <rows="10">
        <paginator="true">
        <lazy="true">
        <paginatorTemplate="{CurrentPageReport} {FirstPageLink} {PreviousPageLink} {PageLinks} {NextPageLink} {LastPageLink} {RowsPerPageDropdown}">
        <rowsPerPageTemplate="5,10,15">
    </p: dataTable>
</h:form>
```

## Alguns elementos do Primefaces Mensagens

 Info Mensagem informando algo!

Mensagem Informativa

Mensagem de Erro

Mensagem de Aviso

Mensagem Fatal

 Aviso! Isso é um aviso!

Mensagem Informativa

Mensagem de Erro

Mensagem de Aviso

Mensagem Fatal

 Erro! Ocorreu um erro!

Mensagem Informativa

Mensagem de Erro

Mensagem de Aviso

Mensagem Fatal

 Fatal! Parou de funcionar!

Mensagem Informativa

Mensagem de Erro

Mensagem de Aviso

Mensagem Fatal

O PrimeFaces possui dois componentes de mensagem vinculados ao mecanismo de mensagens do FacesMessages do JSF. O componente **Growl** recebe as mensagens adicionadas por **FacesContext.getCurrentInstance().addMessage()**

```
public void sendMessage() {  
    FacesContext context = FacesContext.getCurrentInstance();  
    context.addMessage(null,  
        new FacesMessage(FacesMessage.SEVERITY_INFO, "Sua mensagem: " +  
        msg));  
    context.addMessage(null,  
        new FacesMessage(FacesMessage.SEVERITY_INFO, "Additional Message  
        Detail"));  
}
```

```
<h:form>  
    <p:growl id="growl" showDetail="true" sticky="true" />  
    <p:commandButton value="Save"  
        actionListener="#{growlView.mensagem}" update="growl" />  
</h:form>
```

O Growl e **<p:messages/>** são de uso geral enquanto o componente **<p:message for="id"/>** é vinculada a um componente.

## Alguns elementos do Primefaces Mensagens

```
<h:panelGrid id="panel" columns="3" cellpadding="5" style="">
    <p:outputLabel value="nome" for="name"/>
    <p:inputText id="name" required="true"/>
    <p:message for="name" />

    <p:outputLabel value="sobrenome" for="sobrenome"/>
    <p:inputText id="sobrenome" required="true"/>
    <p:message for="sobrenome" display="text" />

    <p:outputLabel value="idade" for="idade"/>
    <p:inputText id="idade" required="true"/>
    <p:message for="idade" display="icon" />

    <p:outputLabel value="telefone" for="telefone"/>
    <p:inputText id="telefone" required="true"/>
    <p:message for="telefone" display="tooltip" />
</h:panelGrid>
```

Mensagem padrão,  
com ícone e texto.

Mensagem somente  
com texto.

Mensagem somente  
com ícone

Mensagem na forma  
de tooltip.

A tag `<p:message>` mostra uma mensagem para um campo específico em uma grid, como no exemplo.

No exemplo acima, temos uma grid com três colunas, onde a primeira é um `outputLabel`, a segunda é um `inputText` o qual devemos atribuir um `id` e a por último tag `<p:message>`, onde devemos configurar o atributo `for` informando o `id` do campo a ser validado e o atributo `display`.

O atributo `display` serve para indicar qual será o modo de exibição da mensagem, sendo os valores válido:

- **text** – mostra uma mensagem de texto.
- **Icon** – mostra um ícone indicando um erro.
- **Tooltip** – apenas destaca o campo onde está o erro, mostrando a mensagem somente se o usuário passar o mouse por cima desse campo.

Caso não defina um `display`, o padrão é exibir uma mensagem de texto junto ao ícone de erro. Veja abaixo como são exibidas essas mensagens

## Alguns elementos do Primefaces Mensagens

 nome: Erro de validação: o valor é necessário. nome: Erro de validação: o valor é necessário.  
sobrenome: Erro de validação: o valor é necessário. sobrenome: Erro de validação: o valor é necessário.  
idade: Erro de validação: o valor é necessário. idade: Erro de validação: o valor é necessário.  
telefone: Erro de validação: o valor é necessário. telefone: Erro de validação: o valor é necessário.

nome *	<input type="text"/>	 nome: Erro de validação: o valor é necessário.
sobrenome *	<input type="text"/>	sobrenome: Erro de validação: o valor é necessário.
idade *	<input type="text"/>	
telefone *	<input type="text"/>	
<input type="button" value="Enviar"/>		

## Alguns elementos do Primefaces Ajax

```
<p:inputText id="nome" value="#{clientes.nome}">
    <p:ajax event="keyup" update="output"/>
</p:inputText>
<h:outputText id="output" value="#{clientes.nome}" />
```



Teste Ajax!

Teste Ajax!

<p:ajax> para utilizarmos ajax no Primefaces.

**event:** indica qual evento no lado do cliente deve ativar as requisições.

**update:** indica o que será atualizado.

Há também o atributo **listener** que realiza alguma ação assim que a requisição é feita.

O PrimeFaces torna o uso de ajax bastante simples, através da tag <p:ajax>. Por padrão, uma requisição ajax é disparada de acordo com o componente que ele está associado. Para componentes de entrada, o disparo é feito uma vez que o evento `onchange` aconteceram.

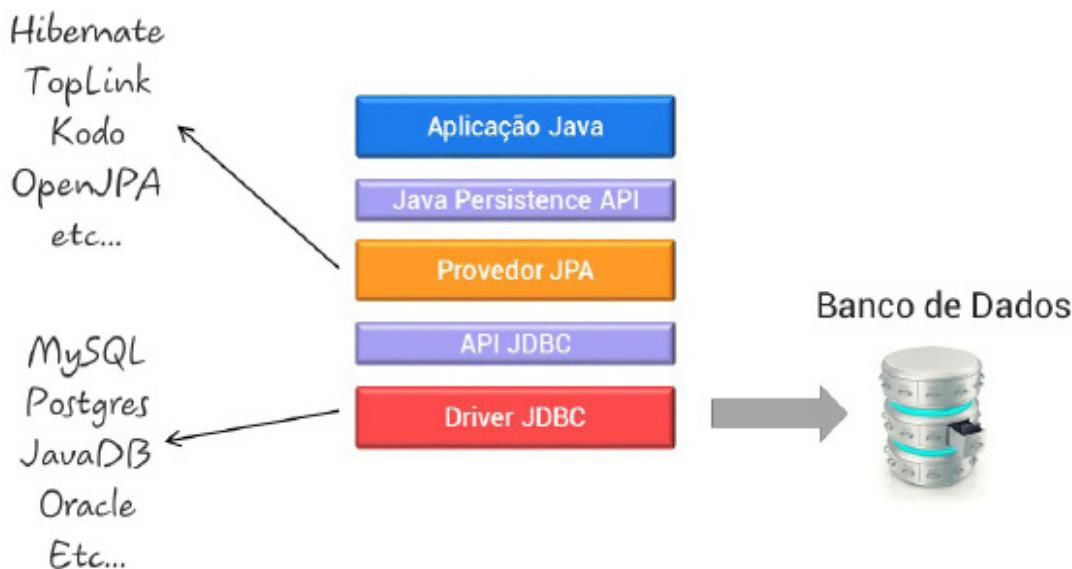
Para componentes de comando, o disparo é feito assim que o evento `onclick` acontece. É aplicável você sobrescrever o comportamento padrão, indicando um evento de sua preferência no atributo `event`.

No exemplo acima utilizamos o evento “keyup”, que dispara a requisição ao soltar a tecla. Isso faz com que a medida que o usuário for digitando, o texto vai mudando. O atributo `update` deve indicar o id do campo que será atualizado.

Há também o atributo `listener` que pode executar algum trecho de código assim que uma requisição for disparada. Veja um exemplo:

```
<h:form>
<h:panelGrid columns="3">
    <h:outputText value="Keyp: " />
    <p:inputText id="counter" value="#{bean.nome}">
        <p:ajax event="keyup" update="output"
            listener="#{bean.executaAlgumaCoisa}" />
    </p:inputText>
    <h:outputText id="output" value="#{bean.texto}" />
</h:panelGrid>
</h:form>
```

## JPA - Arquitetura



A JPA é uma 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 ideias dos frameworks de persistência principais como Hibernate e TopLink da Oracle.

O Java Persistence API (JPA) fornece um mecanismo para gerenciamento de persistência e mapeamento relacional de objeto e funções para as especificações EJB 3.0 e EJB 3.1.

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

## persistence.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
  xmlns="http://java.sun.com/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
  <persistence-unit name="Banco-ServicePU">
    <provider>org.hibernate.ejb.HibernatePersistence</provider> ← Implementação da JPA a ser utilizada
    <class>threeway.projeto.modelo.Agencia</class>
    <class>threeway.projeto.modelo.Banco</class>
    <class>threeway.projeto.modelo.Cliente</class> ← Classes persistidas
    <class>threeway.projeto.modelo.Conta</class>
    <class>threeway.projeto.modelo.Pessoa</class>
    <class>threeway.projeto.modelo.Transacao</class>
    <properties> ← Conexão com banco e configurações do Hibernate
      <property name="javax.persistence.jdbc.driver" value="org.postgresql.Driver" />
      <property name="javax.persistence.jdbc.url" value="jdbc:postgresql://localhost:5432/threeway" />
      <property name="javax.persistence.jdbc.user" value="postgres" />
      <property name="javax.persistence.jdbc.password" value="123456" />
      <property name="hibernate.hbm2ddl.auto" value="update" />
      <property name="hibernate.show_sql" value="true" />
      <property name="hibernate.format_sql" value="true" />
      <property name="hibernate.dialect" value="org.hibernate.dialect.PostgreSQLDialect" />
    </properties>
  </persistence-unit>
</persistence>

```

As unidades de persistência são definidas no arquivo persistence.xml, o qual deve estar localizado no diretório META-INF. Um arquivo persistence.xml é um descriptor de desenvolvimento exigido no JPA e pode incluir definições para uma ou mais unidades de persistência.

Uma unidade de persistência é definida dentro da tag <persistence-unit> e é necessário colocar o atributo name identificando a unidade de persistência a ser instanciando por um EntityManagerFactory.

Além do atributo name obrigatório, temos alguns subelementos opcionais:

**<class>** - especifica o nome de classes a serem persistidas. A JPA exige que todas as classes persistentes sejam registradas, as quais são referidas pela JPA como classes gerenciáveis

**<properties>** - Define propriedades padrões para especificar a url do banco de dados, usuário e senha.

**<provider>** - indica qual implementação da JPA deverá ser usada. Por exemplo, org.hibernate.ejb.HibernatePersistence é uma das implementações que podemos usar. Caso não seja especificada uma implementação, a primeira encontrada na classpath será utilizada.

Podemos ter mais de um banco de dados em uma aplicação, de modo que podemos ter mais do que uma tag <persistence-unit> dentro do persistence.xml, te permitindo usar um segundo banco de dados.

Uma Unidade de Persistência consiste de metadados declarativos que descrevem os objetos das classes de entidade para um banco de dados relacional. O EntityManagerFactory utiliza esses dados para criar um contexto de persistência que possa ser acessado por meio do EntityManager.

O EntityManager é uma interface que gerencia as operações de persistência sobre objetos e o EntityManagerFactory gerencia as instâncias do EntityManager, entraremos em detalhes mais adiante.

## Persistence.xml

Conexão com o Banco de Dados

```
<property name="javax.persistence.jdbc.driver" value="org.postgresql.Driver" />
<property name="javax.persistence.jdbc.url" value="jdbc:postgresql://localhost:5432/threeway" />
<property name="javax.persistence.jdbc.user" value="postgres" />
<property name="javax.persistence.jdbc.password" value="123456" />
```

senha

usuário

driver

url

Exibe a SQL bem

formatada no console.

```
<property name="hibernate.hbm2ddl.auto" value="update" />
<property name="hibernate.show_sql" value="true" />
<property name="hibernate.format_sql" value="true" />
<property name="hibernate.dialect" value="org.hibernate.dialect.PostgreSQLDialect" />
```

habilita a exibição  
no console do SQL  
gerado.

Propriedades do Hibernate

Valida ou exporta automaticamente  
o schema ddl para o banco de dados  
quando o SessionFactory é criado.

Values:

- validate
- update
- create
- create-drop

Dialeto que o  
hibernate irá utilizar.

Dentro da tag <properties> você pode definir várias propriedades comuns a todas as implementações de provedores JPA como javax.persistence.jdbc.\* , e outras são configurações específicas de uma implementação do provider, como hibernate.hbm2ddl.

Nesse exemplo vemos propriedades do Hibernate e a configuração da conexão do banco de dados.

## Mapeamento por Anotações

### Física

Determinam o Relacionamento entre as classes e o Banco de dados.

```
@Entity  
public class Carro {  
  
    @Id  
    private long id;  
  
    private String modelo;  
  
    private String placa;  
  
    @ManyToOne  
    @JoinColumn(name = "id_proprietario")  
    private Pessoa proprietario;
```

### Lógica

Definem a modelagem da classe com relação ao sistema.

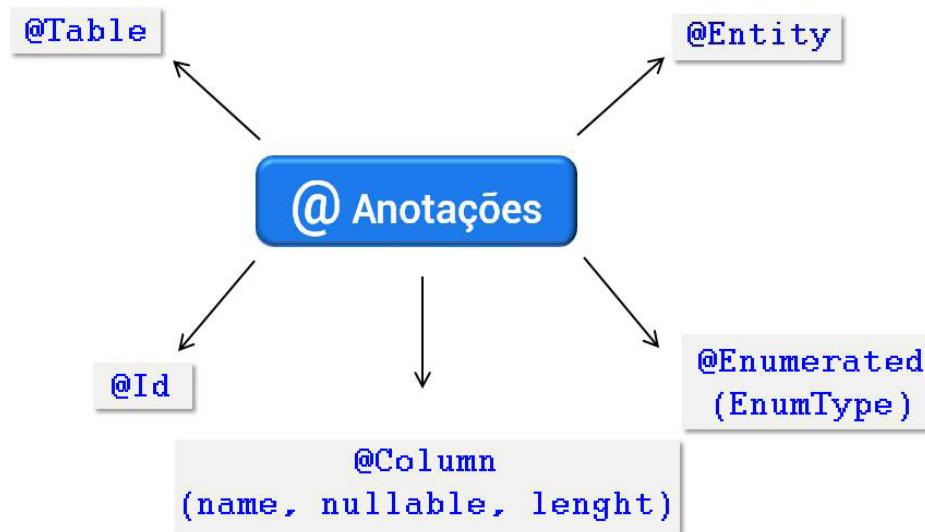
Os objetos que você vai persistir usando JPA, são normalmente denominados de objetos POJO (Plain Old Java Objects), o que significa de dizer que não há nada especial nestes objetos, são objetos que definem propriedades privadas e possuem métodos Getter/Setter públicos.

Para que este objetos possam ser mapeados e persistidos em um banco de dados relacional adicionamos algumas anotações nas classes que representam as entidades do sistema.

Podemos definir dois tipos de anotação:

- **Física** – determina o relacionamento entre as classes e o Banco de dados.
- **Lógica** – definem a modelagem da classe com relação ao sistema.

## Anotações



Anotações são uma forma especial de declaração de metadados que podem ser adicionadas ao código fonte pelo desenvolvedor. A JPA define as seguintes anotações:

- **@Entity** – torna uma classe persistentes.
- **@Table** – informa o nome da tabela.
- **@Id** – define o atributo como um identificador único (chave primária) para os objetos desta classe.
- **@GeneratedValue** – faz com que o framework de persistência gere valores automaticamente para a chave primária na tabela.
- **@Column** - não precisamos anotar atributos primitivos, mas opcionalmente podemos utilizar @Column para informar algumas configurações (nome da coluna, tamanho, se pode receber valor nulo (campo obrigatório) e se é única).
- **@Temporal** – utilizado em atributos que referem datas (Date ou Calendar).
- **@Enumerated** – utilizado para constantes (Enums).

## Anotações

```
public class Pessoa {
    private long id;
    private String nome;
    private String registroGeral;
    private Endereco endereco;
```

Indica que a coluna deve receber o nome "RG"

```
@Entity
public class Pessoa {
```

```
@Id
private long id;
```

```
private String nome;
```

```
@Column (name = "RG")
private String registroGeral;
```

```
@OneToOne
private Endereco endereco;
```

Indica que a classe é uma tabela no banco.

Indica que o campo será chave primária.

Anotação de relacionamento.

TABELA CORRESPONDENTE

Pessoa		
ID	NOME	RG

Ao anotar a classe Pessoa com **@Entity** informamos que objetos dessa classe são objetos de persistentes, podem ser salvos no banco de dados.

Todo objeto persistente tem obrigatoriamente que possuir um campo anotado com **@Id** que define o atributo como um identificador único (chave primária) para os objetos desta classe.

Os campos anotados com **@Column** te permitem associar ou diferenciar o nome do campo na classe ao nome real da coluna no banco de dados.

Não precisamos anotar atributos primitivos, mas opcionalmente podemos utilizar **@Column** para informar algumas configurações (nome da coluna, tamanho, se é campo obrigatório ou se é único).

Quando o campo é um outro objeto, como o campo endereco, precisamos informar qual é o tipo de relacionamento entre os objetos, para isso usamos **@OneToOne** dizendo que há um objeto do tipo Endereço para cada objeto do Pessoa.

## Geração Automática de Chaves Primárias IDENTITY

```
@Entity  
public class Pessoa {  
  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private long id;
```

O próprio banco de dados decide qual será o próximo ID.

**SQLite, MySQL (AUTO\_INCREMENT), SQL Server**

Toda entidade precisa de um identificador único, como já foi dito, mas a maneira como geramos esse identificador pode variar de acordo com o banco de dados alvo.

Através da tag @GeneratedValue podemos gerar valores automaticamente para as chaves primárias, mas precisamos definir a estratégia para essa geração automática de chaves primárias.

A primeira estratégia é a IDENTITY, onde o próprio banco de dados define qual será o próximo ID gerado, também conhecido como autoincrement.

`@GeneratedValue(strategy = GenerationType.IDENTITY)`

## Geração Automática de Chaves Primárias SEQUENCE

```

@Entity
@SequenceGenerator(name="seq_gen", initialValue=1, allocationSize=100)
public class Pessoa {

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator="seq_gen")
    private long id;
}

```

Indica que haverá uma sequência no banco de dados

Oracle, SQL Server, PostgreSQL  
Padrão do PostgreSQL e Oracle

Indica que uma sequência salva no banco de dados será usada.

Mantém salvo o próximo valor a ser usado.

A estratégia SEQUENCE utiliza um tipo sequência do bando de dados que irá retornar o próximo valor a ser utilizado. Para criar a sequência, devemos utilizar a anotação @SequenceGenerator e configurar a sequência. Devemos definir os seguintes atributos:

- **name (obrigatório)** – nome da sequência
- **sequenceName (opcional)** – nome da sequência no banco de dados
- **initialValue (opcional)** – valor inicial da sequência
- **allocationSize (opcional)** – tamanho da sequência

@SequenceGenerator(name, sequenceName, initialValue, allocationSize)

Para definirmos a sequência a ser utilizada, devemos indicá-la na anotação @GeneratedValue:

@GeneratedValue(strategy = GenerationType.SEQUENCE, generator="nome\_sequencia")

## Geração Automática de Chaves Primárias TABLE

```
@Entity  
@TableGenerator(name="tab_gen", initialValue=0, allocationSize=100)  
public class Pessoa {  
  
    @Id  
    @GeneratedValue(strategy = GenerationType.TABLE, generator="tab_gen")  
    private long id;
```

Utiliza uma tabela para armazenar o id atual de cada entidade. Mantém salvo o último valor que foi usado.

