03

Configurando o CDI

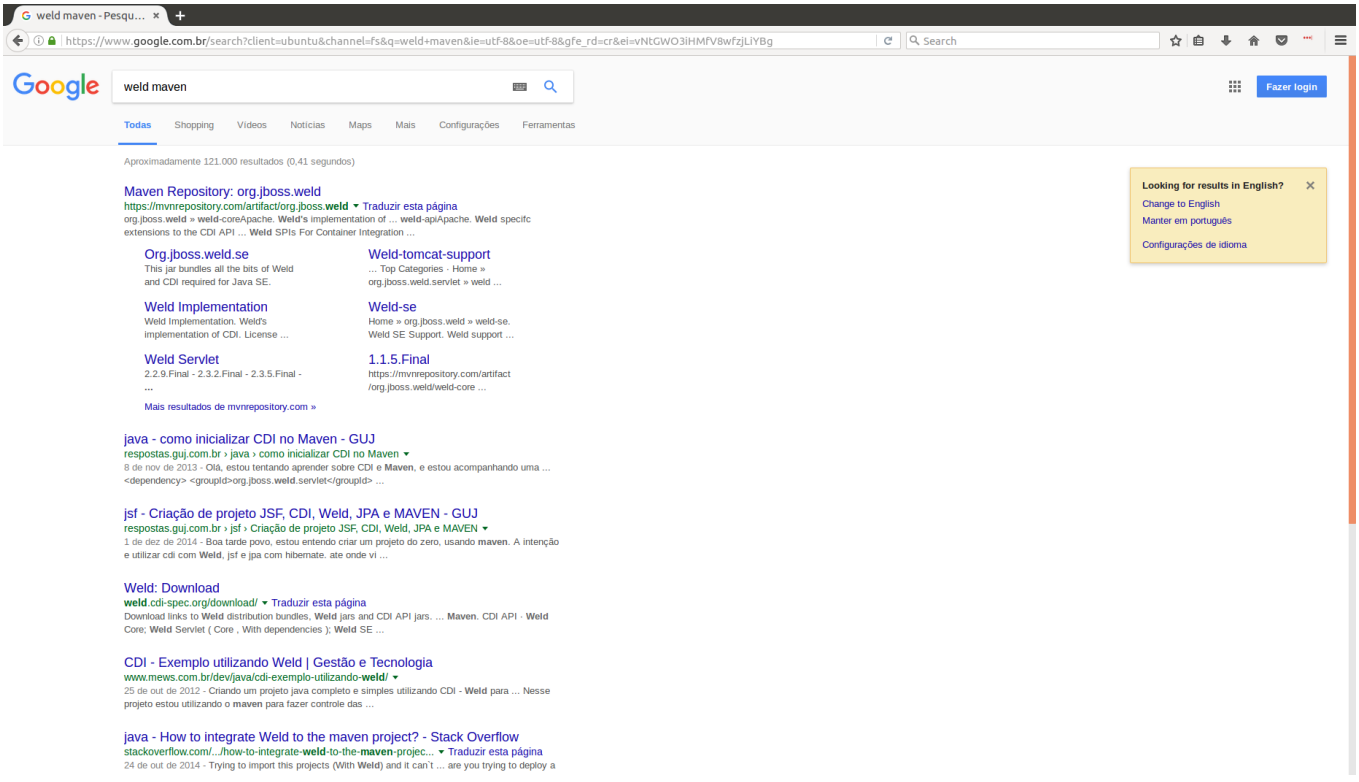
Transcrição

Antes de começarmos a falar sobre as configurações, vamos deixar claro que as alterações feitas na aula anterior foram desfeitas para essa aula.

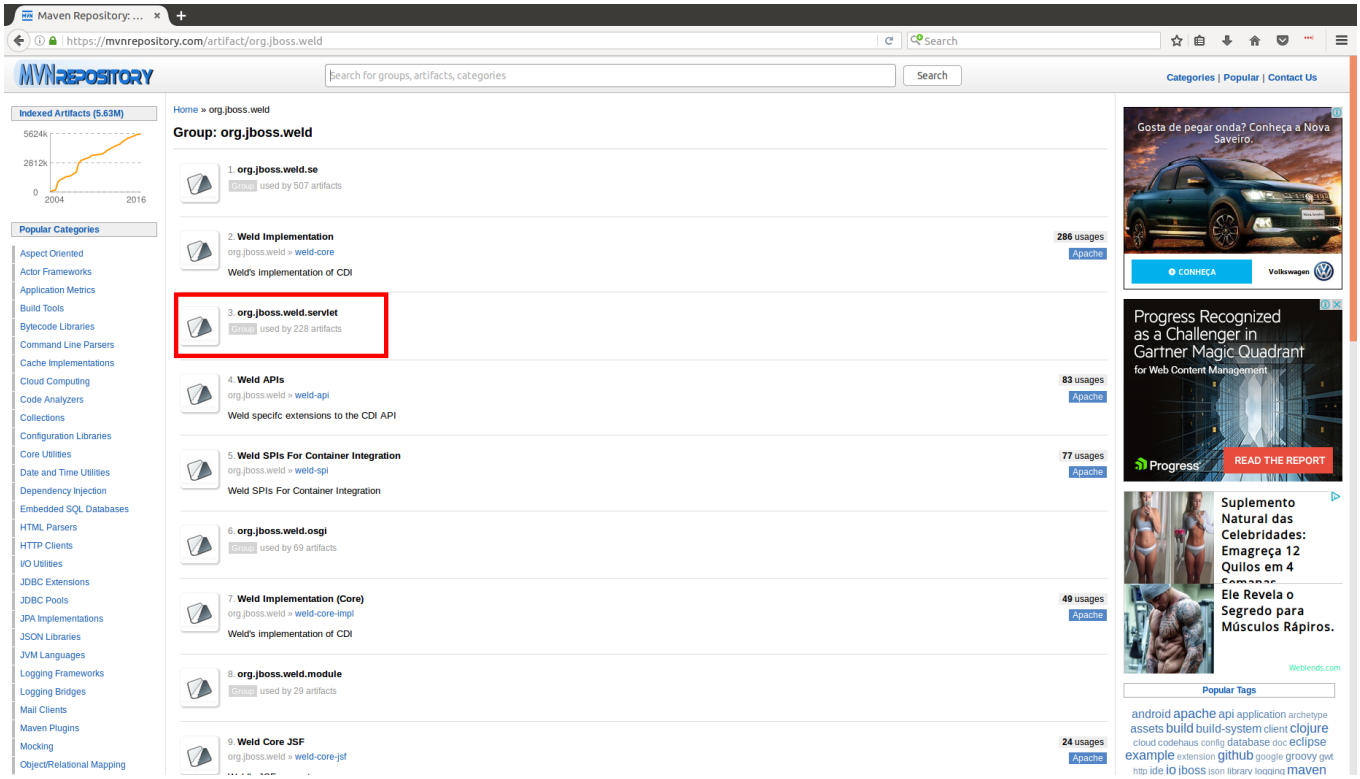
Agora que já vimos a ideia por trás do CDI, vamos ver como configurá-lo em nosso projeto. Vamos declarar a dependência do CDI no arquivo `pom.xml` e um `jar` será baixado e inserido dentro do nosso projeto.

