03

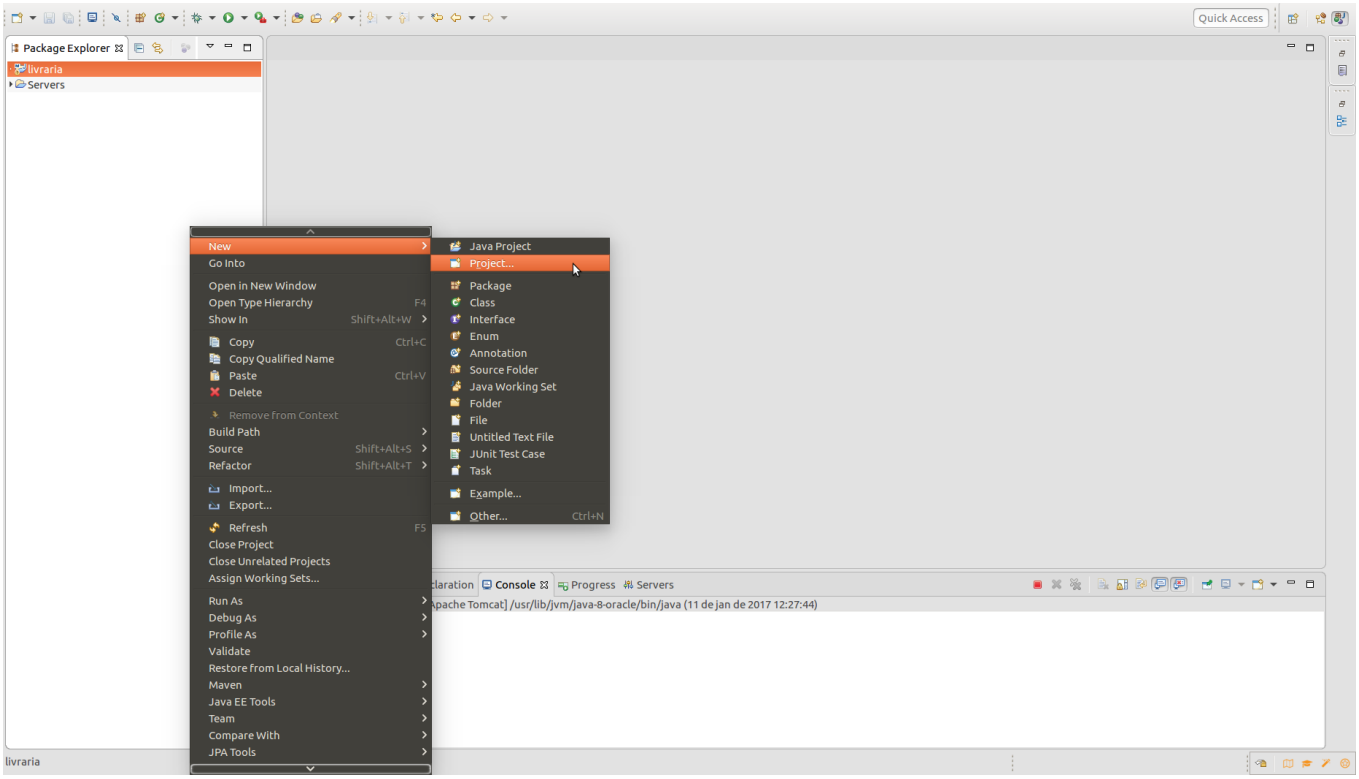
Criando lib

Transcrição

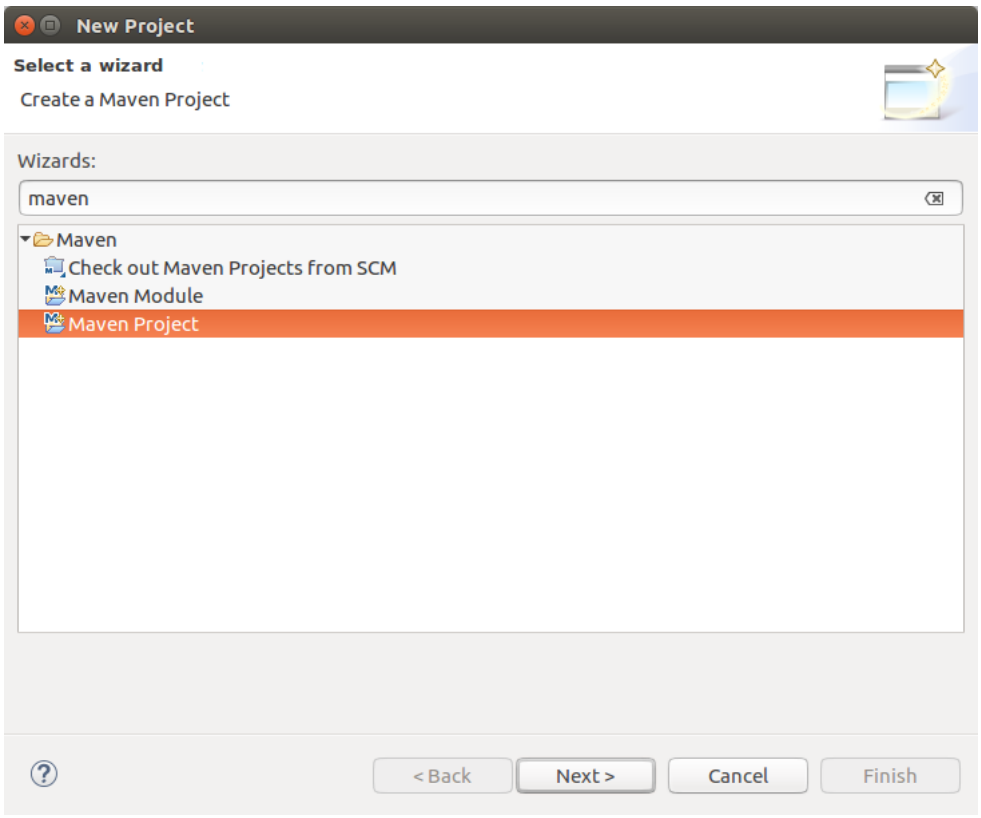
Essa ideia de termos um `DAO` genérico, ou um JPAUtil é uma ideia que podemos reutilizar em outros projetos. Se estivermos em um outro projeto Web que utilize o CDI e um Servlet Container podemos reutilizar o `DAOFactory` para injetar os DAOs. E o `JPAUtil` para pode injetar o `EntityManager`.