### Única estratégia suportada em qualquer BD Portabilidade

A estratégia TABLE utiliza uma tabela para armazenar o id atual de cada entidade armazenada. A tabela utilizada consiste de duas colunas, uma com o nome da tabela e outra com o último valor utilizado como ID. Para criar a tabela, devemos utilizar a anotação `@TableGenerator(name, table, pkColumnName, pkColumnValue, valueColumnNames)`. Podemos definir os seguintes atributos:

- **name (obrigatório)** – nome do gerador.
- **table (opcional)** – nome da tabela que irá armazenar os valores gerados.
- **initialValue (opcional)** – valor inicial da sequência.
- **allocationSize (opcional)** – quantidade de incrementos disponíveis quando utilizando o gerador.
- **pkColumnName (opcional)** – nome da chave primária na tabela.
- **pkColumnValue (opcional)** – valor da chave primária na tabela geradora que distingue esse conjunto de valores gerados de outros que podem estar armazenados na tabela.
- **valueColumnName (opcional)** – nome da coluna que guarda o último valor gerado.

Após você vai definir o gerador a ser utilizado, devendo indicá-lo na anotação `@GeneratedValue`:

```
@GeneratedValue(strategy = GenerationType.TABLE, generator="nome_sequencia")
```

A estratégia TABLE é única estratégia que pode ser utilizada em qualquer banco de dados.

## Geração Automática de Chaves Primárias AUTO

```
@Entity  
public class Pessoa {  
  
    @Id  
    @GeneratedValue(strategy = GenerationType.AUTO)  
    private long id;
```

### IDENTITY

- MySQL, SQLite e MsSQL

Indica que a JPA deve selecionar uma estratégia de geração apropriada de acordo com o banco.

### SEQUENCE

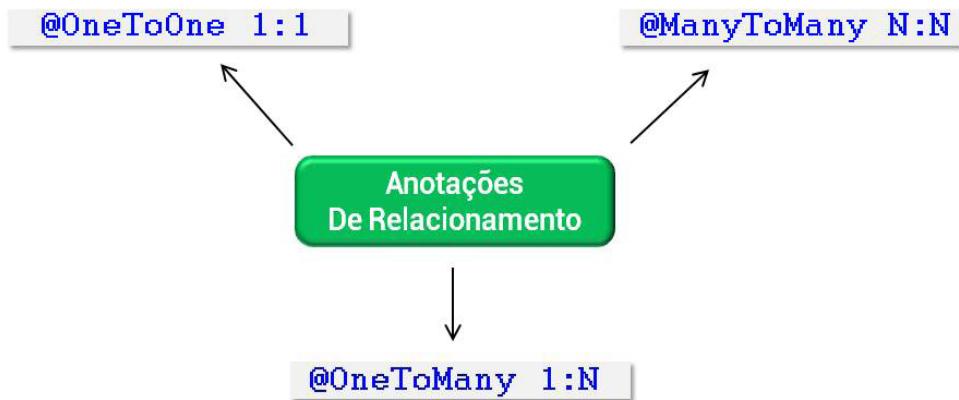
- Oracle e PostgreSQL

Outra estratégia é a AUTO, onde a própria JPA deve selecionar uma estratégia de geração apropriada de acordo com o banco de dados. Esse é o padrão da anotação `@GeneratedValue`.

`@GeneratedValue(strategy = GenerationType.AUTO)`

Nos banco MySQL, SQLite e MsSQL a estratégia selecionada é a IDENTITY. Nos bancos Oracle e PostgreSQL a estratégia SEQUENCE é selecionada.

## Anotações de Relacionamento



Além de mapear entidades a serem persistidas no banco de dados, devemos também mapear as relações em que os objetos estão envolvidos.

Para fazer o mapeamento de relacionamento, a JPA oferece as anotações: `@OneToOne`, `@ManyToOne`, `@OneToMany` e `@ManyToMany`.

## One To One Unidirecional

```

@Entity
public class Pessoa {
    @Id
    private long id;
    private String nome;
    @Column (name = "RG")
    private String registroGeral;
    @OneToOne
    private Endereco endereco;
}

@Entity
public class Endereco {
    @Id
    private long id;
    private String rua;
    private String cep;
}

```

TABELA CORRESPONDENTE



O relacionamento @OneToOne (um para um) pode ser entendido como Pessoa tem um Endereço. Para acomodar essa relação temos a tabela Endereco e a tabela Pessoa (uma Pessoa tem um Endereço) que recebe o ID da tabela Endereco como chave estrangeira.

O relacionamento acima é considerado unidirecional, ou seja, somente um dos objetos sabe da existência do outro.

Note que somente a entidade Pessoa possui referência à Endereço, assim Pessoa é o lado dominante da relação ou seja, a tabela que mapeia a classe Pessoa possui a chave estrangeira. Toda relação deve ter um lado dominante.

É possível definir o nome da chave estrangeira com `@JoinColumn(name="id_endereco")`.

## One To One Bidirecional

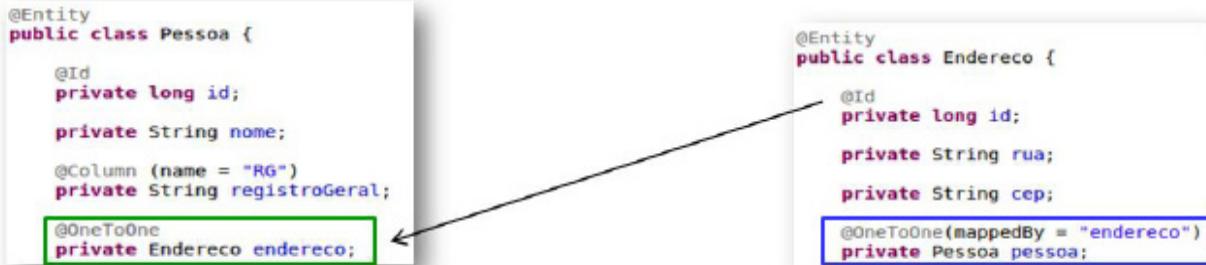


TABELA CORRESPONDENTE

Pessoa			
ID	NOME	RG	ID_ENDERECO

Endereco		
ID	RUA	CEP

Relacionamentos também podem ser bidirecionais, ou seja, ambos os lados tem referência de um para o outro.

Para o correto mapeamento devemos estabelecer o lado dominante do relacionamento usando atributo mappedBy da anotação @OneToOne no lado não dominante. O atributo mappedBy deve ter o nome do campo que representa a entidade “dominada” na entidade “dominante”. Neste caso temos o campo “endereco” em Pessoa.

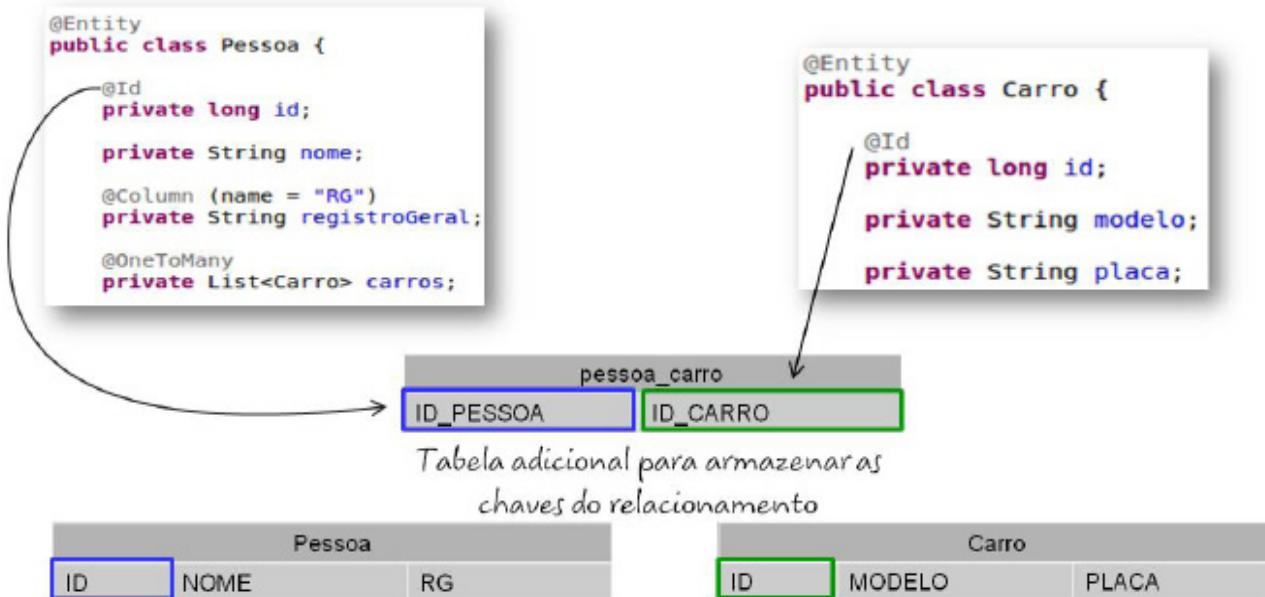
Relacionamentos bidirecionais não são automáticos, sou seja você precisará fazer a associação na sua programação Java, veja:

```

pessoa.setEndereco(endereco) .
endereco.setPessoa(pessoa);

```

## One To Many Unidirecional

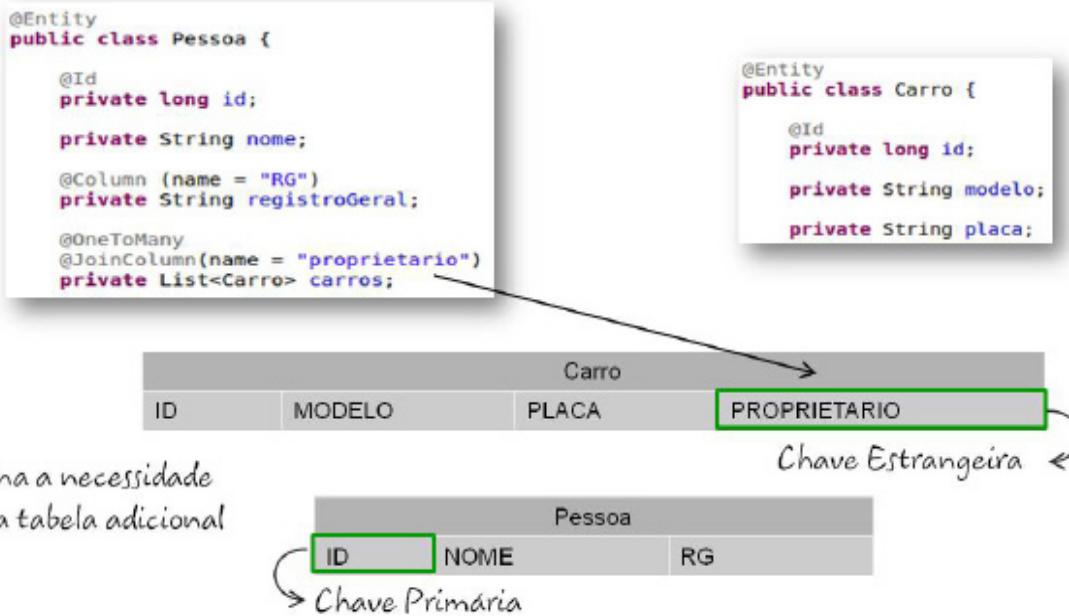


Uma Pessoa pode possuir mais de um Carro, em Java representamos essa associação como uma Collection, no caso o campo `List<Carro> carros`.

Para a JPA devemos marcar esse relacionamento como sendo `@OneToMany` (um para muitos). Note que do ponto de vista de pessoa, a relação é 1:N, enquanto do ponto de vista do carro, a relação é N:1.

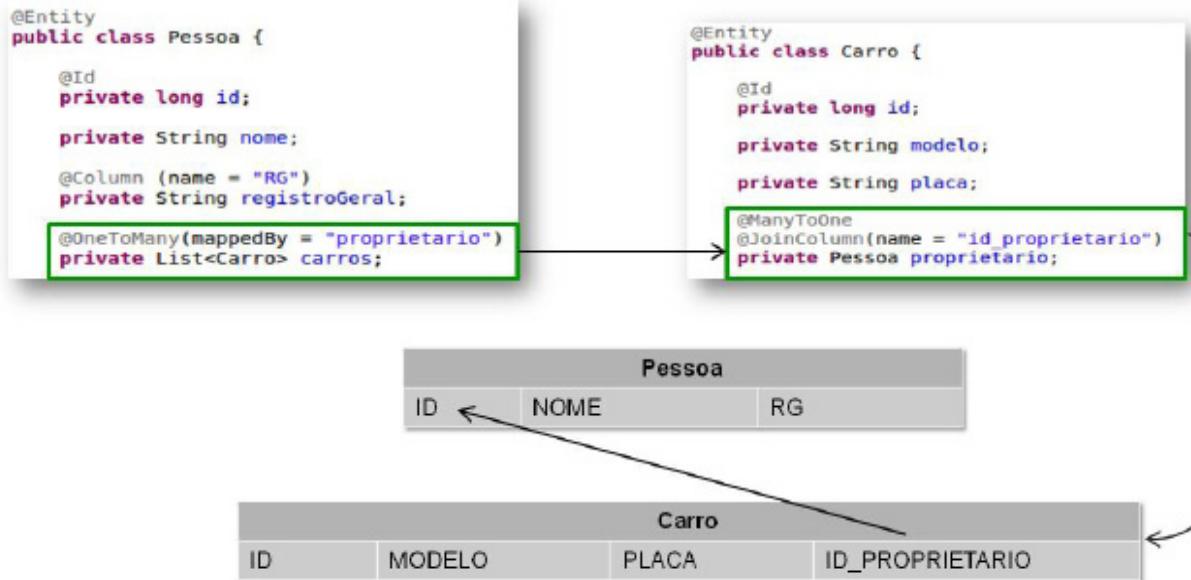
No caso do relacionamento unidirecional a JPA cria uma tabela adicional para armazenar as chaves do relacionamento. Para evitar a criação dessa tabela adicional, podemos utilizar a anotação `@JoinColumn`.

## One To Many ndo Join Column



Usando a anotação `JoinColumn`, podemos especificar uma coluna adicional na tabela carro que irá referenciar seu proprietário (Pessoa). A chave fica na tabela carro pelo fato de um carro possuir apenas um proprietário.

## One To Many Bidirecional



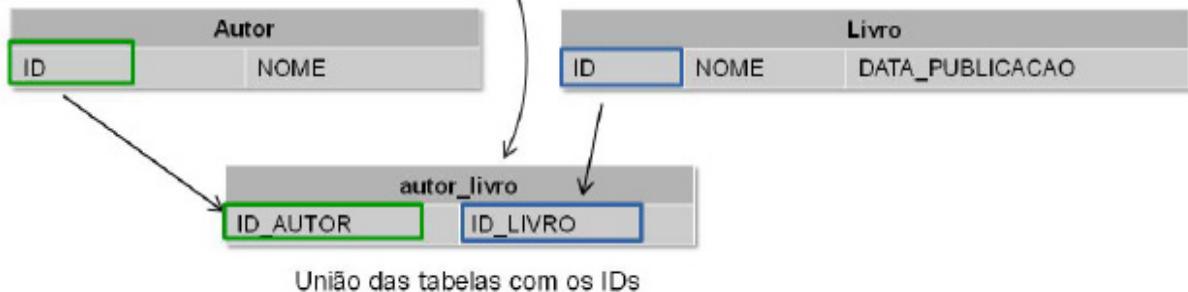
Para criar um relacionamento bidirecional entre essas duas entidades, devemos alterar a entidade Carro com anotação `@ManyToOne` e `@JoinColumn` e a entidade Pessoa com a anotação `@OneToMany` e a propriedade `mappedBy`, estabelecendo Carro como lado dominante.

Utilizando a anotação `@JoinColumn` você pode criar uma coluna com a chave que referencia o ID do proprietário.

## Many To Many

```
@Entity
public class Autor {
    @Id
    private long id;
    private String nome;
    @ManyToMany
    @JoinTable(name="autor_livro")
    private List<Livro> livros;
```

```
@Entity
public class Livro {
    @Id
    private long id;
    private String nome;
    @Column(name="DATA_PUBLICACAO")
    @Temporal(TemporalType.DATE)
    private Date ano;
    @ManyToMany(mappedBy="livros")
    private List<Autor> autores;
```



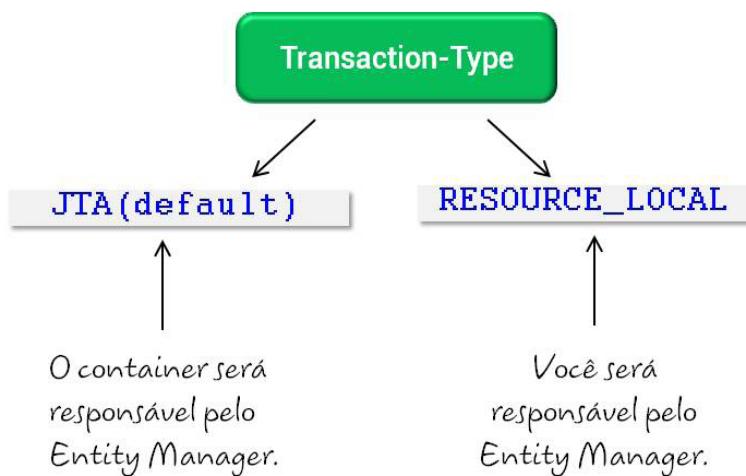
Um livro possui muitos autores e um autor possui muitos livros. Para informar a cardinalidade do relacionamento entre livros e autores, você deve usar a anotação `@ManyToMany` na coleção.

No relacionamento `@ManyToMany` é necessário uma tabela que faça a união dos IDs das entidades relacionadas. Não há chave estrangeira nesse tipo de relação, já que temos Collections de ambos os lados.

Adicionamos a propriedade `mappedBy`, em `Livro`, para informar qual é o lado NÃO dominante. Caso você não defina o lado dominante, a JPA criará duas tabelas de relacionamento: `autor_livro` e `livro_autor`.

Anotação `@JoinTable` te permite definir a tabela de união.

## Tipos de Transação



Na configuração da unidade de persistência é preciso declarar o atributo transaction-type

```
<persistence-unit name="JPA" transaction-type="[JTA ou RESOURCE_LOCAL]">
```

O valor do atributo é usado para indicar à JPA se a transação será local (RESOURCE\_LOCAL) ou controlada pelo servidor (JTA).