O CDI é uma especificação, e por isso precisamos de alguém que implemente essa especificação para que seja possível utilizá-lo. A implementação de referência do CDI é o Weld. Portanto, vamos procurar a dependência do Weld para o Maven e assim adicionarmos no arquivo `pom.xml` .

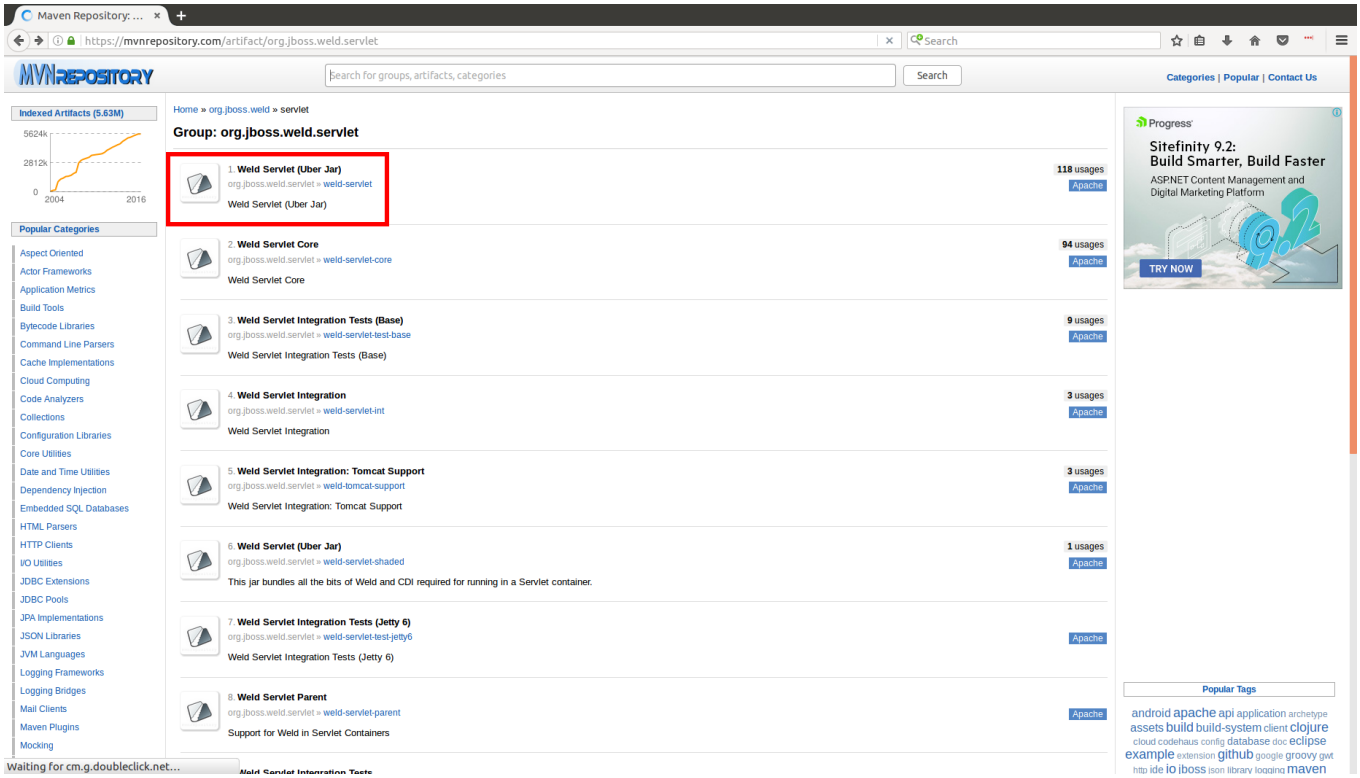
Para isso, podemos fazer uma simples busca no Google, pesquisando por "weld maven". O primeiro link da pesquisa deve ser do site www.mvnrepository.com (www.mvnrepository.com). Ao clicarmos no link, veremos algumas dependências disponíveis.