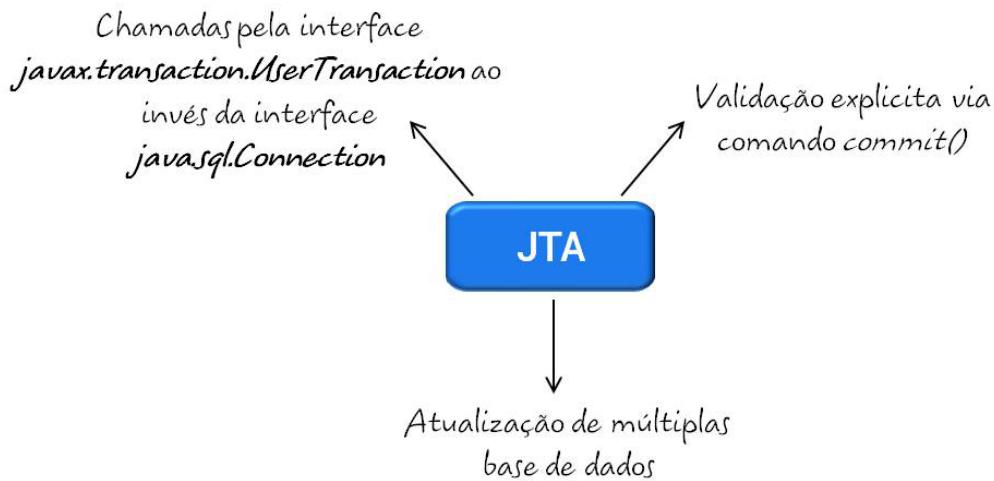
Se você usar RESOURCE\_LOCAL, você ficará responsável pela criação e rastreamento do EntityManager:

- você deve usar EntityManagerFactory para obter um EntityManager
- o EntityManagerFactory só pode ser injetado por @PersistenceUnit
- você deve usar EntityTransaction (begin/commit) entorno de cada chamada ao seu EntityManager
- se você invocar entityManagerFactory.createEntityManager() mais de uma vez, você terá mais de um EntityManager

```

EntityManagerFactory emf = Persistence.createEntityManagerFactory("MeuPU");
EntityManager em = emf.getEntityManager();
em.getTransaction().begin();
Estudante estudante = new Estudante();
//...
em.persist(estudante);
em.getTransaction().commit();
em.close();
emf.close();
  
```

## JTA



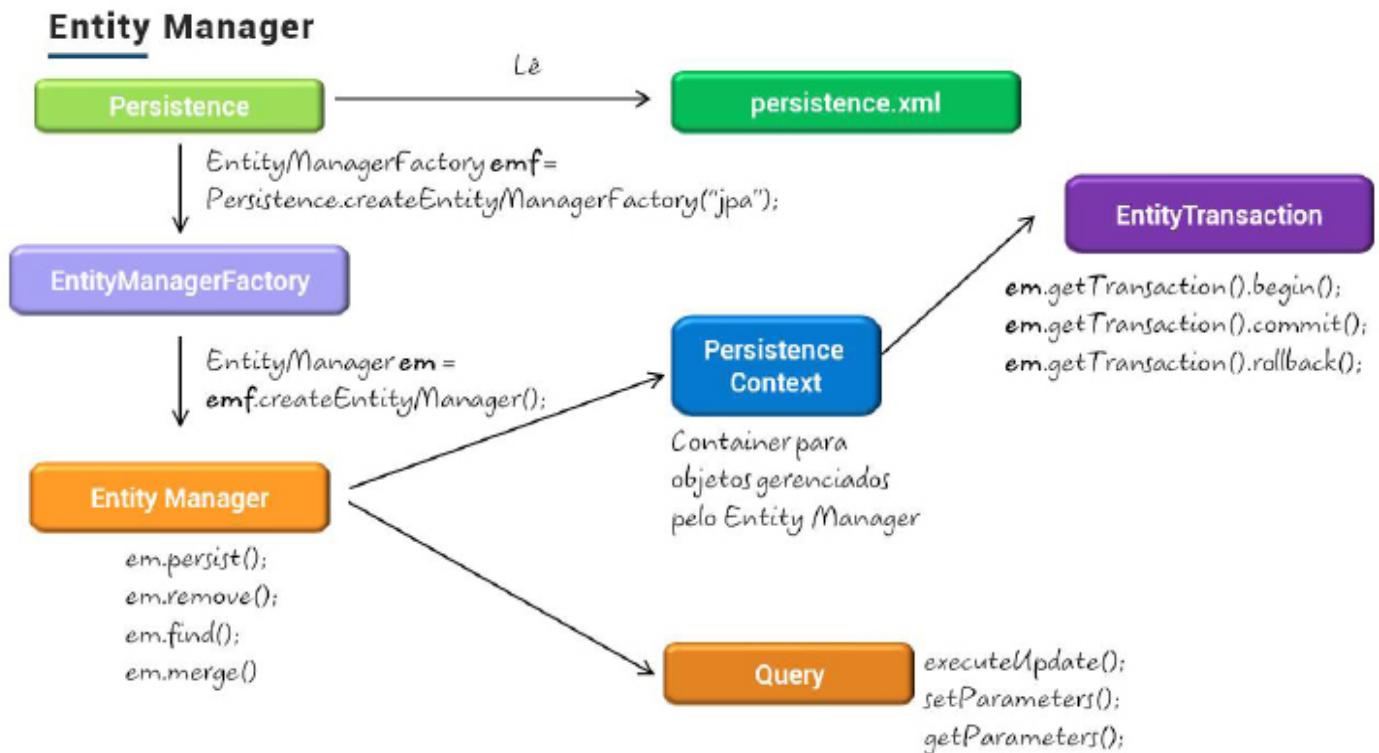
Com a JTA o inicio (begin) e fim da transação (commit/rollback) são invocados através da interface `javax.transaction.UserTransaction` ao invés da interface `java.sql.Connection`.

- você não usa EntityManagerFactory para obter um EntityManager
- o EntityManager deve ser injetado por @PersistenceContext
- as transações são controladas (begin/commit) são pelo container.

`@Stateless`

```
public class EstudanteBean {
    @PersistenceContext
    EntityManager em;
    public void salvarEstudante(Estudante estudante) {
        em.persist(estudante);
    }
}
```

com a JTA a transação inicia-se com a invocação do método `salvarEstudante()` e faz commit após o a invocação ser completada. Se houver falha e uma exceção for lançada no método a transação automaticamente fará um rollback.



O EntityManager é responsável por gerenciar o estado dos objetos e sincronizar os dados destes objetos com as respectivas tabelas no banco de dados.

Você pode obter um EntityManager através de um EntityManagerFactory:

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("MeuPersistenceUnit");
```

```
EntityManager em = emf.createEntityManager();
```

Ou por injeção :

```
@PersistenceContext(unitName="MeuPersistenceUnit")
```

```
EntityManager em
```

Em ambos a JPA procura pelo arquivo persistence.xml, e fará a leitura da entidades mapeadas, validação de conexão com o banco de dados e outras ações que estejam configuradas como: validação de schema, criação ou atualização de estrutura de tabelas, criar objetos de sequencia, etc.

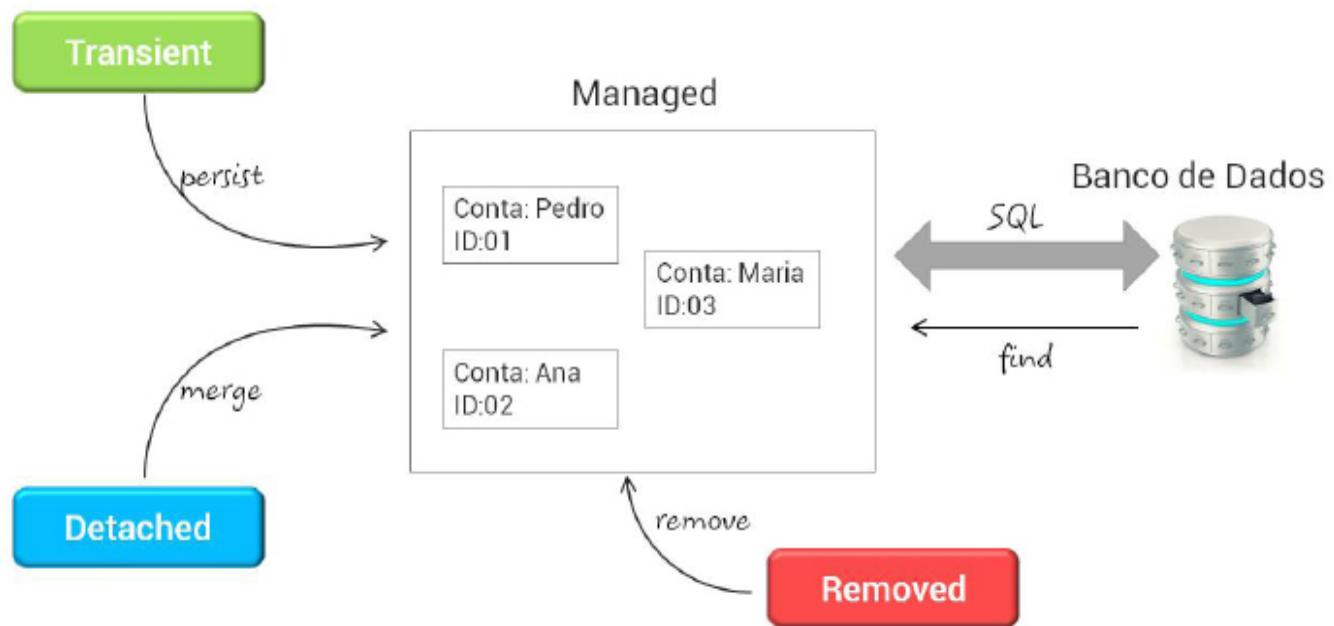
A execução dos métodos `find()`, `persist()`, `merge()`, `getReference()`, ou `querires` (`entityManager.createQuery()`) de `EntityManager` coloca ou tira ( `remove()`, `delete query` ) as entidades no Persistence Context (um cache de entidades). Se um objeto um objeto está no Persistence Context, dizemos que esse objeto é gerenciado, **Managed**, ou seja está sob controle do `EntityManager`.

Operações que alteram o estado do objeto como persist(), remove(), merge() devem ser executadas dentro de uma transação. O EntityManager possui o método getTransaction() que retorna uma instância de EntityTransaction, te permitindo iniciar uma transação e finalizar uma transação.

```
EntityTransaction et = em.getTransaction();  
et.begin();  
em.persist(estudante);  
et.commit();  
em.close();
```

Como já sabemos se em sua unidade persistência o atributo `transaction-type="RESOURCE_LOCAL"` o controle da transação fica sob sua responsabilidade devendo ser feito explicitamente.

## Estados de uma Entidade



É importante saber o ciclo de vida de uma entidade e o estado que pode assumir.

### Estado Transient

Quer dizer que o objeto acabou de ser criado e a JPA ainda não sabe da sua existência. A entidade não existe no banco de dados.

### Estado Managed

Quer dizer que a JPA está gerenciando a existência do objeto-entidade. A entidade existe no banco de dados.

### Estado Detached

Quer dizer que a JPA não está gerenciando o objeto-entidade, então alterações não serão registradas. A entidade pode ou não existir no banco de dados.

### Estado Removed

Quer dizer que o objeto-entidade foi removido do banco de dados.

## EntityManager Criar

Criação do Entity Manager

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("jpa");
EntityManager em = emf.createEntityManager();

Pessoa pessoa = new Pessoa();
pessoa.setNome("Joao"); ←
pessoa.setCPF("123456");

em.getTransaction().begin();
em.persist(pessoa);
em.getTransaction().commit();
```

Instanciação de um novo objeto.  
Neste momento, o objeto está no estado *Transient*.

Persiste o novo objeto no banco de dados e ele passa para o estado de *Managed* (gerenciado).

Ao criar um objeto, a JPA ainda não o reconhece pelo fato dele nunca ter passado pelo EntityManager. Essa entidade também não existe no banco de dados, então quer dizer que ela está em um estado transient.

Um objeto Pessoa recém instânciado com todas as suas propriedades preenchidas está no estado transient.

Após criar esse objeto, com o EntityManager, retornamos o objeto EntityTransaction através do método getTransaction() e iniciamos a transação com o método begin().

Iniciada a transação, podemos persistir o objeto através do método persist(entidade) tornando o estado do objeto managed, garantindo que esse objeto terá o valor de seus atributos salvos no banco de dados. Depois de persistir o objeto, devemos salvar a transação no banco de dados através do método commit()

## EntityManager Recuperar

```
Pessoa pessoa = em.find(Pessoa.class, 1);
```

Tipo da entidade  
a ser buscada

Busca feita pelo valor  
da Primary Key

Para recuperar uma entidade através do EntityManager devemos utilizar o comando `find(Class<T>, Object primaryKey)`. Os parâmetros desse método são a classe da entidade a ser buscada e seu ID. Poderíamos também usar uma Query

```
String consulta = "SELECT p FROM Pessoa p WHERE p.id= :id";
TypedQuery<Pessoa> query = entityManager.createQuery(consulta, Pessoa.class);
query.setParameter("id", id);
List<Pessoa> resultado = query.getSingleResult();
```

## EntityManager Update

Objeto deixa o estado de Gerenciado e passa para o estado Detached (destacado)

```
Pessoa pessoa = em.find(Pessoa.class, 1);
em.getTransaction().begin();
em.detach(pessoa);

pessoa.setNome("Maria Tereza");

em.merge(pessoa);
em.getTransaction().commit;
```

Busca o objeto, deixando-a em estado Managed

Alteração do nome

Objeto volta para o estado de Gerenciado e as alterações podem ser salvas no banco

Utilizamos o método detach() do EntityManager para demonstrar como um objeto pode passar para o estado detached. O método detach() remove a entidade do persistence context. Para retornar esse objeto para o persistence context, devemos utilizar o método merge().

Isso pode ser útil caso um objeto possa sofrer várias alterações seguidas, evitando que a cada alteração nesse objeto seja replicada no banco de dados. Podemos deixar esse objeto de fora do persistence context até que a alteração final seja feita, só assim voltando para o estado managed.

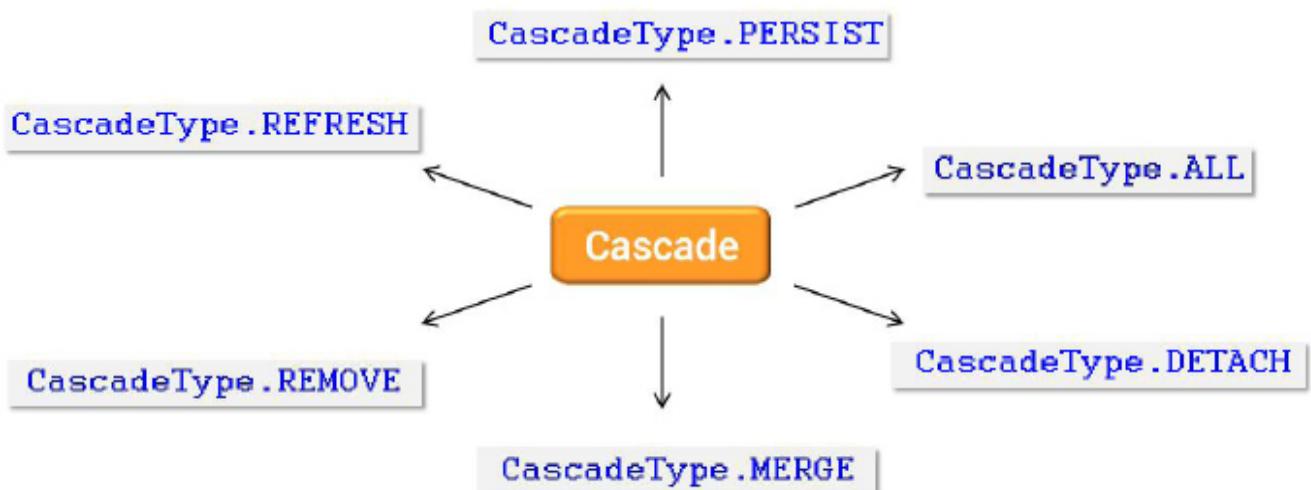
## EntityManager Deletar

```
Busca o objeto  
↓  
Pessoa pessoa = em.find(Pessoa.class, 1);  
em.getTransaction().begin();  
em.remove(pessoa);  
em.getTransaction().commit();
```

Remove a entidade. Entidade  
agora está no estado *Removed*

Para remover uma entidade, devemos primeiro recuperar a entidade a ser removida e utilizar o método remove() do EntityManager. O método remove() remove a instância dessa entidade. Isso torna o estado da entidade como removed.

## Cascade



As definições de Cascade são associados às anotações `@OneToOne`, `@OneToMany` e `@ManyToMany`. Indicam com que ação em cascata o relacionamento será tratado, ou seja, especifica quais operações deverão ser em cascata do objeto dono da relação para os objetos associados. Cascade pode assumir os seguintes valores:

- **`CascadeType.PERSIST`** – executado toda vez que uma nova entity é inserida no banco de dados por `entityManager.persist()`.
- **`CascadeType.REMOVE`** – executado quando uma entity é removida do banco de dados, os relacionamentos marcados também serão eliminados por `entityManager.remove()`.
- **`CascadeType.DETACH`** – executado toda vez que uma entity é removida do Persistence Context. Comandos que podem disparar essa ação: `entityManager.detach()` , `entityManager.clear()` . Ocorrerá um detach também quando o Persistence Context deixar de existir;
- **`CascadeType.REFRESH`** – executado quando uma entity for atualizada com informações no banco de dados, `entityManager.refresh()`;
- **`CascadeType.MERGE`** – executado toda vez que uma alteração é executada em uma entity. Essa alteração pode acontecer ao final de uma transação com a qual uma managed Entity foi alterada, ou pelo comando `entityManager.merge()` ;
- **`CascadeType.ALL`** – junção de todos os tipos de cascade.

## Cascade

```
@Entity
public class Pessoa {
    @Id
    private long id;
    private String nome;
    @Column (name = "RG")
    private String registroGeral;
    @OneToOne
    private Endereco endereco;
```

```
@Entity
public class Pessoa {
    @Id
    private long id;
    private String nome;
    @Column (name = "RG")
    private String registroGeral;
    @OneToOne(cascade = CascadeType.PERSIST)
    private Endereco endereco;
```

*CascadeType deve ser passado dentro na anotação de relacionamento*

*CascadeType.Persist faz com que o endereço também seja persistido*

```
Pessoa pessoa = new Pessoa();
pessoa.setCPF("123456");
pessoa.setNome("Maria");

Endereco endereco = new Endereco();
endereco.setRua("Rua 2015");
endereco.setCep("12345678");

pessoa.setEndereco(endereco);

em.getTransaction().begin();
em.persist(pessoa);
em.getTransaction().commit();
```

O exemplo acima mostra como funciona o CascadeType.PERSIST. Na anotação de relacionamento devemos adicionar o atributo cascade e o tipo de cascade a ser aplicado.

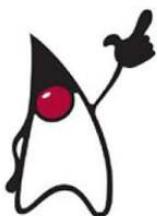
Repare que temos duas entidade, Pessoa e Endereço, onde Endereço depende de Pessoa. Ao utilizar o CascadeType.PERSIST não precisamos utilizar o método persist no Endereço, ao persistimos Pessoa, Endereço automaticamente também será persistido. A mesma idéia vale para os outros tipos de cascade.

## OneToOne, OneToMany e ManyToMany

\*Orphan Removal\*

(`orphanRemoval = true`)

A entidade do lado inverso do lado dominante é deletada do banco de dados quando não há relacionamento com outra entidade dominante



Quando temos um objeto que só pode existir na presença de outros estamos utilizando uma relação de composição.

Um registro que só deve existir em função do registro pai e não é removido do banco de dados junto com a remoção do registro pai é um registro órfão. Para evitar problemas de registros órfãos usamos a opção `orphanRemoval`.

O atributo `orphanRemoval` só pode ser aplicado em relacionamento do tipo `@OneToOne` ou `@OneToMany` e assume os valores `true` e `false`.

Funciona da seguinte forma: a entidade do lado inverso do lado dominante é deletada do cando de dados quando não há relacionamento com outra entidade, ou seja, levando em consideração o exemplo anterior, se a entidade `Pessoa` for removida e por algum motivo o `Endereço` ainda estiver armazenado, esse `Endereço` será removido pelo fato de não estar relacionado a uma `Pessoa`.

## FetchType

### FetchType.LAZY

- As informações do atributo marcado não serão carregadas.

```
@Entity
public class Pessoa {
    @Id
    private long id;
    private String nome;
    @Column (name = "RG")
    private String registroGeral;
    @OneToMany(mappedBy = "proprietario", fetch = FetchType.LAZY)
    private List<Carro> carros;
```

### FetchType.EAGER

- As informações do atributo marcado serão carregadas.

```
@Entity
public class Pessoa {
    @Id
    private long id;
    private String nome;
    @Column (name = "RG")
    private String registroGeral;
    @OneToOne(cascade = CascadeType.PERSIST, fetch = FetchType.EAGER)
    private Endereco endereco;
```

Por padrão a carga dos dados de relacionamentos são sempre Lazy (preguiçoso), ou seja após

```
Pessoa pessoa = entityManager.find(Pessoa.class, 30);
```

a propriedade carros só será carregado quando referenciarmos o relacionamento,

```
pessoa.carros.get(1);
```

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 carregado da base de dados, o atributo mapeado com fetch=FetchType.EAGER fará com que conteúdo do relacionamento também seja carregado. No exemplo acima temos um atributo do tipo Endereço, sendo Endereço uma entidade no banco de dados, então quando você fizer a busca de uma pessoa, o endereço dela também será trazido.

**FetchType.LAZY** – sempre que o objeto “pai” for carregado da base de dados, o atributo mapeado com fetch=FetchType.LAZY fará com que o seu conteúdo somente seja carregado quando acessado pela primeira vez. No exemplo acima temos uma lista de carros em Pessoa, então ao fazer a busca de uma pessoa, a lista de carros pertencente a essa Pessoa não será carregada, a não ser que você tente acessar algum objeto dessa lista.

Os relacionamentos do tipo OneToMany e ManyToMany têm por padrão o tipo de recuperação Lazy, enquanto os relacionamentos OneToOne e ManyToOne tem o tipo de recuperação Eager.

## JPQL Java Persistence Query Language



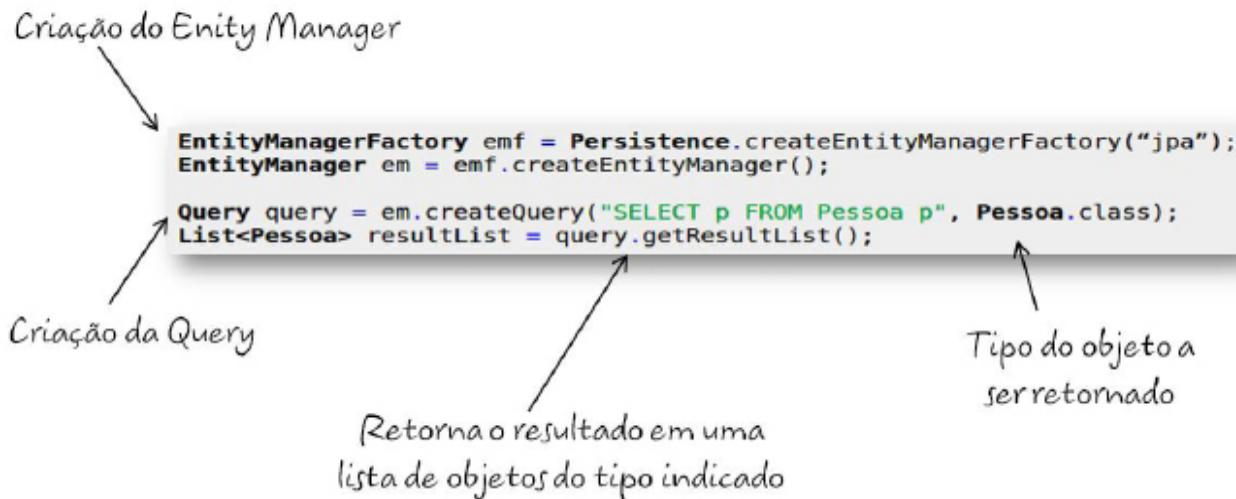
Java Persistence Query Language (JPQL) é uma plataforma independente de linguagem de consulta orientada a objeto, foi definida como parte da especificação Java Persistence API. JPQL é usado para fazer consultas com entidades armazenadas em um banco de dados relacional. É fortemente inspirada pelo SQL e suas consultas assemelham às 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.

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 são classes e objetos. Uma consulta JPQL recupera e retornar objetos de entidade, em vez de apenas os valores de campo a partir de tabelas de banco de dados, como acontece com a SQL.

A JPQL não oferece um método INSERT como na SQL, portanto devemos utilizar o método persist() do EntityManager para persistir um objeto recém criado.

## JPQL Recuperar



Utilizamos o método `createQuery` do `EntityManager` criando uma instância do objeto `Query` para executar um statement da JPQL. No método `createQuery` os parâmetros são uma String com a query de busca e um segundo argumento que deve ser a classe dos objetos que vão ser recuperados na query criada. Para pegar o resultado da consulta, utilizamos o método `getResultSet()` que irá retornar uma lista de objetos do tipo `Pessoa`.

## JPQL Update

Criação do Entity Manager

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("jpa");
EntityManager em = emf.createEntityManager();

Query query = em.createQuery(
    "UPDATE Pessoa SET nome = :Pedro" +
    "WHERE nome = :nome");

query.setParameter("nome", "Joao");
query.executeUpdate();
```

Criação da Query

Parâmetro nome



Para realizar alterações em entidades também devemos criar uma instância do objeto Query, porém agora vamos criar uma query parametrizada e substituir os valores depois.

Para utilizarmos parâmetros devemos utilizar dois pontos ( “:” ) e o nome do parâmetro. No exemplo temos o parâmetro :nome. Damos um valor a esse parâmetro através do método setParameter(identificador, valor) passando o nome do parâmetro e o seu valor. Feito isso, basta executar o comando query.executeUpdate(), que executa um statement de update ou delete.

## JPQL Deletar

Criação do Entity Manager

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("jpa");
EntityManager em = emf.createEntityManager();

Query query = em.createQuery("DELETE FROM Pessoa p WHERE p.id = :id ");
query.setParameter("id", "1");
query.executeUpdate();
```

Criação da Query

Parâmetro *id*

Remover entidades segue a mesma idéia de realizar alterações, mas agora devemos criar uma query de DELETE.

## Named Queries

Anotação @NamedQuery na entidade

```
@Entity
@NamedQuery (name = "Cliente.BUSCA_POR_CPF",
              query = "SELECT Cliente WHERE cpf = :cpf")
public class Cliente {
    public String final BUSCA_POR_CPF = "Cliente.BUSCA_POR_CPF";
    ...
}
```

Nome da query

Query parametrizada

\*Boa prática\*  
Constante com o nome da query

```
TypedQuery<Cliente> query = em.createNamedQuery(Cliente.BUSCA_POR_CPF, Cliente.class);
query.setParameter("cpf", "123456");
```

Nome da query

Tipo do objeto a ser retornado

Named queries são consultas JPQL definidas estaticamente. São definidas através de anotações nas classes que implementam as entidades correspondentes, sendo elas:

- **@NamedQuery** – define apenas uma consulta.

- **@NamedQueries** – define várias consultas. Veja que temos dois atributos, nome e query. O atributo nome é um identificador para a consulta que estamos criando (lembrando que esse nome deve ser único) e query é a consulta em si. Uma boa prática é criar uma constante na classe com o nome da consulta, isso ajuda na hora de buscar pelo nome da query e evita de escrevermos o nome errado.

Para utilizar uma Named Query, devemos utilizar a interface TypedQuery<X> que é responsável por controlar a execução de consultas tipadas e o método createNamedQuery do EntityManager. No método createNamedQuery recebe como parâmetro o nome da consulta (por isso criamos a constante com o nome) e a classe da entidade retornada na consulta. Utilize o método getSingleResult() ou getResultList() para executar a query.

## Native Query

Criação do entity manager

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("jpa");
EntityManager em = emf.createEntityManager();

String consulta = "SELECT * FROM Pessoa WHERE id = :id";
Query query = em.createNativeQuery(consulta, Pessoa.class);
query.setParameter("id", 1);

Pessoa pessoa = query.getSingleResult();
```

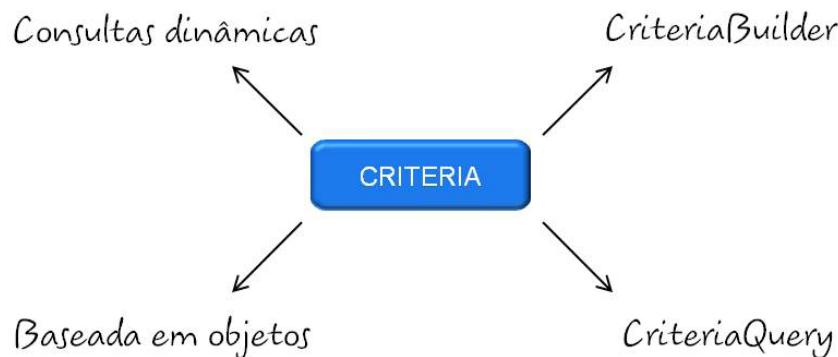
Criação da Query  
Nativa

A query deve estar na linguagem  
nativa do banco que está sendo  
utilizado

Queries nativas são consultas criadas utilizando a linguagem SQL do banco de dados que está sendo utilizado. Devemos utilizar esse tipo de consulta quando desejamos buscar algo muito específico (como consultas que possam retornar mais de uma entidade), já que a JPA não oferece todos os recursos e a otimização de um banco de dados.

A diferença das buscas que vimos anteriormente é que devemos utilizar o método `createNativeQuery` passando a consulta SQL e a classe da entidade para o qual será mapeado os dados retornados.

## CRITERIA



A partir especificação da JPA 2.0 define-se Criteria API para consultas dinâmicas através da construção de uma instância da classe javax.persistence.CriteriaQuery, em vez de uma abordagem baseada em strings usado em JPQL, motamos as consulta de forma programática usando objetos da API para definir a consulta.

A sintaxe é projetada para construir uma árvore de comando que representada por elementos tais como projeções, predicados condicionais de cláusula WHERE ou elementos GROUP BY. Veja exemplo, consulta JPQL:

```

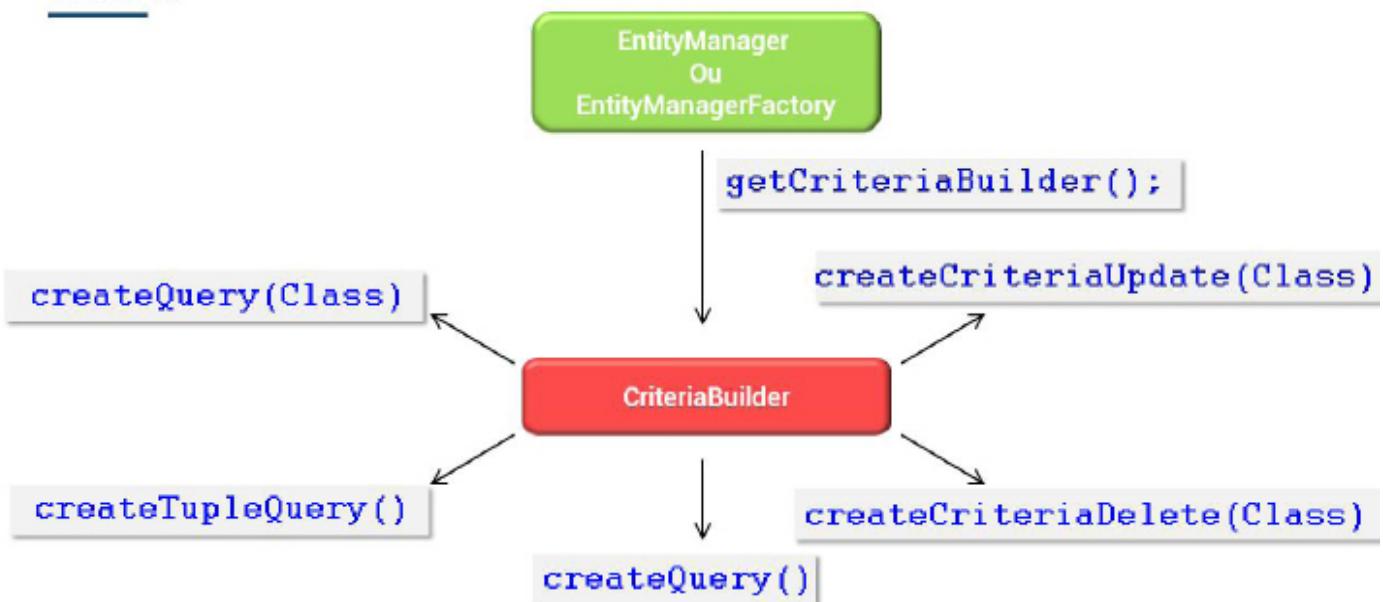
String consulta = "select c from Pessoa c where 1=1 " +
if(nome != null) consulta += " and c.nome = :nome";
if(idade != null) consulta += " and c.idade = :idade" ;
TypedQuery<Pessoa> query = entityManager.createQuery(consulta, Ca-
chorro.class);
if(nome != null) query.setParameter("nome", nome);
if(idade != null) query.setParameter("idade", idade);
  
```

Agora com Criteria API,

```

CriteriaBuilder criteriaBuilder = entityManager.getCriteriaBuil-
der();
CriteriaQuery<Pessoa> criteriaQuery =
criteriaBuilder.createQuery(Pessoa.class);
Root<Pessoa> root = criteriaQuery.from(Pessoa.class);
TypedQuery<Pessoa> query = entityManager.createQuery(criteria-
Query);
  
```

## Criteria

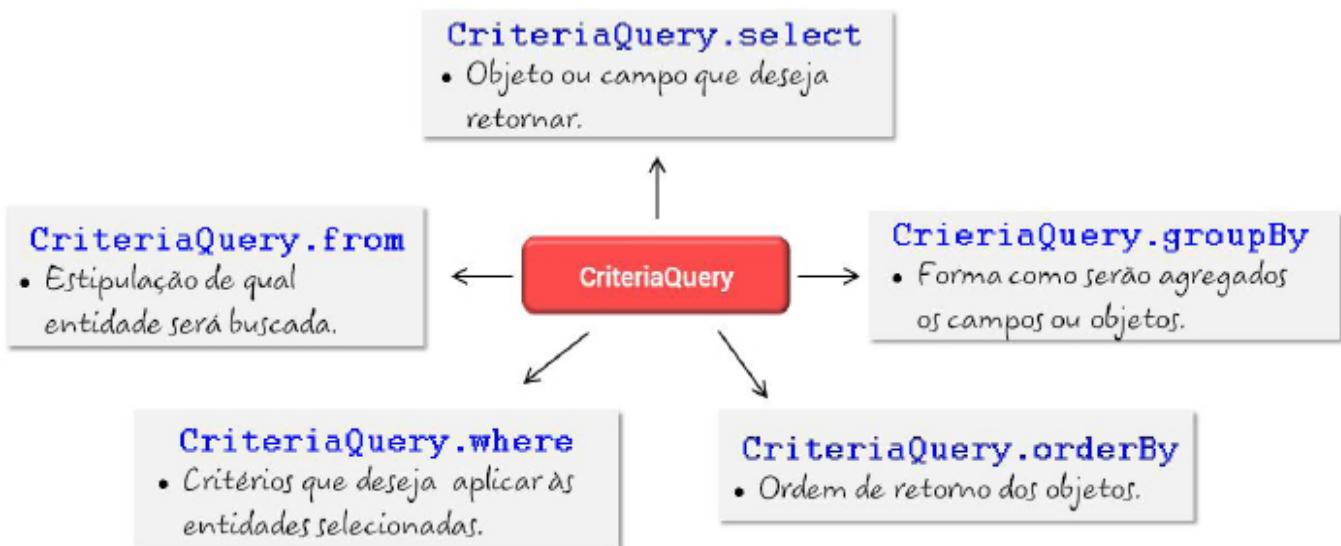


Para utilizar Criteria, devemos primeiro obter o CriteriaBuilder a partir do EntityManager ou do EntityManagerFactory com o comando getCriteriaBuilder(). O CriteriaBuilder é uma interface usada pra construir consultas criteria, expressões, predicados e ordenações.

CriteriaBuilder oferece vários métodos, sendo alguns deles:

- **createQuery(Class)** – cria um objeto de consulta do tipo CriteriaQuery com o resultado do tipo especificado.
- **CreateQuery()** - cria um objeto do tipo CriteriaQuery
- **createCriteriaDelete(Class)** – cria um objeto de consulta do tipo CriteriaDelete para realizar uma operação de delete.
- **createCriteriaUpdate(Class)** - cria um objeto de consulta do tipo CriteriaUpdate para realizar uma operação de update.
- **equal** – cria um predicado para testar a igualdade dos argumentos.
- **diff** – cria uma expressão que retorna a diferença entre seus argumentos.

## CriteriaQuery



A interface CriteriaQuery define funcionalidades que são específicas para consultas de alto nível, como:

- **select** – especifica a seleção de itens que serão retornados ao final da consulta.
- **where** – modifica a consulta para restringir o resultado da consulta de acordo com as restrições especificadas.
- **orderBy** – especifica o ordenamento do resultado da consulta.
- **groupBy** – especifica as restrições/expressões que serão usadas para agrupar o resultado da consulta.
- **from** – especifica a raiz da consulta a ser criada.

## CriteriaQuery

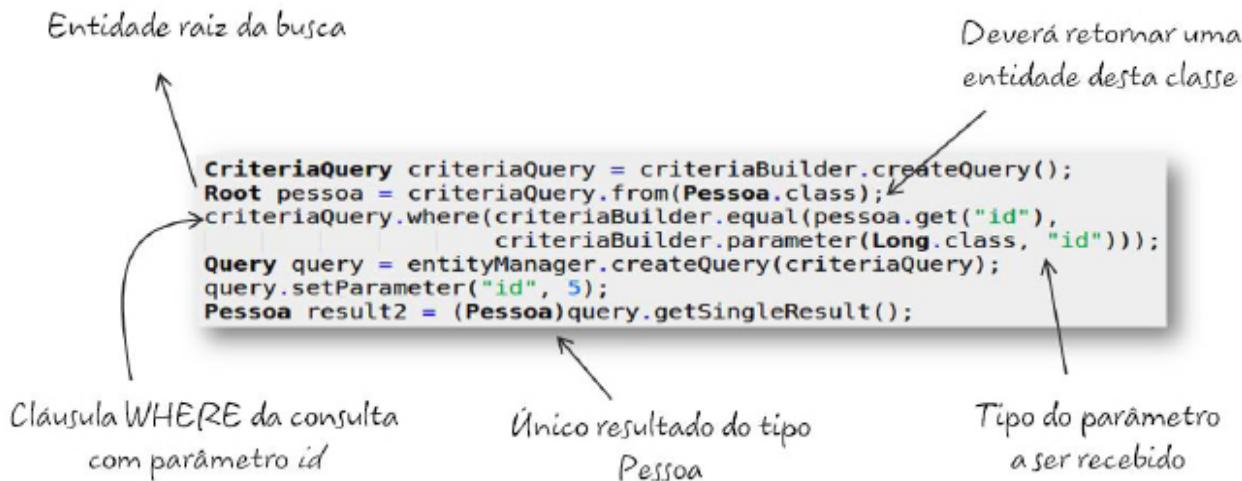
```
Entidade raiz da busca  
Deverá retornar uma  
entidade desta classe  
  
CriteriaQuery criteriaQuery = criteriaBuilder.createQuery();  
Root pessoa = criteriaQuery.from(Pessoa.class);  
criteriaQuery.where(criteriaBuilder.equal(pessoa.get("nome"), "Joao"));  
Query query = entityManager.createQuery(criteriaQuery);  
List<Pessoa> result = query.getResultList();  
  
Cláusula WHERE  
da consulta.  
  