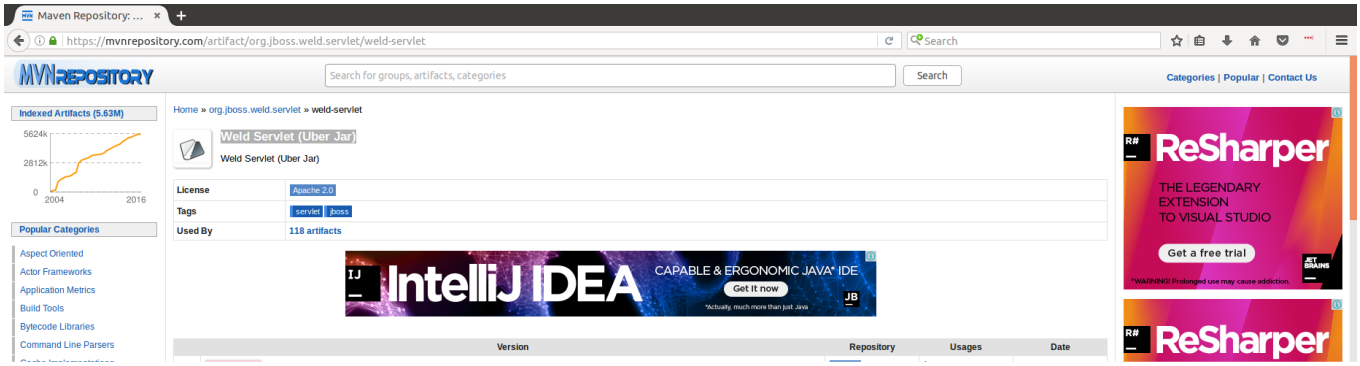
O Weld pode funcionar tanto em ambiente Web (Java EE)) como em ambiente "não Web", como Desktop (Java SE). No nosso caso, como se trata de um ambiente Web, utilizaremos a dependência do `org.jboss.weld.servlet`



Ao acessar o `org.jboss.weld.servlet`, podemos ver algumas formas de baixar o Weld. Aqui vamos utilizar o `Weld Servlet (Uber Jar)`. Um Uber Jar é um *jar* mais "inchado", em que o Weld colocou dentro desse *jar* todas as suas outras dependências. Desta forma, baixamos um único *jar* e já temos tudo que é preciso para rodar.