Resultado será uma  
lista do tipo Pessoa
```

No exemplo criamos uma instância de CriteriaQuery a partir do entityManager.getCriteriaBuilder() e determinamos a entidade raiz (Root) da busca através do método from() de criteriaQuery.

Determinada a entidade de retorno, criamos uma cláusula WHERE através do método where() e a condição com o método equal(), procurando entidades cujo o atributo nome possua valor “Joao”. Criamos então uma instância de Query e a partir do entityManager.createQuery() passando como parâmetro a CriteriaQuery criada anteriormente. Para executar a consulta, utilize o comando getResultList().

Apesar do código bastante verboso há vantagens em se utilizar Criteria para situações que exigem a geração dinâmica de consultas em que não se conheça as estruturas exatas em tempo de execução.

## CriteriaQuery



A diferença de criar uma CriteriaQuery com parâmetros é que devemos definir quais são os parâmetros a partir do método parameter(class, identificador) de CriteriaBuilder. Devemos informar qual o tipo do parâmetro e seu nome. Depois basta utilizar o método setParameter como visto em exemplos anteriores.

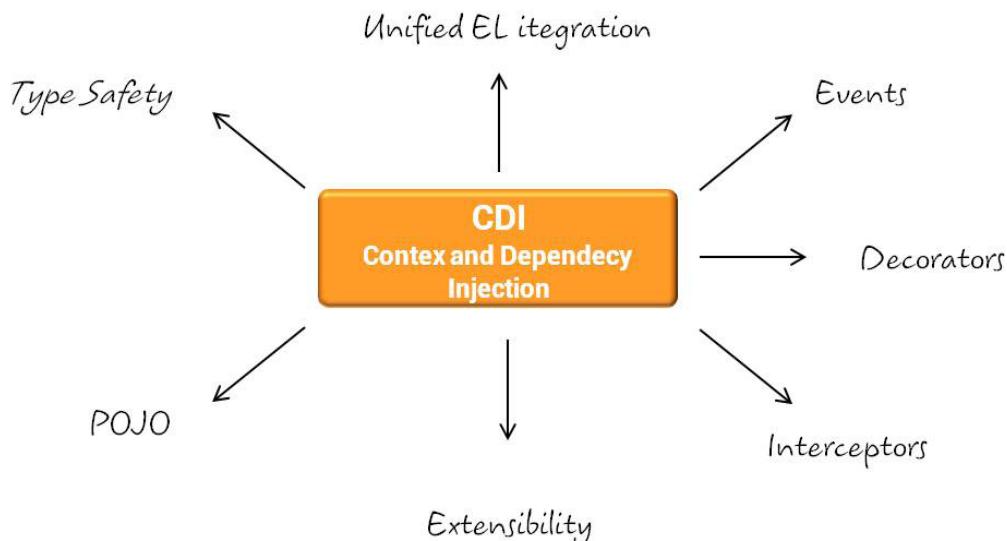
## JPA 2.1

A versão mais recente da JPA, possui algumas melhorias e novidades. Algumas delas são:

- **Attribute Converter** – conversores de código customizados entre o banco de dados e objetos.
- **Criteria UPDATE/DELETE** – updates e deletes em massa através da Criteria API.
- **Schema Generation** – geração automática de tabelas, indexes e esquemas de banco de dados.
- **Named Stored Procedure Query** – permite definir consultas para procedimentos armazenados no banco de dados.
- **Constructor Result Mapping** – suporte para SQLResultSetMapping.
- **Dynamic Named Queries** – criação de consultas em tempo de execução.
- **Melhorias na JPQL** – subconsultas aritiméticas, funções genéricas do banco de dados, cláusulas JOIN ON e opção TREAT.
- **Entity Graph** – permite fetching ou merging parcial ou específico de objetos.
- **Suporte para CDI no Entity Listener** – permite utilizar CDI para fazer injeção de beans em EntityListeners.

Persistence Context Dessincronizado – o persistence context sincronizado para propagar as mudanças no banco de dados é a aproximação padrão da JPA. Agora podemos ter uma dessincronização, possibilitando um maior controle sobre essa propagação com a anotação @PersistenceContext(synchronization=SynchronizationType.UNSYNCHRONIZED). Será necessário chamar EntityManager.joinTransaction() manualmente para sincronizar as mudanças.

## CDI Context and Dependency Injection



Recursos adicionados pelo CDI ao Java EE:

- EL unificada: 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 negócio;
- 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;
- Service Provider Interface (SPI): Permite a integração de frameworks third-party no ambiente Java EE.

Conceitos fundamentais da CDI são:

**Context** : liga o ciclo de vida e iterações entre componentes (com estado) ao ciclo de vida de contextos bem definidos mas flexíveis. Objetos gerenciados pela CDI “vivem” em escopos bem definidos: @RequestScoped, @SessionScoped, @ApplicationScoped e @ConversationScoped.

**Dependency Injection**: quando obtemos instâncias de objetos dependentes de forma indireta, no primeiro exemplo você cria o objeto EntityManager que sua classe precisa,

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("MeuPersistenceUnit");
```

```
EntityManager em = emf.createEntityManager();
```

Ou o container cria por você e injeta no ponto marcado por @PersistenceContext:

```
@PersistenceContext(unitName="MeuPersistenceUnit")  
EntityManager em;
```

## Inversão de Controle

O Framework vai entregar a dependência. Sua classe não precisa busca-la.

```
public class ClienteDaoImpl extends DaoImpl<Cliente> {  
    private static final long serialVersionUID = -1081947125480849486L;  
  
    private EntityManager manager;  
    public ClienteDaoImpl(EntityManager manager) {  
        this.manager = manager;  
    }
```

EntityManager é passado como parâmetro no construtor

Utilizar a injeção de dependências!

Antes de aprofundarmos mais na CDI, precisamos primeiro entender o conceito de inversão de controle.

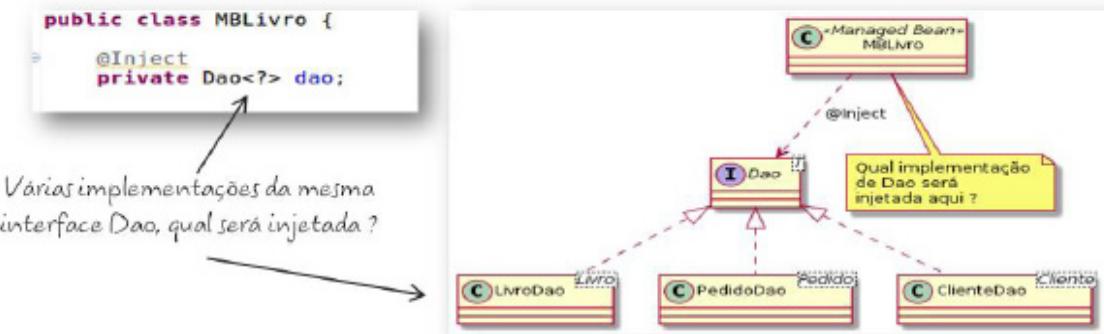
Ao permitirmos que o construtor receba um argumento do tipo EntityManager, a classe ClienteDaoImpl não precisa se preocupar com os detalhes de criação do EntityManager. Podemos delegar esse responsabilidade para outro componente:

```
public class Servico{  
    public void init(){  
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("MeuPersistenceUnit");  
        EntityManager em = emf.createEntityManager();  
        Dao clienteDao = new ClienteDaoImpl(em);  
    }  
}
```

O controle sobre a criação do EntityManager e de responsabilidade da classe Servico que entrega para a classe ClienteDaoImpl a dependência que ela precisa para operar sobre o banco de dados.

A injeção de dependência segue esse princípio, nos delegamos ao framework CDI a responsabilidade da criação e entrega dos objetos dependência nos pontos de injeção marcados com @Inject.

## Qualificadores – Ambiguidade de objetos



Beans são definidos como objetos gerenciados pelo contêiner 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 recursos, 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 modelos, serão interpretados pelo contêiner, que por sua vez, irá gerenciar o ciclo de vida destes objetos. Ou seja, um bean pode requerer instâncias de outros beans.

Beans podem ser integrados com o JSF. Para isso, devemos anotar uma classe bean com `@Named`, isso fará com que o nome da classe possa ser utilizado pela EL.

## Beans

```
public class Carro {  
  
    private String modelo;  
    private String chassi;  
  
    public Carro() {}
```

- Classe não estática
- Construtor vazio ou anotado com @Inject
- Classe não declarada como um bean EJB
- Classe não anotada com um componente EJB

Com a CDI podemos injetar esse EntityManager, apenas utilizando a anotação @Inject para indicar qual dependência está sendo injetada.

Há três classes de objetos que podem ser injetados:

- Managed beans: EJB session beans, classes anotadas com @javax.annotation.ManagedBean, e classes que satisfazem as regras do CDI para se tornarem um Managed Bean CDI.
- Recursos Java EE: datasources, serviços de mensagens, filas, fabricas de conexão, Persistence Contexts (JPA EntityManager)
- Objetos retornados por métodos produtores (@Producer).

Managed Beans CDI, são classes POJO (Plain Old Java Object). Com pouquíssimas exceções, quase toda classe concreta em Java que possui um construtor público com nenhum parâmetro (ou um construtor designado com a anotação @Inject) é um Managed Bean CDI.

Não são Managed Beans CDI: classes internas não estaticas, componentes EJB (não confundir, eles são injetados mas não são geridos pela CDI).

Um ponto de injeção em um bean é marcado pela anotação @Inject, ele pode ocorrer em três locais de seu código: em campos (atributos), em construtores e nos métodos de inicialização (métodos setters).

## Ponto de Injeção – No Campo

```
public class AbstractDao<T> implements Dao<T> {  
    @Inject  
    protected Connection conexao;
```

Injeta um objeto java.sql.Connection, no campo

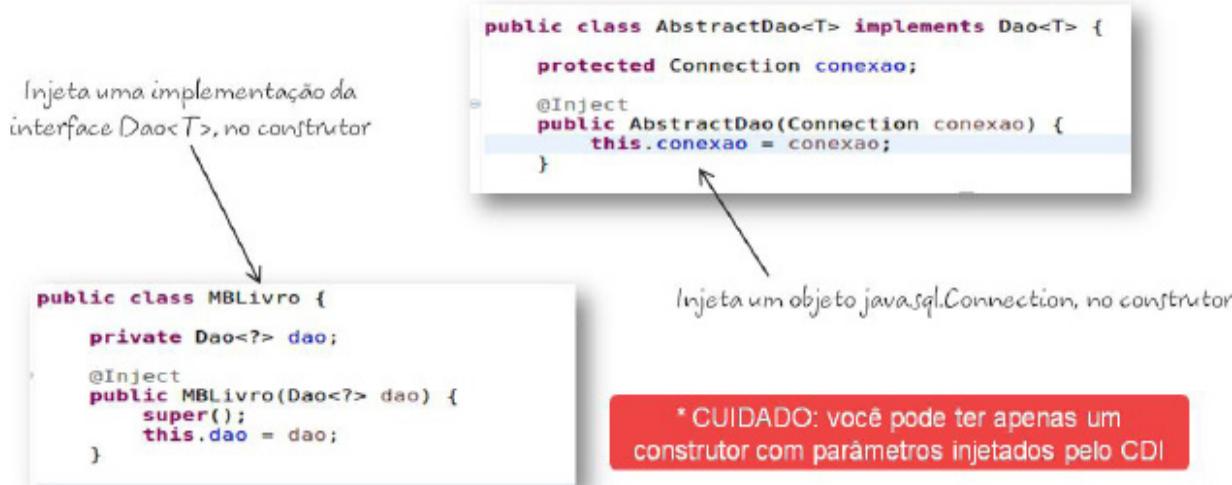
Injeta uma implementação da interface Dao<T>, no campo

```
public class MBLivro {  
    @Inject  
    private Dao<?> dao;
```

Você cria um ponto de injeção através de um atributo, ao anotar o atributo que deseja injetar com `@Inject`.

O container CDI irá verificar se há um bean que satisfaça a definição de tipo usada pelo atributo, durante o processo de inicialização da aplicação. Não havendo um bean com tipo compatível ou se houver alguma dependência ambígua, uma mensagem de alerta será emitida ao usuário e a variável do campo a ser injetado ficará null.

## Ponto de Injeção – No Construtor



Você pode usar a anotação `@Inject` com o construtor, ou pode também mesclar com pontos de injeção por campos e em contrutores na mesma classe se você precisar:

```

@WebServlet(urlPatterns = "/itemServlet")
public class ItemServlet extends HttpServlet {
    private CestaDeItens cesta;
    @Inject
    private ItemEJB itemEJB;
    @Inject
    public ItemServlet(CestaDeItens cesta) {
        this.cesta = cesta;
    }
    ...
}

```

A única restrição é que apenas um ponto de injeção no construtor deve existir, ou seja, não podemos ter dois construtores anotados com `@Inject`.

## Ponto de Injeção – No Método de Configuração

```
public class AbstractDao<T> implements Dao<T> {
    protected Connection conexao;
    @Inject
    public void setConexao(Connection conexao) {
        this.conexao = conexao;
    }
}
```

Injeta um objeto java.sql.Connection, no setter

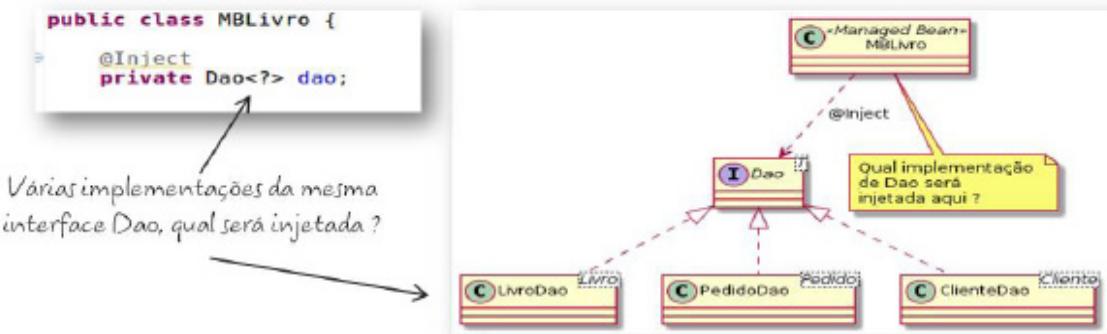
Injeta uma implementação da interface Dao<T>, no setter

```
public class MBLivro {
    private Dao<?> dao;
    @Inject
    public void setDao(Dao<?> dao) {
        this.dao = dao;
    }
}
```

A injeção através de setters é parecida com a injeção por construtores, basta anotar o setter com `@Inject` e qualificar os parâmetros se for preciso. Veja outro exemplo

```
@WebServlet(urlPatterns = "/itemServlet")
public class ItemServlet extends HttpServlet {
    private GeradorDeNumero numeroGenerador;
    private ItemEJB itemEJB;
    @Inject
    public void setNumberGenerator(@TrezeDigitos GeneradorDeNumero numeroGenerador) {
        this.numeroGenerador = numeroGenerador;
    }
    @Inject
    public void setItemEJB(ItemEJB itemEJB) {
        this.itemEJB = itemEJB;
    }
    ...
}
```

## Qualificadores – Ambiguidade de objetos

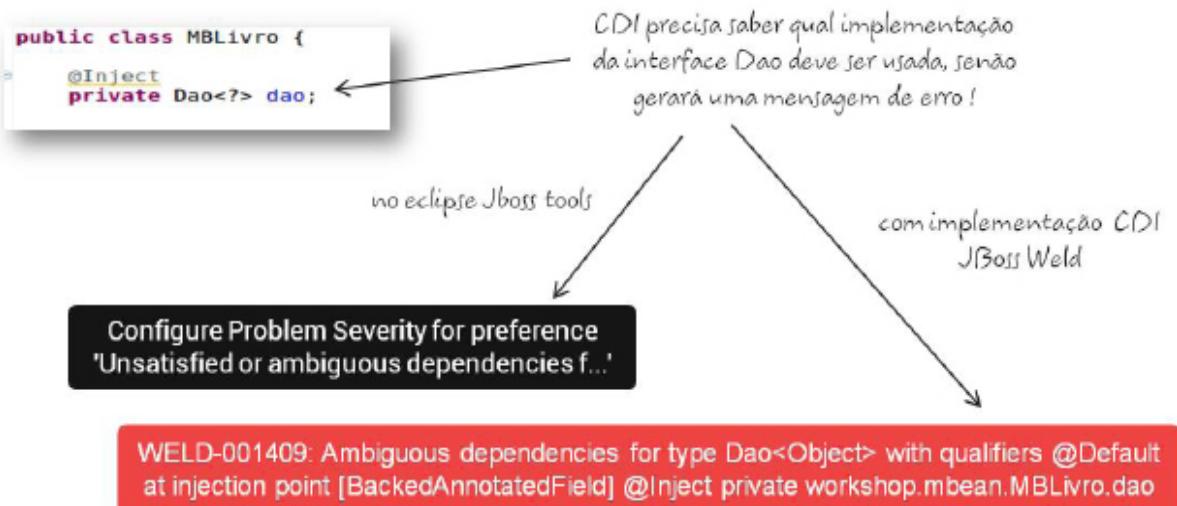


Como você pode especializar classes através de herança ou pela implementação de interfaces, é possível que sua aplicação possua mais do que uma implementação para um mesmo tipo de bean, no exemplo acima temos a interface Dao<T>, essa interface é implementada pela classe abstrata AbstractDao<T>, daí você pode ter várias especializações de AbstractDao<T>, como por exemplo: PedidoDao<Pedido> , ClienteDao<Cliente>, LivroDao<Livro>, etc.

Nesse caso o mecanismo de “typesafe resolution” da CDI, precisa de uma indicação de qual o tipo correto a ser injetado pelo container CDI no ponto de injeção, com apresentado na classe MBLivro, onde queremos que seja injetado um objeto do tipo LivroDao.

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 bean candidato que possa suprir uma dependência de outro.

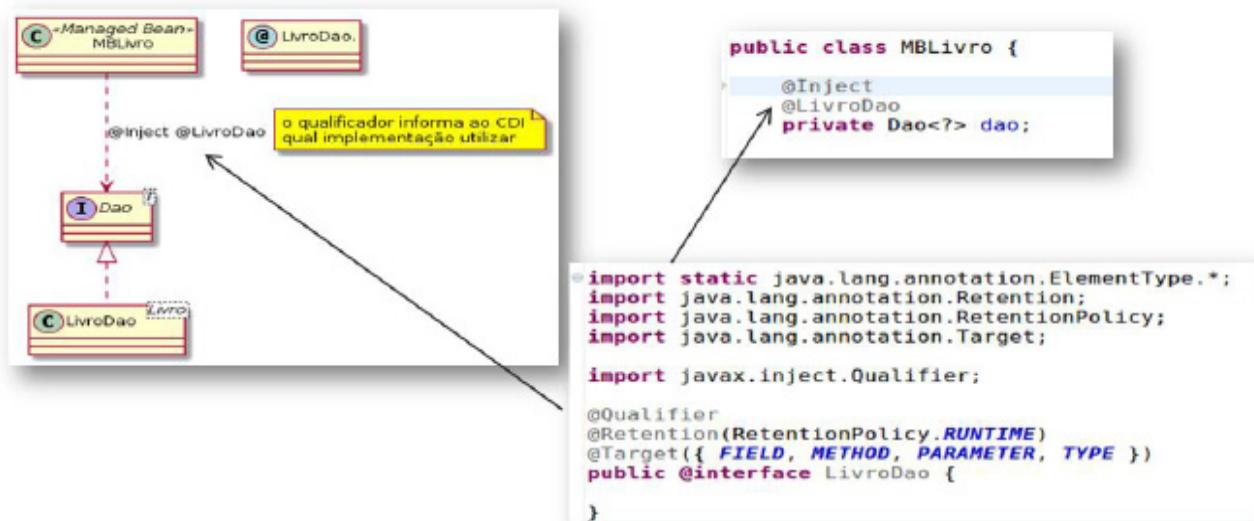
## Qualificadores - Desambiguidade



Uma das vantagens de usarmos java como linguagem de programação é que ela possui um mecanismo de tipagem segura. Esse mesmo benefício está presente também no CDI, o “type-safe resolution” do CDI, emitirá mensagens de erro ou advertência durante o carregamento dos beans CDI, se o mesmo não conseguir resolver adequadamente qual o bean específico será utilizado no ponto de injeção. Acima temos exemplo de uma mensagem de advertência sobre a ambiguidade de tipos; IDE's como o eclipse, com plugin Jboss Tools, também podem emitir alertas durante o desenvolvimento.

Observe que na mensagem de alerta há uma indicação de que o container CDI está tentando injetar beans com qualificadores `@Default` do tipo `Dao<Object>` (mesmo que `Dao<?>`). A anotação `@Default` é um qualificador padrão do CDI, todo bean CDI é anotado com `@Default`, logo senão houver outro qualificador não será possível ao CDI distinguir se usará, neste caso, objetos do tipo `PedidoDao<Pedido>` ou do tipo `ClienteDao<Cliente>` ou do tipo `LivroDao<Livro>`, para o ponto de injeção do exemplo.

## Qualificadores – Criando Anotações Qualificadoras



Para indicar qual o bean CDI deve ser injetado, você fará uso de qualificadores para direcionar a correta injeção. Um qualificador é uma anotação anotada com `@Qualifier`, que você criará, como por exemplo:

```

package annotation;

@Qualifier

@Retention(RUNTIME)
@Target({FIELD, TYPE, METHOD, PARAMETER})
public @interface LivroDao {
}

```

Criado esse qualificador é necessário anotar a classe `LivroDao`,

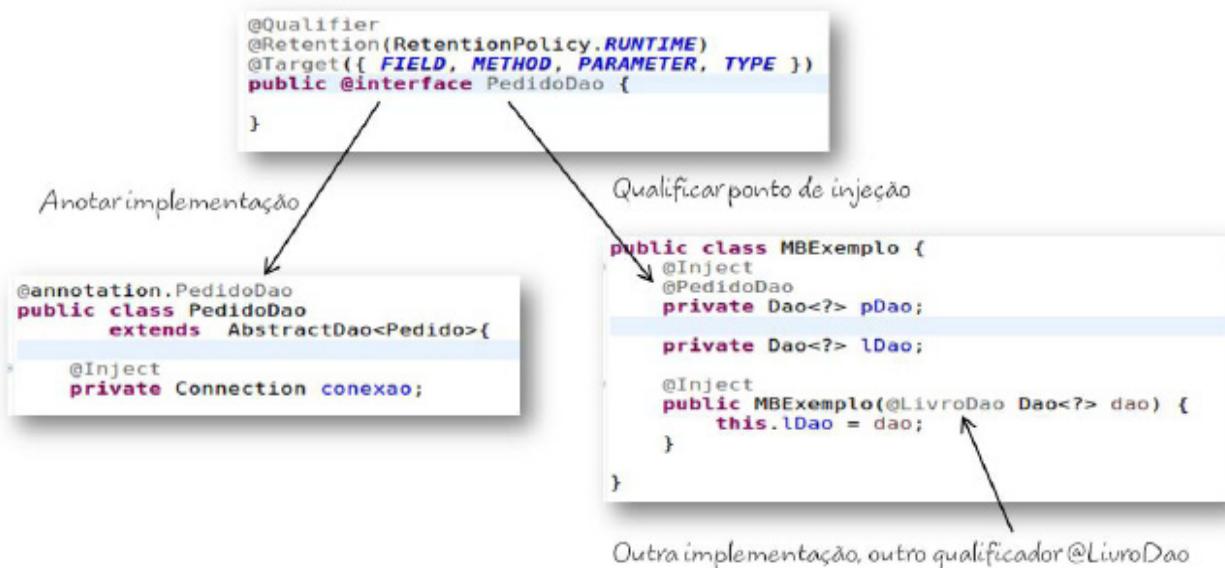
```

@annotation.LivroDao
public class LivroDao extends AbstractDao<Livro>{
    ...
}

```

que é o bean qualificado, e anotar o ponto de injeção, o campo `dao` no bean `MBLivro`, com a nova anotação qualificadora, no caso `@LivroDao`.

## Qualificadores – Usando Qualificadores



Observe como os qualificadores `@PedidoDao` e `@LivroDao` são usados em pontos de injeção em campo e em construtor, respectivamente. O campo `lDao` é configurado pelo construtor que receberá um objeto do tipo `LivroDao` qualificado por `@LivroDao` e o campo `pDao` receberá um objeto `PedidoDao` qualificado por `@PedidoDao`.

A CDI pré-define alguns qualificadores (built-in qualifiers):

- `@Named`
- `@Default`
- `@Any`

Todo bean é anotado como `@Any`, já o qualificador `@Default`, chamado de qualificador padrão, existe em todos os beans implicitamente. O qualificador `@Named` apenas torna um bean disponível para a JSF EL.

## Alternatives

Anotação `@Alternative` permite especificar qual bean utilizar, basta indicar no beans.xml/

```
public class ImplementacaoPadrao implements ExemploInterface {  
    //Implementação padrão  
}  
  
@Alternative  
public class ImplementacaoAlternativa implements ExemploInterface {  
    //implementação alternativa  
}
```

Toda vez que você precisar escolher qual implementação de um bean usar você criará uma anotação. Por exemplo, se você precisar qualificar um bean de conexão para o banco de dados Postgresql, MySql, Oracle e Informix, então você terá que criar, no caso, quatro anotações. Assim à medida que você vai precisando qualificar mais e mais implementações de beans, mais e mais anotações precisam ser criadas.

Um modo de evitar a multiplicação de anotações é usar enumerações como mostrado na figura. Podemos ainda definir um valor padrão para esse qualificador, basta utilizar default value da enumeração, como na definição do qualificador `@Conexao`:

```
EnumConexao value() default EnumConexao.POSTGRES;
```

assim, quando precisar de uma conexão com Postgres use apenas `@Conexao` ou `@Conexao(EnumConexao.POSTGRES)`, se precisar de uma conexão com MySql use `@Conexao(EnumConexao.MYSQL)`.

## Injeção a partir métodos Produtores

Objetos que não  
são managed  
bean CDI podem  
ser injetados com  
@Producers

```

import java.sql.Connection;

public class FabricaConexao {
    static String url = "jdbc:postgresql://localhost:5432/Livraria";
    static String usuario = "postgres";
    static String senha = "postgres";

    @Produces
    public static Connection getConexao() throws SQLException{
        return DriverManager.getConnection(url,usuario,senha);
    }

    public static void closeConnection(@Disposes Connection conn)
        throws SQLException{
        System.out.println("FabricaConexao.closeConnection()");
        conn.close();
    }
}

public class AbstractDao<T> implements Dao<T> {
    protected Connection conexao;

    @Inject
    public AbstractDao(Connection conexao) {
        this.conexao = conexao;
    }
}

```

← @Disposes  
fechando o Connection

Com o CDI você pode injetar qualquer coisa em qualquer lugar, por exemplo você pode injetar um objeto `java.util.String`, `java.util.Date` ou até mesmo um tipo primitivo como `long`. Porém você não pode injetá-los diretamente, você não consegue fazer:

```

@Inject
java.util.Date data;

```

você precisará dizer ao CDI como produzir esses objetos, como criá-los antes de poder injetá-los em sua aplicação.

Para tanto, o CDI tem produtores (`@Produces` uma implementação do design pattern Factory). Um produtor CDI pode expor qualquer tipo de classe, interfaces, tipos primitivos e arrays, para injeção.

Você só precisa anotar o campo ou método produtor com `@Produces`, na figura o método `getConexao()` foi anotado com `@Produces`, o método retorna um objeto do tipo `java.sql.Connection`, daí você já pode injetar objetos deste tipo em qualquer ponto da sua aplicação usando `@Inject`. Da mesma maneira podemos ter produtores

```

@Produces public Date date = new Date(); //no campo
@Produces public long nlong = 17L; //no campo
@Produces @Ontem
public Date ontem() { //método
    return Date.from(i.minus(1, ChronoUnit.DAYS));
}

```

## Injeção de EJB a partir de Produtores

Beans EJB não são managed bean CDI, mas podem ser injetados com @Producers

```
public class FabricaConexao {
    @Produces
    @PersistenceContext
    private EntityManager em;

    public void closeConnection(@Disposes EntityManager em)
        throws SQLException{
        em.close();
    }
}

public class AbstractDao<T> implements Dao<T> {
    @Inject
    protected EntityManager em;
}
```

↑  
@Disposes  
fechando o EntityManager

Então se CDI injeta qualquer coisa em qualquer lugar, se CDI pode injetar até mesmo tipos Java com String e tipos primitivos, então podemos fazer o mesmo com recursos JEE? Isto é uma outra estória e os produtores vão nos ajudar aqui. Como foi dito antes, CDI está fundamentado em segurança de tipos de dado: CDI não usa Strings de nomes de recursos, ele usa o tipo de dados do recurso, então não há como injetar recurso pelo seu nome JNDI tal como @Inject (name="global/datasources/PostgresDS"). Um exemplo comum é o entity manager. Veja como ele é injetado em Java EE 6 sem usar CDI:

```
@Stateless
public class ItemEJB {
    @PersistenceContext(unitName = "cdiPU")
    private EntityManager em;
}
```

Veja na figura como ele é injetado usando CDI, usamos um produtor, e então podemos usar @Inject em algum ponto de injeção.

Recursos alocados normalmente precisam ser liberados, isto é facilmente obtido com @Disposes, como visto na figura, o método closeConnection(@Disposes ...), assim o CDI invoca esse método quando o managed bean que faz uso do recurso for descartado pelo CDI. Não foi comentado ainda, mas o CDI irá criar e descartar recursos dependendo de seu escopo (request, session, application, conversation, etc), veremos mais adiante.

## Desambiguação de Produtores

@Producers ambíguos devem ser qualificados

```
public class FabricaConexao {
    @Produces
    @PersistenceContext(unitName="pu-prod")
    private EntityManager prodEm;

    @Produces
    @PersistenceContext(unitName="pu-test")
    private EntityManager testEm;
}
```

Crie @Qualifier's para diferenciar

```
@Qualifier
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.METHOD, ElementType.FIELD})
@Documented
public @interface ProdDatabase { }
```

```
@Qualifier
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.METHOD, ElementType.FIELD})
@Documented
public @interface TestDatabase { }
```

E porque não podemos simplesmente usar @Inject com um entity manager? Por causa da ambiguidade de tipos de dados, você poderia ter várias unidades de persistência (nomeado por uma string), daí se você simplesmente usar @Inject o CDI não conseguirá resolver qual unidade de persistência usar.

Aqui o problema será resolvido da mesma forma que fizemos anteriormente, é necessário criar qualificadores para todo recurso em que haja a possibilidade de ambiguidade para o CDI. Recorde-se que o CDI irá reportar uma mensagem de erro se ele não conseguir distinguir qual implementação utilizar.

Neste caso você deverá criar e qualificar (se houver mais que uma unidade de persistência) cada produtor.

## Desambiguação de Produtores

```
public class FabricaConexao {
    @Produces
    @ProdDatabase
    @PersistenceContext(unitName="pu-prod")
    private EntityManager prodEm;

    @Produces
    @TestDatabase
    @PersistenceContext(unitName="pu-test")
    private EntityManager testEm;
}
```

@Producers qualificados  
Use qualificador  
@Qualifier no ponto de  
injeção para  
diferenciar o produtor

```
public class AbstractDao<T> implements Dao<T> {
    @Inject
    @ProdDatabase
    protected EntityManager em;
```

O exemplos mostrados são bem simples: produtores anotados e qualificado nos campos e então os injetamos em algum ponto. Mas há produtores com uma lógica de inicialização mais complexa para criação do bean, neste caso devemos usar os métodos produtores. Por exemplo para injetar recursos de mensagens (arquivos de message bundle):

```
public class MessageProvider {

    @Produces @MessageBundle
    public ResourceBundle getBundle() {
        if (bundle == null) {
            Application app=FacesContext.getCurrentInstance().getApplication();
            bundle=context.getApplication().getResourceBundle(context, "msgs");
        }
        return bundle;
    } ...
}
```

Por fim injetamos:

```
@Named
@RequestScoped
public class SomeBean {
    @Inject @MessageBundle
    private ResourceBundle bundle;
...
}
```

## Alternative usando Esterótipo

```
@Alternative  
@Stereotype  
@Retention(RUNTIME)  
@Target(TYPE)  
public @interface ImplAlternativa { }  
  
@ImplAlternativa  
public class ImplementacaoAlternativa implements ExemploInterface {  
    //implementação alternativa  
}
```

### Declaração no beans.xml

```
<alternatives>  
    <stereotype>exemploCdi.ImplAlternativa</stereotype>  
</alternatives>  
  
<alternatives>  
    <class>exemplosCdi.ImplementacaoAlternativa</class>  
</alternatives>
```

Alternatives são usadas quando você tem mais de uma versão de um bean que você usa para diferentes finalidades. Geralmente você teria que alterar o código injetando um qualifier diferente. Alternatives permitem que você faça essa escolha do que usar em tempo de implantação.

Poderíamos criar uma versão completa de um bean e outra mais simples para fins de teste. A implementação mais simples deverá ser anotada com @Alternative.

## Interceptadores Anotação @InterceptorBinding

Criamos uma anotação com  
{@InterceptorBinding} para  
vincularmos um  
interceptador

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import javax.interceptor.InterceptorBinding;

@InterceptorBinding
@Retention(RetentionPolicy.RUNTIME)
@Target({ ElementType.TYPE, ElementType.METHOD })
public @interface Transactional {

}
```

Na CDI, um esteriótipo encapsula várias propriedades incluindo escopo, interceptadores de ligação, qualificadores, etc, dentro de um simples pacote reusável. Em termos de código, um esteriótipo é apenas uma anotação anotada com @Stereotype.

```
@Stereotype
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.METHOD, ElementType.FIELD})
@Documented
public @interface OutraImplAlternativa {
}
```

Para ativar a implementação de bean alternativo, basta declarar no arquivo beans.xml qual esteriótipo será utilizado.

## Produtores e Escopo

Quando será produzido?

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

O tipo de retorno  
define o que será  
produzido

Fechando o EntityManager  
@Disposes

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

Métodos produtores  
são uma forma fácil de  
integrar objetos que  
não são beans ao  
ambiente CDI

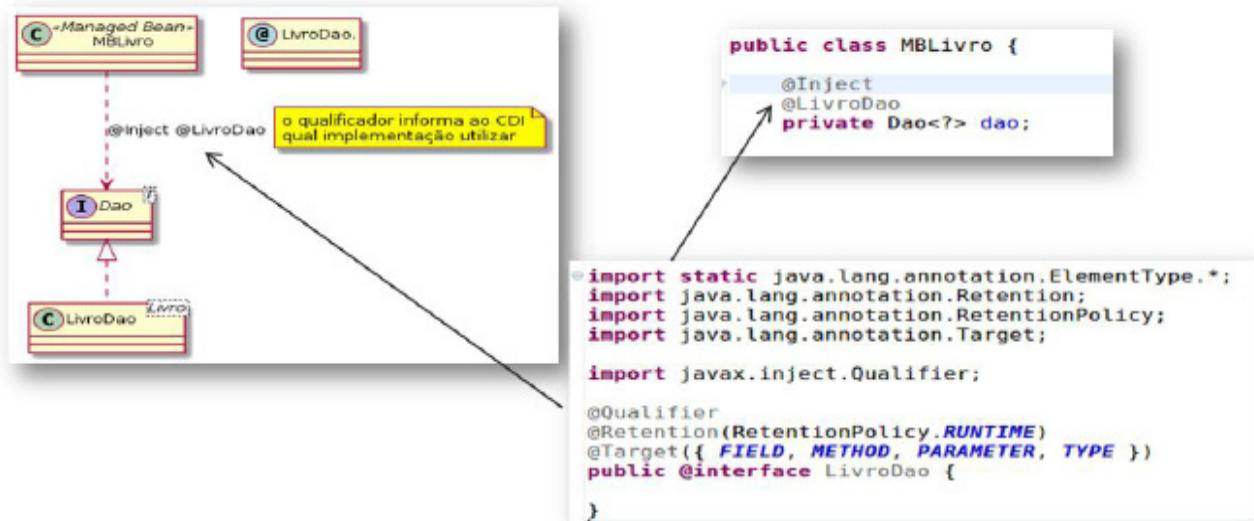


WWW.3way.com.br

Todo managed bean CDI é criado em um escopo, por padrão vimos que ele recebe o escopo dependente, ou seja, o mesmo escopo do bean que solicita a injeção, sendo criado quantas vezes for necessário. Na figura colocamos o escopo de requisição (@RequestScope), assim temos apenas uma instância de EntityManager por requisição.

Como vimos, recursos precisam ser removidos quando o seu trabalho for completado. Você então pode cirar um método disposer, anotado com @Disposes, como no exemplo acima. O método disposer é chamado automaticamente quando o contexto terminar (nesse caso ao fim da requisição já que estamos usando Request Scope). O parâmetro recebido pelo método disposer é o mesmo produzido pelo método produtor.

## Qualificadores – Criando Anotações Qualificadoras



O ciclo de vida de um bean dependerá do escopo em que ele foi inserido. Por sua vez, os diferentes escopos definem ao contêiner 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.

Temos os três escopos padrão:

**Request Scope (@RequestScoped)** – Começa na requisição e termina quando o servidor devolve a resposta ao cliente.

**Session Scope (@SessionScoped)** – Começa com a primeira requisição (início da sessão) feita e termina quando a aplicação encerra a sessão ou por inatividade do cliente.

**Application Scope (@ApplicationScoped)** – Persiste durante toda a vida da aplicação, a partir da hora em que ela for instanciada.

Session, Application e Request funcionam exatamente como são definidos nas aplicações Java EE. Além desses três escopos, temos: escopo de conversação (**@ConversationScoped**) e escopo dependente (**@Dependent**).

**Cuidado:** as anotações de escopos da JSF estão no pacote javax.faces.bean, enquanto as do CDI estão no pacote javax.enterprise.context.

## Managed Beans @Named

A anotação `@Named` integra o CDI com JSF, dando um nome ao bean.

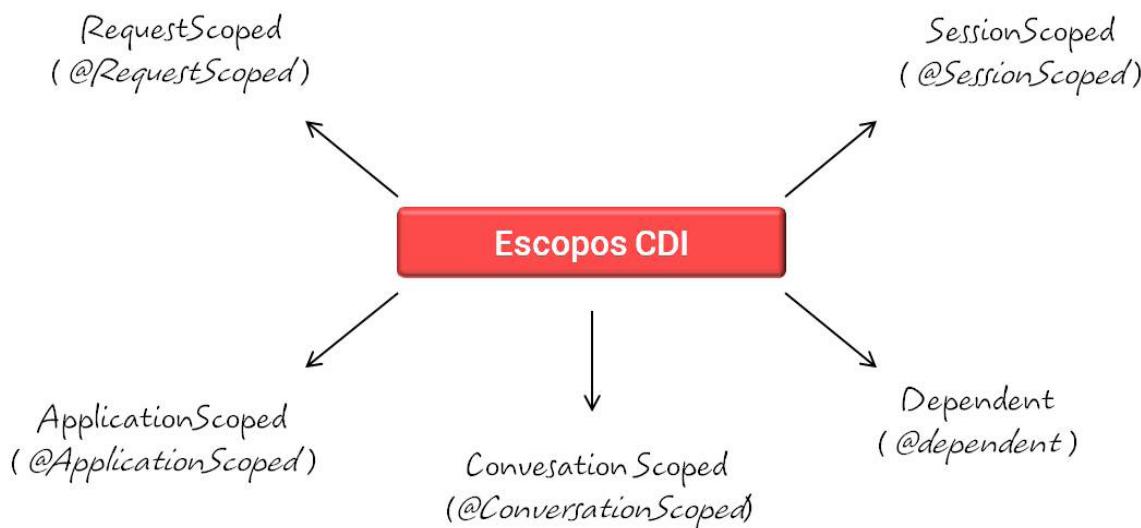
```
@Named  
@RequestScoped  
public class LoginBean implements Serializable {
```