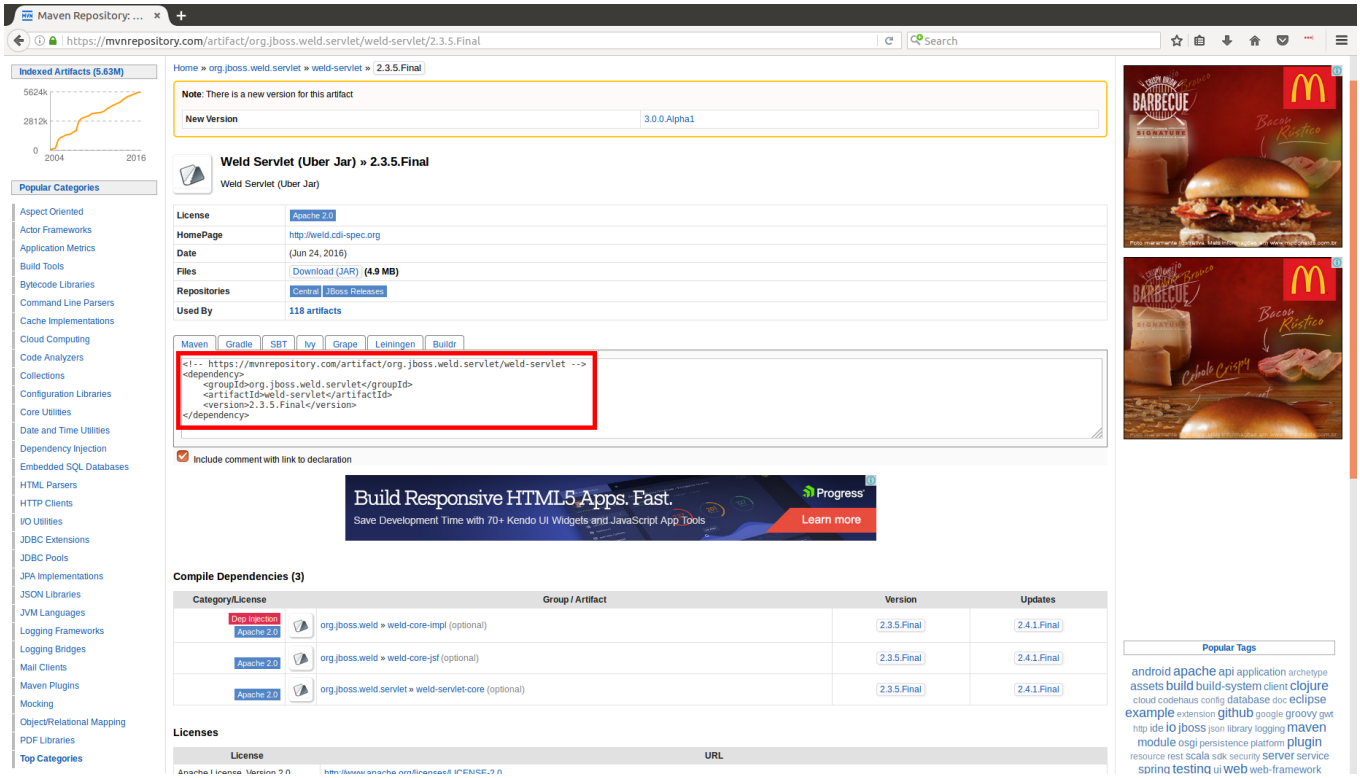


Após clicar em `Weld Servlet (Uber Jar)`, vamos ver uma página com várias versões dessa dependência. A versão que iremos utilizar neste curso será a `2.3.5.Final` - a última versão estável no momento que o curso foi gravado. Vamos clicar no link da versão `2.3.5.Final` e veremos o conteúdo que devemos adicionar no arquivo `pom.xml`.



Vamos adicionar o conteúdo referente à dependência no arquivo `pom.xml`, dentro da tag `dependencies`.

Pode ser que demore um pouco para baixar a dependência, caso ela já não esteja em seu repositório local.




```
<dependencies>
    <!-- outras dependencias -->

    <dependency>
        <groupId>org.primefaces</groupId>
        <artifactId>primefaces</artifactId>
        <version>5.3</version>
```

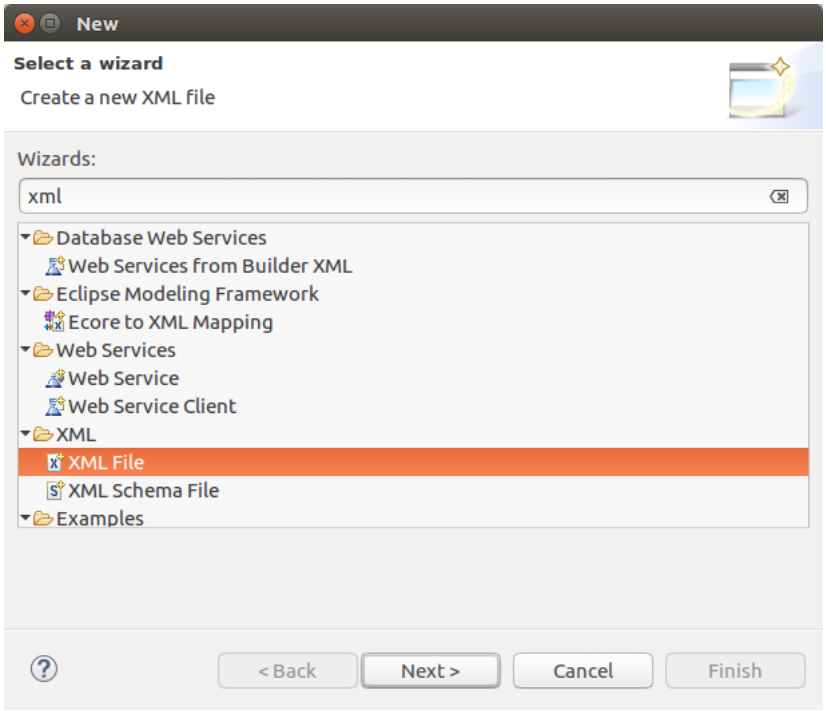
```
        </dependency>

        <dependency>
            <groupId>org.jboss.weld.servlet</groupId>
            <artifactId>weld-servlet</artifactId>
            <version>2.3.5.Final</version>
        </dependency>
    </dependencies>
```

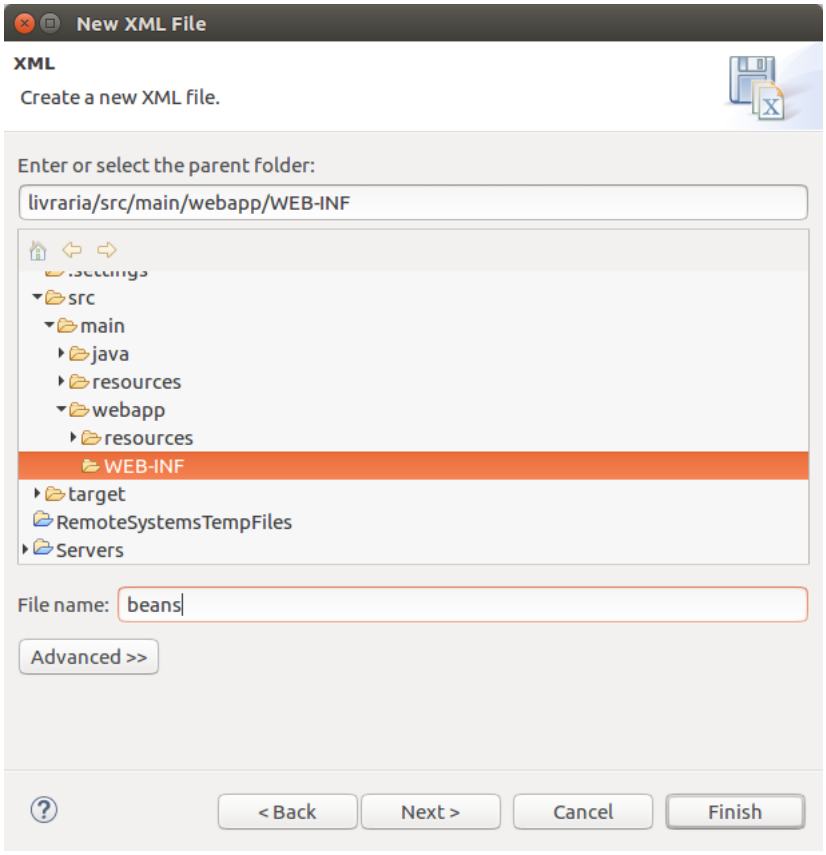
O arquivo `beans.xml`

Após a conclusão do Download da dependência do Weld, temos que realizar algumas configurações. A especificação do CDI pede que tenhamos um arquivo chamado `beans.xml` dentro do diretório `WEB-INF`.