```
<p:commandButton action="#{loginBean.realizarLogin()}"
```

Conversation Scope (@ConversationScoped) é parecido com o Session, pois também mantém o estado associado a um usuário do sistema. O início e o fim do escopo são programáveis, o container associa automaticamente um parâmetro CID na resposta à requisição que inicia a conversação. Este CID deve ser utilizado para recuperar o escopo nas próximas requisições. Veja um exemplo de uso:

```
@Inject  
private Conversation conversation;  
...  
public void inicioConversation(){  
    if (!FacesContext.getCurrentInstance().isPostback()  
        && conversation.isTransient()) {  
        conversation.begin();  
    }  
}  
  
public String fimConversation(){  
    if(!conversation.isTransient()) {  
        conversation.end();  
    }  
    return "step1?faces-redirect=true";  
}
```

## Escopos CDI



O escopo dependente (@Dependent) é o padrão para qualquer bean. Quando injetamos um objeto em um bean, esse objeto assume o mesmo escopo do bean. As informações do objeto do bean injetado ficarão disponíveis enquanto o bean que o injetou estiver disponível.

## Escopos CDI

Funcionam como os escopos do JSF

### **RequestScoped ( @RequestScoped )**

Começa na requisição e termina quando o servidor devolve a resposta.

### **SessionScoped ( @SessionScoped )**

Começa com a primeira requisição feita e termina quando a aplicação encerra a sessão ou por inatividade.

### **ApplicationScoped ( @ApplicationScoped )**

Dura enquanto a aplicação estiver ativa.

*Agora temos Conversation Scoped*

CDI Beans criados por @Producers tem por padrão escopo @Dependent. Um produtor sem escopo definido será invocado toda vez que o container CDI precisar injetar um bean do tipo e qualificador que case com esse produtor. Isso implica que você vai ter uma instância separada do bean para cada chamada que container fizer ao produtor, algo nem sempre desejável.

Você pode determinar quando esse objeto será produzido, basta adicionar uma das anotações de escopo.

Veja o exemplo, no caso colocamos o escopo de requisição @RequestScoped, já que precisamos de apenas um EntityManager por requisição.

Os objetos gerados por um método produtor em algum momento precisa ser removido quando o seu trabalho for completado. Você então pode usar @Disposes no método, como no exemplo acima. O método disposer é chamado automaticamente quando o contexto terminar (nesse caso ao fim da requisição já que estamos usando Request Scope). O parâmetro recebido pelo método disposer é o mesmo produzido pelo método produtor.

## Interceptadores

Indica que essa classe será a interceptadora

A anotação `@AroundInvoke` indica para o sistema qual método realizará a interceptação dos métodos de negócio

```

@Interceptor
@Transactional
public class TransactionInterceptor implements Serializable {

    private static final long serialVersionUID = 1L;

    @Inject
    private EntityManager manager;

    @AroundInvoke
    public Object invoke(InvocationContext context) throws Exception {
        EntityTransaction trx = manager.getTransaction();
        boolean criador = false;
        try {
            if (!trx.isActive()) {
                trx.begin();
                trx.rollback();
                trx.begin();
                criador = true;
            }
            return context.proceed();
        } catch (Exception e) {
            if (trx != null && criador) {
                trx.rollback();
            }
            throw e;
        } finally {
            if (trx != null && trx.isActive() && criador) {
                trx.commit();
            }
        }
    }
}

```

Um interceptador, como o próprio nome diz, intercepta invocações e eventos do ciclo de vida em uma classe alvo, por exemplo: uma chamada de método da classe.

Podemos usar interceptadores para vários coisas, como registrar logs, auditoria ou executar tarefas repetitivas e que não fazem parte da regra de negócios do sistema.

Você precisa definir uma anotação anotada com a meta-anotação `@InterceptorBinding`. Observe que `@Target` define o elemento do programa que poderá ser interceptado. Neste caso a anotação `@Transactional` pode ser aplicada a um método ou tipo (class, interface ou enum).

## CDI e JPA/EJB

✓ Os beans do EJB são:

- Transacionais
- Remotos ou locais
- Podem passar Stateful beans, liberando recursos
- Podem fazer o uso de Timer
- Podem ser assíncronos

✓ Um bean anotado com `@Stateless` não precisa lidar com transações, o servidor irá cuidar disso.

✓ Caso queira utilizar JPA em um container como Tomcat, as transações devem ser feitas manualmente.

✓ Beans EJB são beans CDI, portanto tem os mesmos benefícios, mas dizer o contrário não é válido. \

Para aplicar um interceptador você precisa marcar sua classe com a anotação `@Intercept-`  
`or`.

`@Transactional`, define o tipo do interceptador, uma implementação de interceptador específica.

`@AroundInvoke` indica que este método interceptador se interpõem aos métodos de negócio. Apenas um método da classe interceptadora marcado com essa anotação será invocado.

Observe a assinatura do método `Object invoke(InvocationContext context)`, o parâmetro `InvocationContext` possui informações de contexto sobre as invocações interceptadas e as operações, pode ser usada para controlar o comportamento da cadeia de invocações, além de informações sobre a invocação do método de negócio e seus parâmetros.

O método retorna o próximo interceptador da cadeia invocando `context.proceed()`.

Interceptadores suportam injeção de dependência, no exemplo estamos injetando um objeto `EntityManager`.

## Injeção a partir métodos Produtores

Objetos que não  
são managed  
bean CDI podem  
ser injetados com  
@Producers

```

import java.sql.Connection;

public class FabricaConexao {
    static String url = "jdbc:postgresql://localhost:5432/Livraria";
    static String usuario = "postgres";
    static String senha = "postgres";

    @Produces
    public static Connection getConexao() throws SQLException{
        return DriverManager.getConnection(url,usuario,senha);
    }

    public static void closeConnection(@Disposes Connection conn)
        throws SQLException{
        System.out.println("FabricaConexao.closeConnection()");
        conn.close();
    }
}

public class AbstractDao<T> implements Dao<T> {
    protected Connection conexao;

    @Inject
    public AbstractDao(Connection conexao) {
        this.conexao = conexao;
    }
}

```

@Disposes  
fechando o Connection

Por padrão todos os interceptadores estão desabilitados. Eles precisam ser habilitados explicitamente e ordenados via anotação @Priority na classe interceptadora.

Você habilita um interceptador declarando explicitamente o nome canonico da classe de implementação no arquivo beans.xml.

```

<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
           http://xmlns.jcp.org/xml/ns/javaee
           http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
       bean-discovery-mode="annotated">

    <interceptors>
        <class>interceptor.exemplo.TransactionInterceptor</class>
    </interceptors>
</beans>

```

Você pode adicionar vários interceptadores, inclusive do mesmo tipo. Os interceptadores são invocados na ordem em que eles são especificados dentro da tag <interceptors>.

## Injeção de EJB a partir de Produtores

Beans EJB não são managed bean CDI, mas podem ser injetados com @Producers

```
public class FabricaConexao {
    @Produces
    @PersistenceContext
    private EntityManager em;

    public void closeConnection(@Disposes EntityManager em)
        throws SQLException{
        em.close();
    }
}
```

@Disposes  
fechando o EntityManager

```
public class AbstractDao<T> implements Dao<T> {
    @Inject
    protected EntityManager em;
```

O CDI fornece um modelo de eventos baseado em anotações que implementam o padrão Observer. Produtores disparam eventos que são consumidos por observadores. O objeto de evento, normalmente um POJO, carrega o estado do objeto produtor para o objeto consumidor. O objeto produtor e consumidor são completamente desacoplados e a comunicação entre eles somente se dá através de estado.

O objeto produtor disparará um evento através da interface Event:

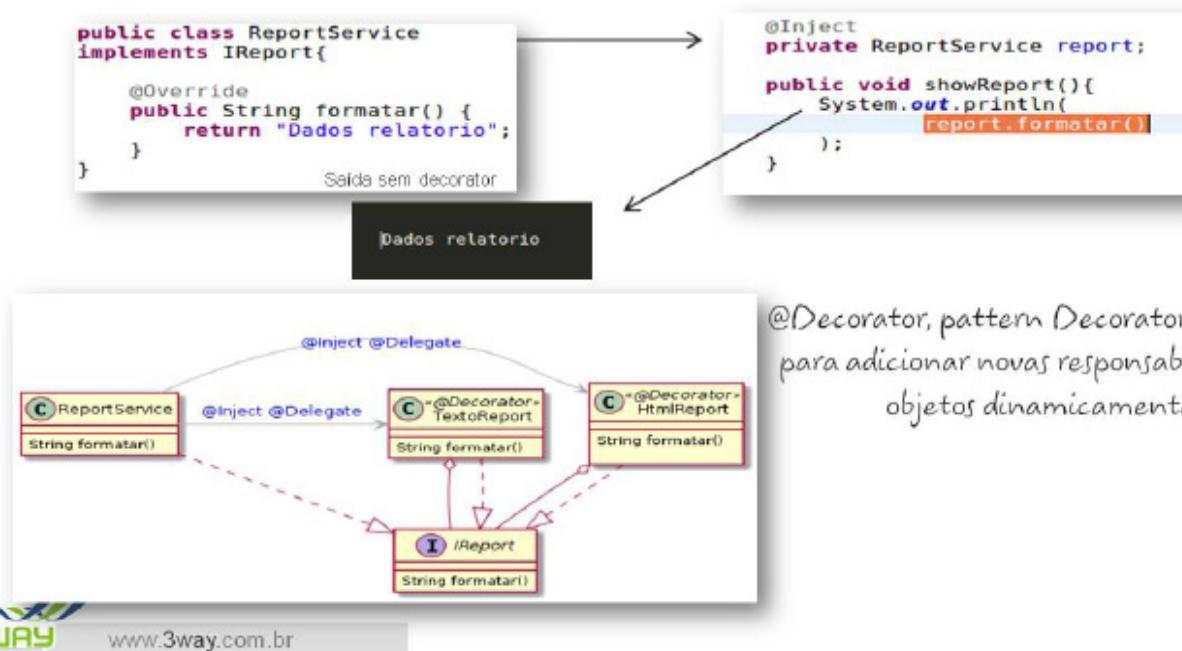
```
@Inject Event<Carro> event;
//....
event.fire(carro);
```

Um bean consumidor (observador), com seguinte assinatura de método irá receber o evento:

```
void notificarCliente(@Observes Carro carro) { ... }
```

Neste código, carro carrega o estado do evento.

## Decorators – Decorando seu Bean



O funcionamento é similar ao que vimos com `@Alternative @Stereotype`, ou seja, podemos indicar uma implementação específica a ser injetada habilitando a classe “decoradora” em beans.xml.

Decorator é um design pattern que te permite reaproveitar código existente e criar algum recurso a mais sem precisar reescrever ou extender subclasses da classe original.

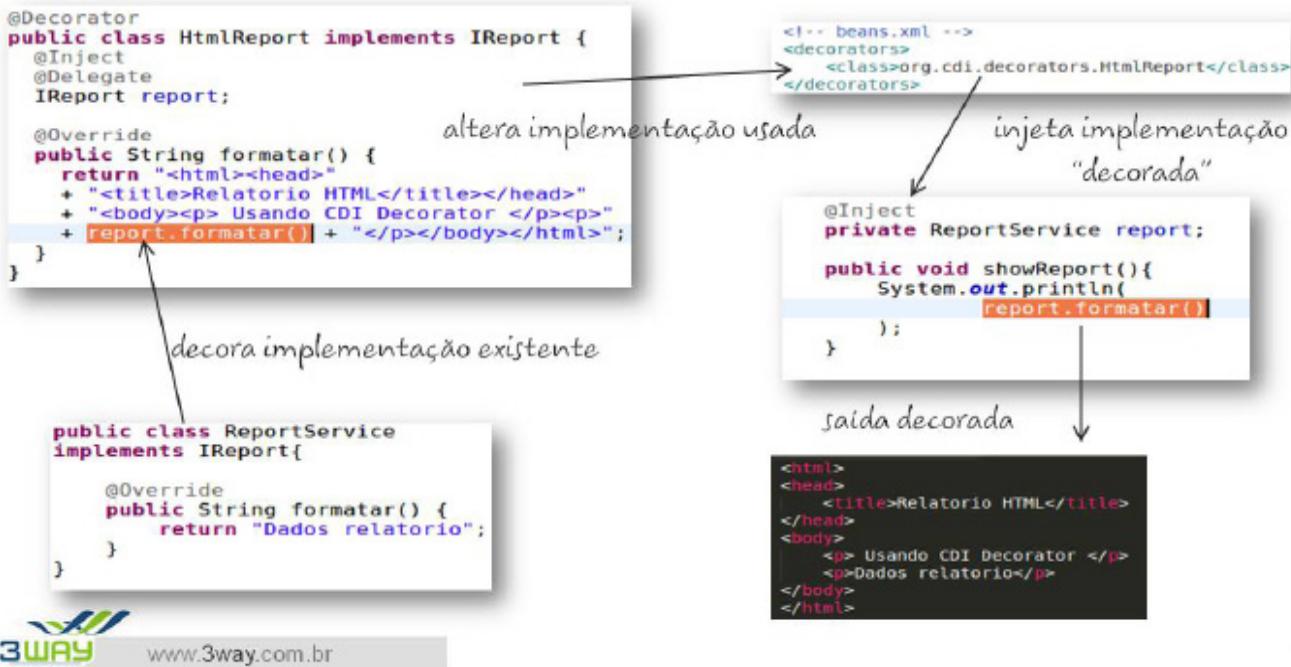
Uma classe “decoradora” deve ser marcada com a anotação `@Decorator`.

Na classe “decoradora” injetamos o objeto “legado”, a classe original . Esta será injetada na classe “decoradora” através de `@Inject @Delegate`.

No exemplo você pode observar a classe `ReportService` que implmenta o método `formatar()` da interface `IReport`. Um objeto `ReportService` é injetado noutra classe cliente e seu método `showReport()` invoca o método `formatar()` gerando uma saída sem formatação, simplesmente imprime uma String.

Então se pergunte como posso alterar a formatação apresentado por `showReport()` sem ter que mudar a classe `ReportService` ou a classe cliente? Com `@Decorator` você consegue!

## Decorators – formatando html



Aqui a classe decoradora, HtmlReport (@Decorator) formata o texto de ReportService, injetando uma instância deste através de @Inject @Delegate. Invocamos o método formatar( ) de ReportService e obtemos o texto original sem formatação. Sobreponemos o método formatar() enfeitando a String resultante com tags html.

Em seguida registramos em beans.xml a classe decoradora para que o CDI saiba que a classe cliente deve receber uma instância de HtmlReport no ponto injeção.

Assim você obtém uma comportamento diferente para sua aplicação sem ter que recompilar ou modificar as classes originais. Veja abaixo outra classe decoradora que formata o texto com tabulação ao invés de html, simplesmente alterando a classe no beans.xml

## Decorators – formatando texto



## CDI e JSF

Anotações de escopo do CDI javax.enterprise.context

```
import javax.enterprise.context.RequestScoped;  
  
@Named  
@RequestScoped  
public class LoginBean implements Serializable {
```

Anotações de escopo do JSF javax.faces.bean

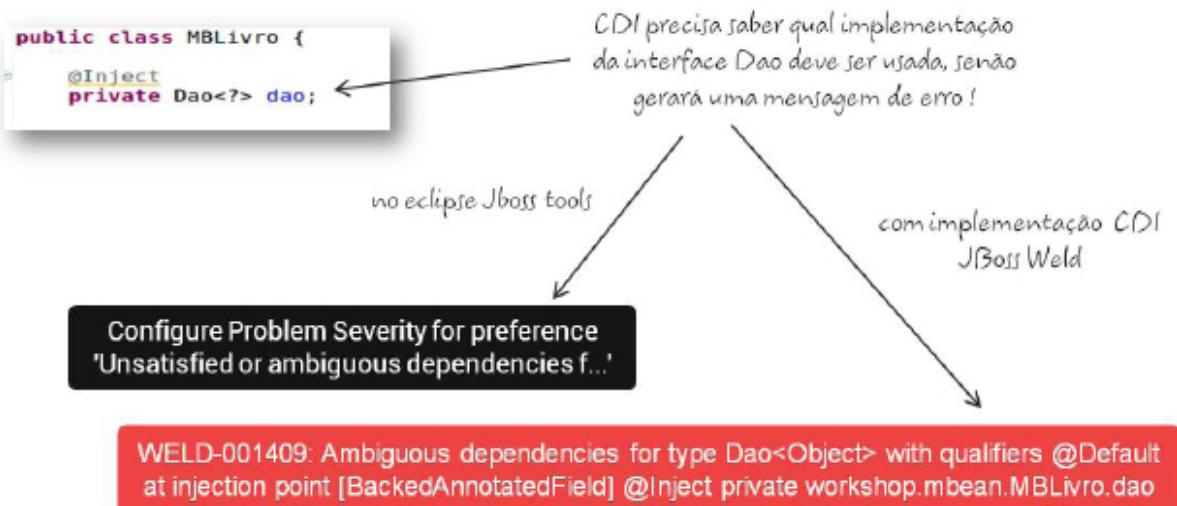
```
import javax.faces.bean.ViewScoped;  
  
@ManagedBean  
@ViewScoped  
public class HelloWorldBean implements Serializable {
```

Com JEE 7, JSF 2.2 e CDI 1.1, temos um conflito de nomes relação às anotações de escopo. As anotações de escopo do JSF se encontram no pacote javax.faces.bean e infelizmente possuem função e nome similar às anotações CDI no pacote javax.enterprise.context. As anotações javax.faces.bean não são geridas pelo CDI e não devem ser mais utilizadas, a própria documentação sugere que essas anotações poderão ser depreciadas no futuro.

Agora que você já sabe como usar métodos produtores, você poderá injetar vários objetos de contexto da API JSF com CDI. Como exemplo:

```
public class FacesProducers {  
  
    @Produces  
    private FacesContext context = FacesContext  
        .getCurrentInstance();  
  
    @Produces  
    private ExternalContext externalContext = FacesContext  
        .getCurrentInstance().getExternalContext();  
  
    @Produces  
    private Application application = FacesContext  
        .getCurrentInstance().getApplication();  
}
```

## Qualificadores - Desambiguidade



Um bean pode ser referenciado em uma expressão EL (expression language), tal como páginas JSF e JSP, mas é necessário definir um nome EL:

A anotação **@Named** especifica um nome EL para um managed bean CDI. Outro exemplo

```
@Named("livro")
```

```
public class HistoricoLivro implements Serializable{  
}
```

Nós agora poderemos acessar o bean CDI numa página JSF:

```
<h:outputText value="#{livro.isbn}" />
```

Se você quiser, não precisa definir um nome EL para o seu bean, ou seja, a anotação **@Named** aparece sem o atributo name. Nesse caso é usado o nome não qualificado da classe com os primeiros caracteres convertidos para minúsculas, no exemplo anterior teríamos nome da EL como historicoLivro:

```
<h:outputText value="#{historicoLivro.isbn}" />
```

De modo geral sua aplicação JSF pode ser convertida seguindo os passos:

1. Adicione um arquivo beans.xml no WEB-INF
2. Substitua @ManagedBean da JSF para @Named do CDI
3. Substitua @ManagedProperties da JSF para @Inject do CDI
4. Substitua anotações de escopo da JSF pelas anotações de escopo CDI.

## CDI e JPA/EJB

✓ Os beans do EJB são:

- Transacionais
- Remotos ou locais
- Podem passar Stateful beans, liberando recursos
- Podem fazer o uso de Timer
- Podem ser assíncronos

✓ Um bean anotado com `@Stateless` não precisa lidar com transações, o servidor irá cuidar disso.

✓ Caso queira utilizar JPA em um container como Tomcat, as transações deviam ser feitas manualmente, agora temos `@Transactional` na CDI.

✓ Beans EJB são beans CDI, portanto tem os mesmos benefícios, mas dizer o contrário não é válido.



Algo que você deve ter notado é que muitos recursos oferecidos em CDI já existiam em outras tecnologias JEE como ManagedBeans JSF e seus escopos, injeção de dependência com EJB, entre outros. O propósito da CDI é tornar esses recursos mais simples, mais integrados e flexíveis, porém há recursos como operações assíncronas, temporizadores, pools de serviços, transações que são essências para aplicações que exigem escalonamento e segurança, esses recursos só estão disponíveis com EJB.

Num container EJB bastaria anotar seu bean com `@Stateless` ou `@Statefull` para que todos os métodos dentro do bean sejam transacionais, mas se você não usa um container EJB como Jetty ou Tomcat você teria que lidar com essas transações manualmente:

```
public class Bean {  
    @Inject  
    private EntityManager em;  
    public void salvar(Entidade entidade) {  
        EntityTransaction tx = em.getTransaction();  
        tx.begin();  
        em.persist(entidade);  
        tx.commit();  
    }  
}
```

A partir de Java EE 7, EJBs não são únicos que tem natureza transacional. A nova anotação `@Transactional` te permite usar transações declarativas com beans CDI simplesmente anotando o Bean ou um método com a mesma:

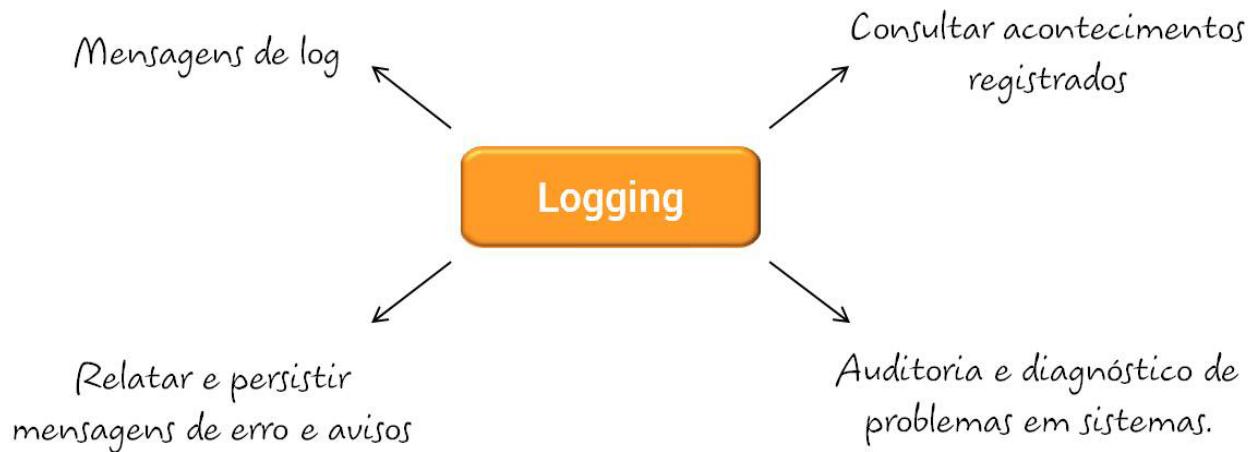
```
@javax.transaction.Transactional
public class Bean {
    @Inject
    private EntityManager em;
    public void salvar(Entidade entidade) {
        em.persist(entidade);
    }
}
```

ou aplicar em único método tornando-o transacional:

```
public class Bean {
    @Inject
    private EntityManager em;
    @Transactional
    public void salvar(Entidade entidade) {
        em.persist(entidade);
    }
}
```

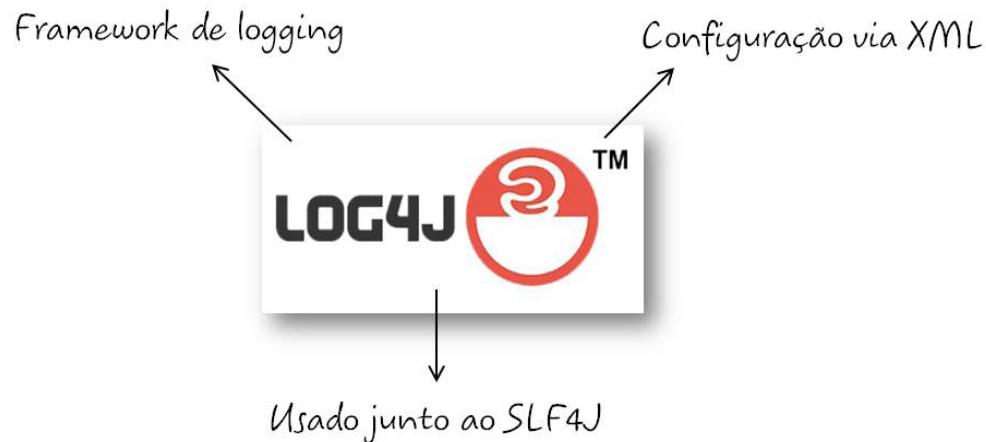
Como está vendo o JEE é uma coleção de especificações, temos EJB, JPA, JSF, JMS, JTA, Beans Validation, entre outros tantos, um número de coisas que podem causar estranheza para que está inciando no mundo Java. Mas no meio disso tudo, você agora poderá utilizar a CDI para expor esses componentes, em sua aplicação, sem muito esforço, ou quase.

## Logging



Logging se refere ao registro de eventos relevantes num sistema. Esses registros podem ser utilizados para restabelecer o estado original de um sistema ou para que um administrador conheça o seu comportamento no passado. Um arquivo de log pode ser utilizado para auditoria e diagnóstico de problemas em sistemas. Vários frameworks padronizam o processo de logging na plataforma Java.

## Log4J



O log4j é um framework de logging simples e flexível. Com o log4j é possível habilitar o registro de logs em tempo de execução sem modificar os binários da aplicação. O comportamento de logging pode ser controlado via edição do arquivo de configuração xml.

## Dependência do Log4j

Adicione a seguinte dependência em seu *pom.xml*

```
<dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.17</version>
</dependency>
```

Para usar o log4j na sua aplicação, devemos adicionar a seguinte dependência em seu pom.xml:

```
<dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.17</version>
</dependency>
```

## log4j.xml Exemplo de configuração

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration debug="true"
    xmlns:log4j='http://jakarta.apache.org/log4j/'>

    <appender name="console" class="org.apache.log4j.ConsoleAppender">
        <layout class="org.apache.log4j.PatternLayout">
            <param name="ConversionPattern" value="%d{yyyy-MM-dd HH:mm:ss} %-5p %c{1} - %m%n" />
        </layout>
    </appender>

    <root>
        <level value="debug" /> ← Nível de log
        <appender-ref ref="console" />
    </root>

</log4j:configuration>
```

Appender do  
console

Neste caso temos um *appender* que registrará o *log* no console

Além de adicionar a dependência, devemos configurar um arquivo xml para que os logs sejam registrados

## Usando um objeto Logger

```

import org.apache.log4j.Logger;
public class ExemploLogger {
    final static Logger logger = Logger.getLogger(ExemploLogger.class);
    public static void main(String[] args) {
        logger.info("Inicio da aplicação");
        System.out.println("O valor da soma é "+soma(5, 6));
        logger.info("Fim da aplicação");
    }
    public static int soma(int valor1, int valor2) {
        int soma;
        soma = valor1 + valor2;
        if (soma > 10) {
            logger.warn("O valor da soma ultrapassou 10! Soma = " + soma);
        }
        return soma;
    }
}

```

Método getLogger(class)

```

2016-01-19 14:21:00 INFO  ExemploLogger:9 - Início da aplicação
2016-01-19 14:21:00 WARN   ExemploLogger:23 - O valor da soma ultrapassou 10! Soma = 11
O valor da soma é 11
2016-01-19 14:21:00 INFO  ExemploLogger:13 - Fim da aplicação

```

Saída no console

Veja no exemplo acima como podemos utilizar um Logger e como fica saída no console de acordo com a configuração do log4j.xml visto anteriormente. Lembre-se de importar o pacote correto: org.apache.log4j.Logger

Veja que para criar um registro, devemos criar uma instância de Logger, obtida através do método getLogger():

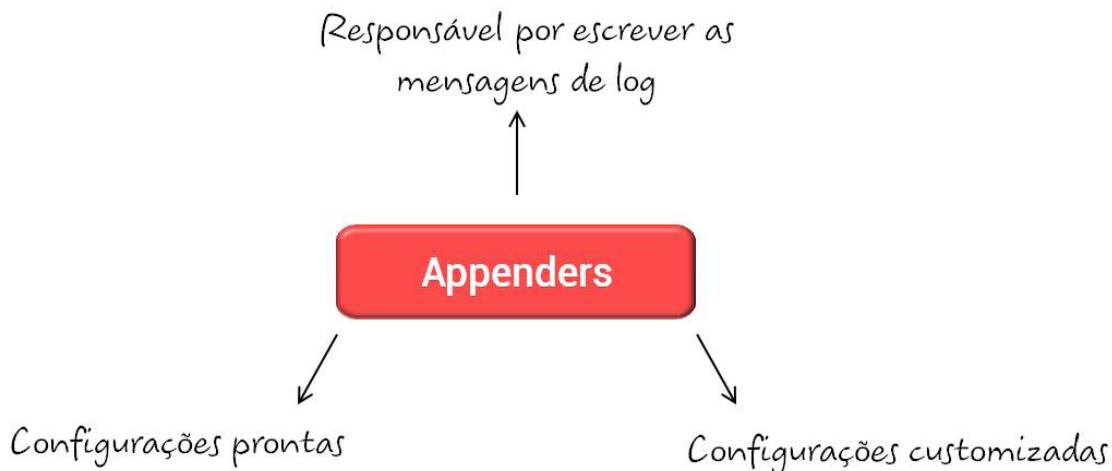
```

Logger logger = Logger.getLogger(MinhaClasse.class);
logger.info("Log nível de informação");
logger.warn("Log nível de aviso");
logger.error("Log nível de erro");
logger.fatal("Log nível fatal");
logger.debug("Log nível debug");
logger.trace("Log nível trace");

```

Os níveis de log serão melhor explicados logo adiante.

## Appenders



Appender são responsáveis por entregar eventos de log ao seu destino de registro.

Log4j permite que requisições de log sejam registradas em múltiplos destinos (console, arquivos, banco de dados).

Exemplo de appender que registra em um arquivo:

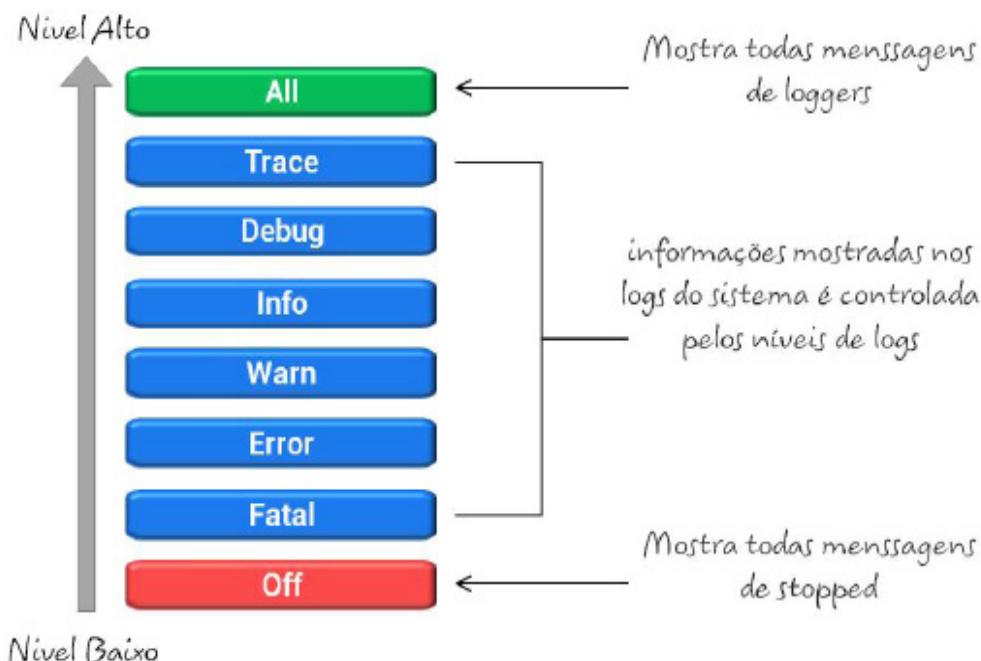
```
<appender name="default.file" class="org.apache.log4j.FileAppender">
    <param name="file" value="/log/logfile.log" />
    <param name="append" value="false" />
    <param name="threshold" value="debug" />
    <layout class="org.apache.log4j.PatternLayout">
        <param name="ConversionPattern"
              value="%d{ISO8601} %-5p [%c{1}] - %m%n" />
    </layout>
</appender>
```

O padrão de conversão (conversion pattern) está relacionado com a função de printf em C. Um padrão de conversão é composto de literais e expressões de controle de formatação, chamados de especificadores de conversão. Cada especificador de conversão é precedido pelo sinal de porcentagem (%) e seguido de um caractere de conversão.

No exemplo utilizamos o caractere “c”, que é usado para mostrar a categoria do evento de logging e “p”, usado para mostrar a prioridade do evento de logging. Para saber todos os caracteres de conversão, acesse o link para ver a documentação: <http://logging.apache.org/log4j/1.2/apidocs/org/apache/log4j/PatternLayout.html>

Veja que no exemplo acima também devemos indicar o caminho para o arquivo onde os logs serão registrados (param="file" value="caminho/do/arquivo").

## Nível de Log



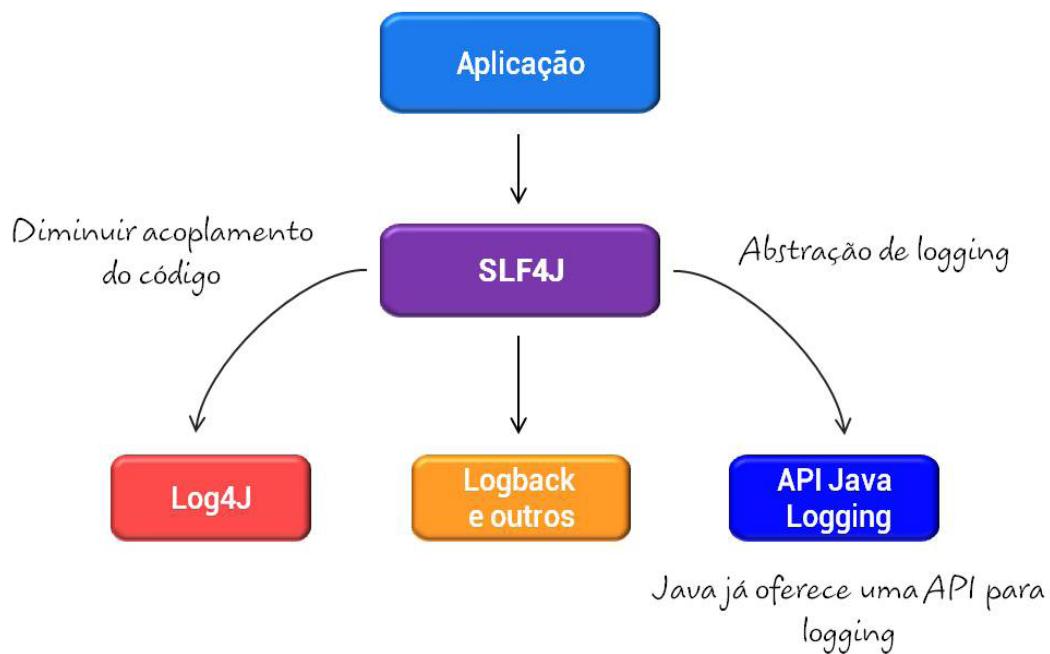
A quantidade e tipo de informações mostradas nos logs do sistema é controlada pelos níveis de logs do log4j. O log4j possui os seguintes níveis de log:

Nível	Valor int
OFF	0
FATAL	100
ERROR	200
WARN	300
INFO	400
DEBUG	500
TRACE	600
ALL	Ineger.MAX_VALUE

- **OFF** – O logging está desativado.
- **FATAL** – Indica que algo muito ruim aconteceu, podendo ser a parada da aplicação.
- **ERROR** – Indica que algo deu errado e deve ser corrigido o mais rápido possível. Significa que a aplicação entrou em um estado indesejado.
- **WARN** – Indica que algo pode dar errado na aplicação. A aplicação pode tolerar mensagens de aviso, mas elas devem ser examinadas para evitar que alguma coisa dê erra-

- **INFO** – Indica que algum processo relevante terminou da forma esperada. Os administradores de sistema monitoram os logs de info para assegurar o que está acontecendo no sistema e se há algum problema no fluxo normal.
- **DEBUG** – Toda e qualquer mensagem de log irá ser registrada no arquivo de log se esse nível de log estiver ativado.
- **TRACE** – Adiciona mais informações aos logs de nível DEBUG.
- **ALL** – Ativas todos os níveis de log.

## SLF4J



Simple Logging Facade for Java (SLF4J) serve como uma simples fachada ou abstração para vários frameworks de logging (ex.:logback, log4j) permitindo o usuário final usar o framework de log desejado em tempo de implantação. Com o SLF4J, o acoplamento do código pode ser diminuído, já que nosso código não dependerá de um framework específico para fazer as tarefas de logging. O log4j não substitui os frameworks de logging, eles trabalham juntos.

## Dependência do SLF4J com Log4J

```
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-log4j12</artifactId>
    <version>1.7.13</version>
</dependency>
```

## Usando um objeto Logger com SLF4J

```
Logger logger =
LoggerFactory.getLogger(MinhaClasse.class);

logger.info("Informação de que algo aconteceu");
```

A idéia é utilizarmos o SLF4J junto ao Log4J. Para isso, o SLF4J já possui um dependência que adiciona a biblioteca do Log4J ao baixar a biblioteca do SLF4J:

```
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-log4j12</artifactId>
    <version>1.7.13</version>
</dependency>
```

O que muda agora é como vamos obter uma instância de Logger. Essa instância agora deve ser obtida através de LoggerFactory:

```
Logger logger = LoggerFactory.getLogger(MinhaClasse.class);
logger.info("Informação de que algo aconteceu");
```

A classe LoggerFactory é uma classe de utilidade, produzindo objetos Logger para várias APIs de logging, mais notavelmente para a log4j, logback e JDK 1.4 logging.

Além dos frameworks de logging, Java já oferece uma API para fazer essa tarefa. O pacote java.util.Logging oferece as mesmas funcionalidades desses frameworks específicos.