Dentro de `Deplyed Resources > webapp > WEB-INF` será criado um novo arquivo `.xml`. Vamos selecionar a pasta `WEB-INF` e utilizar o atalho "Ctrl + N" do Eclipse. Na tela que irá aparecer, basta digitar "xml" no campo de busca.



O nome do arquivo deve ser `beans.xml`. Feito isso, vamos clicar em "Finish".



Este é o arquivo de configuração para que seja possível rodar o CDI. Vamos abrir o arquivo e clicar na tab "Source", do Eclipse, para que seja possível ver o código do arquivo. O arquivo deve ter o seguinte conteúdo:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/beans
     version="1.2" bean-discovery-mode="all">

</beans>
```

Temos a tag `<beans>` no `beans.xml`, onde são definidos os *namespaces* que serão necessários. Temos também o trecho `version="1.2"` que indica a versão do CDI utilizada. Encontraremos ainda o `bean-discovery-mode="all"`, que se não for declarado, não terá o valor `all` por padrão. O `discovery-mode` é a estratégia que o CDI utilizará para encontrar as nossas classes e saber se elas podem ou não ser injetadas. Existem três estratégias:

- `annoted` : Somente as classes que forem anotadas serão candidatas a serem injetadas
- `all` (todas)
- `none` (nenhuma)

O valor padrão do `bean-discovery-mode` é `annoted`.

Configurações do Servlet Container

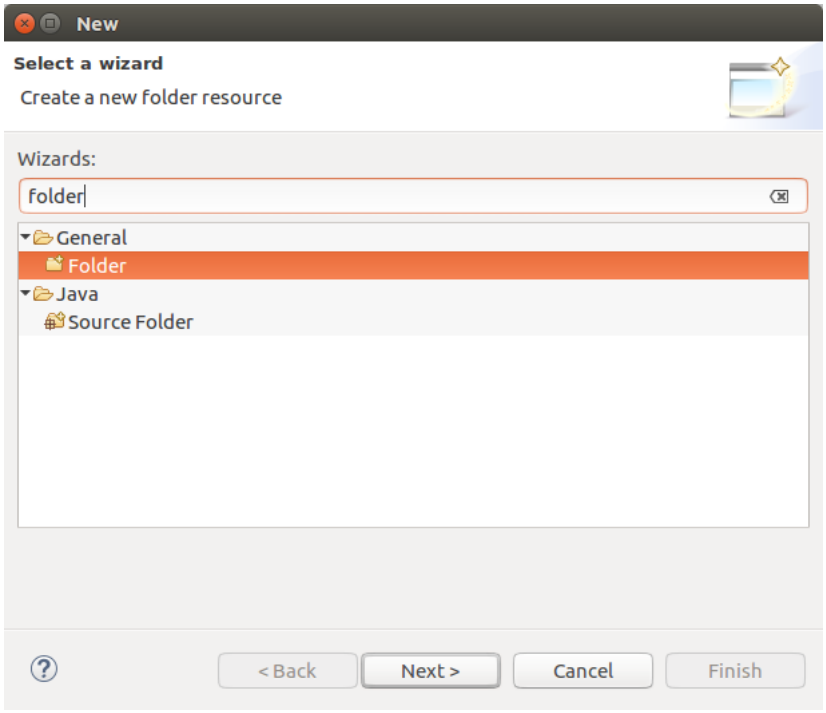
Essa é a única configuração que é solicitada pela especificação para que comecemos a utilizar o CDI dentro do nosso projeto. Mas isso ocorre apenas em um projeto Java EE, quando estamos utilizando um servidor de aplicação. Nem mesmo seria necessária a dependência no `pom.xml` porque o servidor já possui uma implementação do CDI.

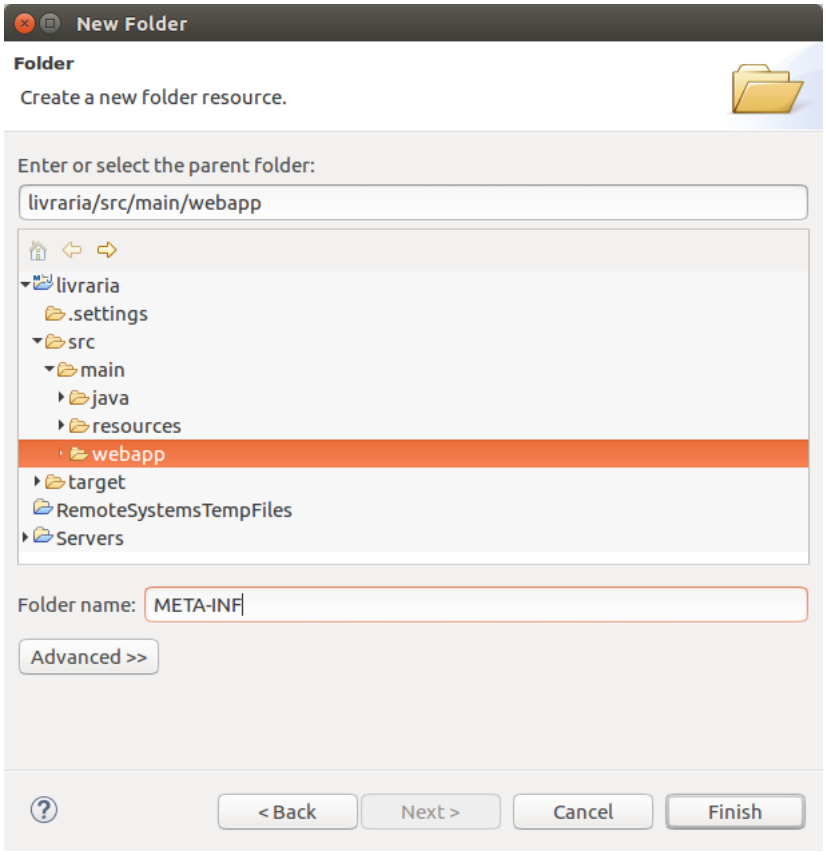
Mas como estamos utilizando um *Servlet Container*, precisaremos ensinar para o Tomcat nos casos em que ele cria um objeto que o CDI precisará ou quando deve ser iniciado o CDI, por exemplo. Então vamos fazer essas configurações.

Para o CDI funcionar, é preciso que exista um objeto do tipo `BeanManager`. Este irá abrir um contexto, localizar uma dependência (candidato a ser injetado), disparar um evento e diversas outras coisas. É um objeto muito importante para o CDI. O CDI pede na especificação que esse objeto esteja acessível via JNDI, que é uma outra especificação.

Na JNDI, temos um nome e é a partir dele, que ela irá pegar no `container`, e depois, será devolvido o objeto. Então o que vamos ensinar agora para o Tomcat é como criar o objeto. Em seguida, no arquivo `web.xml`, vamos indicar que o nome `BeanManager` está associado àquele tipo específico. Quando alguém pedir por um `BeanManager`, será devolvido um objeto desse tipo.

O primeiro passo é selecionar o diretório `webapp` e criar um outro diretório chamado `META-INF`. Após selecionar o diretório vamos utilizar o atalho "Ctrl + N" e pesquisar por folder na tela mostrada.





Dentro do diretório `META-INF`, vamos criar agora o arquivo de configuração do Tomcat chamado `context.xml`.

Dentro deste arquivo, vamos declarar que dado um tipo, qual será o objeto utilizado para gerar esse tipo. Então se precisamos de um `BeanManager`, vamos declarar quem cria esse `BeanManager` para nós. O arquivo `context.xml` ficará com o seguinte conteúdo:

```
<Context>
    <Resource name="BeanManager"
       auth="Container"
       type="javax.enterprise.inject.spi.BeanManager"
       factory="org.jboss.weld.resources.ManagerObjectFactory"/>
</Context>
```

Temos um recurso (`Resource`) chamado `BeanManager` e sempre que for preciso um objeto do tipo `javax.enterprise.inject.spi.BeanManager` será utilizado o `org.jboss.weld.resources.ManagerObjectFactory` para criar esse tipo.

Vamos salvar o arquivo e em seguida realizar a configuração no arquivo `web.xml`. Vamos criar um *listener*, alguém que irá ficar ouvindo quando subir o contexto - e também será o responsável por subir o CDI.

Antes do fechamento da tag `</web-app>`, vamos declarar o *listener* e em seguida que `BeanManager` estará associado ao tipo que configuramos no `context.xml`. O arquivo ficará com o seguinte conteúdo:

```
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:web="http://java.sun.com/xml/ns/javaee"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
    version="3.0">
```

```
    <!-- restante do conteudo -->


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

    <resource-env-ref>
        <resource-env-ref-name>BeanManager</resource-env-ref-name>
        <resource-env-ref-type>
            javax.enterprise.inject.spi.BeanManager
        </resource-env-ref-type>
    </resource-env-ref>
</web-app>
```

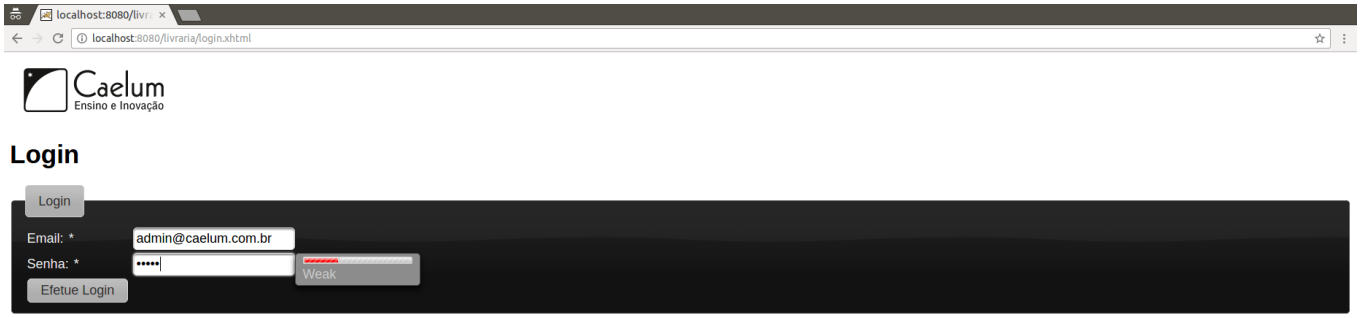
Na tag `<resource-env-ref>`, estamos indicando que quando precisarmos de um recurso com o nome `BeanManager`, vamos devolver um objeto do tipo `javax.enterprise.inject.spi.BeanManager`. E lá no arquivo `context.xml` definimos que quem cria um objeto do tipo `javax.enterprise.inject.spi.BeanManager` é o `org.jboss.weld.resources.ManagerObjectFactory`.

Após fazer essas alterações já é possível subir a nossa aplicação. Vamos iniciar o Tomcat e acompanhar o console.Ao acessar a aba `Console` do Eclipse, é possível ver a mensagens indicando que o Weld foi inicializado com sucesso:

```
...
INFO: WELD-ENV-001008: Initialize Weld using ServletContainerInitializer
dez 08, 2016 10:00:56 AM org.jboss.weld.bootstrap.WeldStartup <clinit>
INFO: WELD-000900: 2.3.5 (Final)
dez 08, 2016 10:00:57 AM org.jboss.weld.bootstrap.WeldStartup startContainer
INFO: WELD-000101: Transactional services not available. Injection of @Inject UserTransaction not av
dez 08, 2016 10:00:58 AM org.jboss.weld.environment.tomcat.TomcatContainer initialize
INFO: WELD-ENV-001100: Tomcat 7+ detected, CDI injection will be available in Servlets, Filters and
dez 08, 2016 10:00:58 AM org.jboss.weld.environment.servlet.Listener contextInitialized
INFO: WELD-ENV-001006: org.jboss.weld.environment.servlet.EnhancedListener used for ServletContext r
dez 08, 2016 10:00:58 AM com.sun.faces.config.ConfigureListener contextInitialized
...
```

Utilizando o CDI

Agora vamos acessar http://localhost:8080/livraria/login.xhtml (http://localhost:8080/livraria/login.xhtml) e realizar o login. O login vai verificar se existe um usuário no banco de dados com a senha que foi fornecida. Caso existe seremos direcionados para a página de livros.



Agora que verificamos que tudo continua funcionando, vamos começar a utilizar o CDI dentro do nosso projeto.

Quem está criando os nossos beans é o JSF, utilizando a anotação `@ManagedBean`. Por exemplo:

```
@ManagedBean
public class AutorBean implements Serializable {
```

Por esse motivo, não vamos conseguir injetar dependências, pois o JSF não sabe fazer essa tarefa. Precisamos que o CDI crie os nossos *beans* e que, a partir dele, comece a injetar as dependências.

Quando definimos a anotação `@ManagedBean` em uma classe, o *bean* estará acessível via *Expression Language*(EL) com o nome da classe com a primeiro letra minúscula (Ex.: `AutorBean` ficará disponível como `autorBean`).

Se olharmos dentro de `autor.xhtml`, temos o seguinte trecho de código, nas primeiras linhas do arquivo:

```
<f:viewParam name="autorId" value="#{autorBean.autorId}" />
```

Temos o nome da classe com a primeira letra em minúsculo. Precisamos trocar a anotação para que o CDI crie o objeto, ao mesmo tempo que é necessário que continue sendo possível acessar o *bean* na *view* via *Expression Language*(EL). Outro ponto é quando definimos um `@ManagedBean`, o escopo padrão dele será de *request*.

Para deixar objeto acessível por meio de `EL` ao mesmo tempo em que o objeto é criado pelo CDI, utilizamos a anotação `@Named`.

```
@Named
public class AutorBean implements Serializable {
```

Porém o escopo padrão do CDI não é *request*. O escopo padrão é chamado `@Dependent`. O `@Dependent` é um escopo que irá depender de quem criou. Alguém precisa definir o escopo para a classe.

Vamos utilizar como exemplo a classe `DAO`. No `DAO` não vamos ter um escopo explícito (então é o padrão `@Dependent`), então se for definido que no `AutorBean` que é necessário injetar um `DAO` e o `AutorBean` tiver um escopo de *request*, o `DAO` também terá um escopo de *request*. O `DAO` irá assumir o escopo de quem está solicitando a injeção dele.

No caso queremos manter o escopo de *request*. Por isso, vamos utilizar também a anotação `@RequestScoped`:

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

Tome cuidado com o *import*, pois temos duas opções. Escolha a opção do pacote `javax.enterprise.context`. Por fim, na classe, utilize o atalho `Ctrl + Shit + O` do Eclipse, para organizar os imports da classe.

Vamos fazer essa mesma alteração nas demais classes do sistema:

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

A classe `Tema` *bean* é um pouco diferente e possui escopo de sessão.

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

@ManagedBean
@SessionScoped
public class TemaBean implements Serializable {
```

Queremos manter a classe com escopo de sessão, e o CDI tem uma escopo para isso, que também se chamado `SessionScoped`. O mesmo nome da anotação do JSF. Por isso, vamos trocar o *import*:

```
import javax.enterprise.context.SessionScoped;
import javax.inject.Named;

@Named
@SessionScoped
public class TemaBean implements Serializable {
```

Vamos agora realizar as alterações no `LivroBean`. O `LivroBean` é um pouco diferente, pois possui um `ViewScoped`, que é um escopo pré-definido pelo JSF, e não temos esse mesmo escopo pré-definido pelo CDI.

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

Para que seja possível utilizar o `ViewScoped` no CDI, o JSF criou uma forma de integrar as duas especificações. Mais uma vez é necessário trocar o *import*: em vez de escolher o escopo vindo de `javax.faces.bean`, vamos utilizar o escopo vindo de `javax.faces.view`.

```
import javax.faces.view.ViewScoped;
import javax.inject.Named;

// outros imports

@Named
@ViewScoped
public class LivroBean implements Serializable {
```

No `VendasBean` vamos fazer as mesmas alterações realizadas no `LivroBean`:

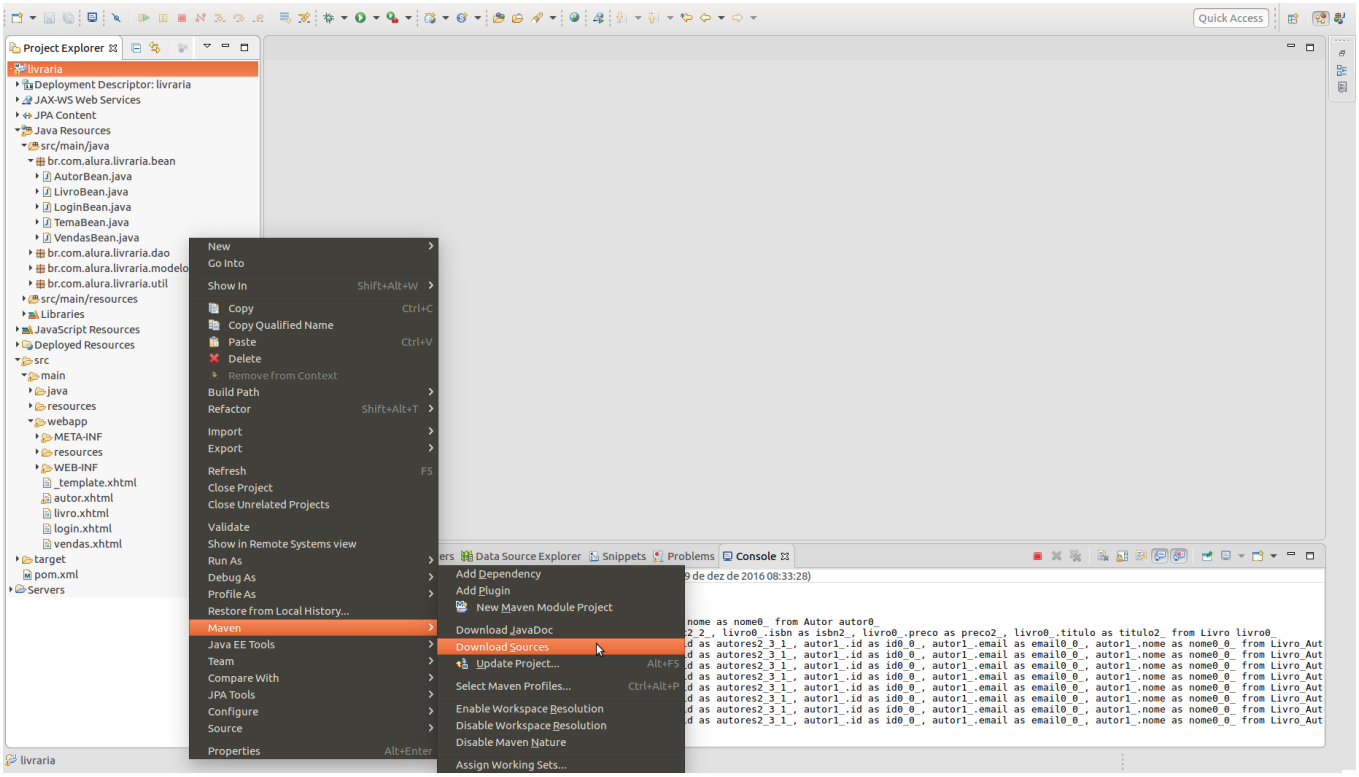
```
import javax.faces.view.ViewScoped;
import javax.inject.Named;

// outros imports

@Named
@ViewScoped
public class VendasBean implements Serializable {
```

Pronto! Conseguimos migrar nossos *beans* para serem gerenciados pelo CDI. Vamos subir nossa aplicação e verificar se tudo continua funcionando. Ao subir o servidor e atualizar a página de livros (http://localhost:8080/livraria/livro.xhtml (http://localhost:8080/livraria/livro.xhtml)), podemos ver que tudo está funcionando como esperado.

Vamos baixar o código-fonte das dependências do nosso projeto. Para isso clique com o botão direito no projeto e escolha a opção "Maven > Download Sources". Pode levar um tempo para baixar tudo.



Na classe `TemaBean`, segure a tecla "Ctrl" e clique em `@SessionScoped`. Dessa forma vamos navegar até o código da anotação. No código existe a seguinte linha de código:

```
@NormalScope(passivating = true)
```

Alguns escopos do CDI, geralmente escopos maiores que sessão, possuem o valor de `passivating = true`. No que isso implica? Por esse objeto ficar em memória por muito tempo, o CDI pode serializar o objeto no disco caso ele não esteja sendo utilizado. Quando a aplicação precisar novamente do objeto ele será devolvido para a memória principal. Dessa forma é possível diminuir o uso da memória principal

Para que seja possível fazer esse processo, é necessário que as nossas classes implementem a interface `Serializable`.

```
public class TemaBean implements Serializable {
```

Pronto! Conseguimos migrar nossa aplicação para utilizar as anotações relacionadas ao CDI em vez das anotações relacionadas ao JSF. Agora já é possível utilizar a injeção de dependências em nosso sistema.

Além dos escopos `@Dependent`, `@RequestScoped` e `@SessionScoped`, temos também o `@ApplicationScoped` e o `@ConversationScoped`.