Pensando nessa questão, vamos extrair todas as classes que criamos para uma biblioteca (*lib*). Sempre que precisarmos tanto nesse projeto quanto em outros projetos, podemos adicionar o arquivo `.jar` no projeto e simplesmente utilizá-la.

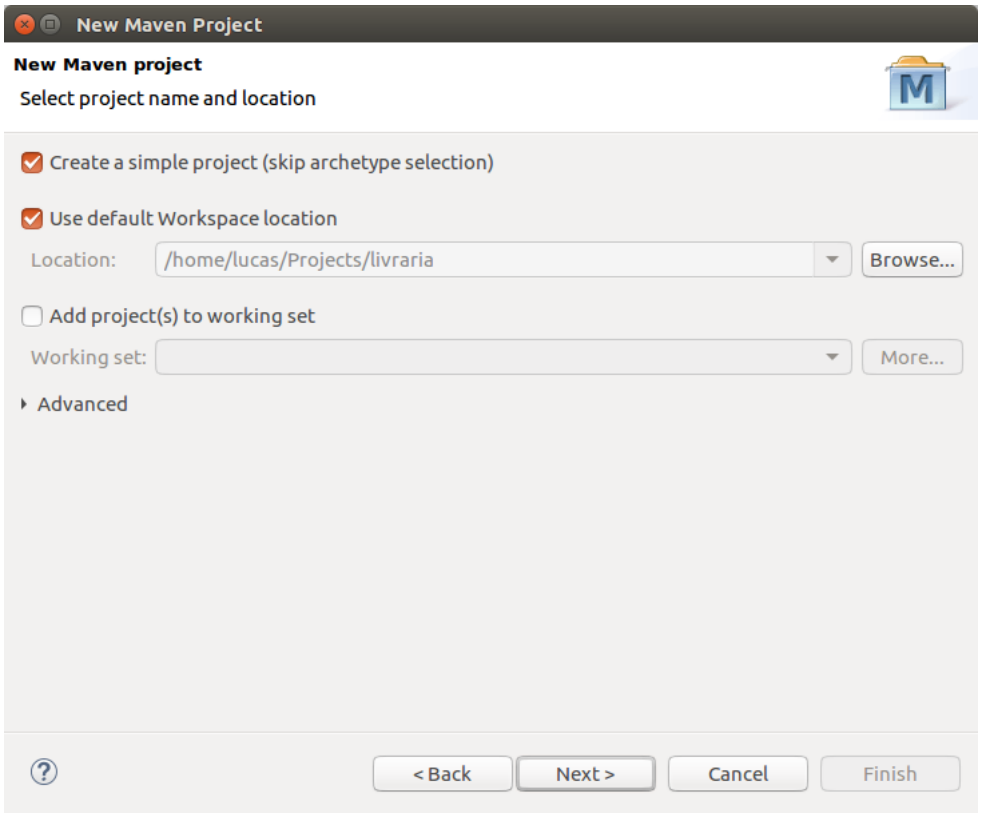
No Eclipse, vamos criar um novo projeto.



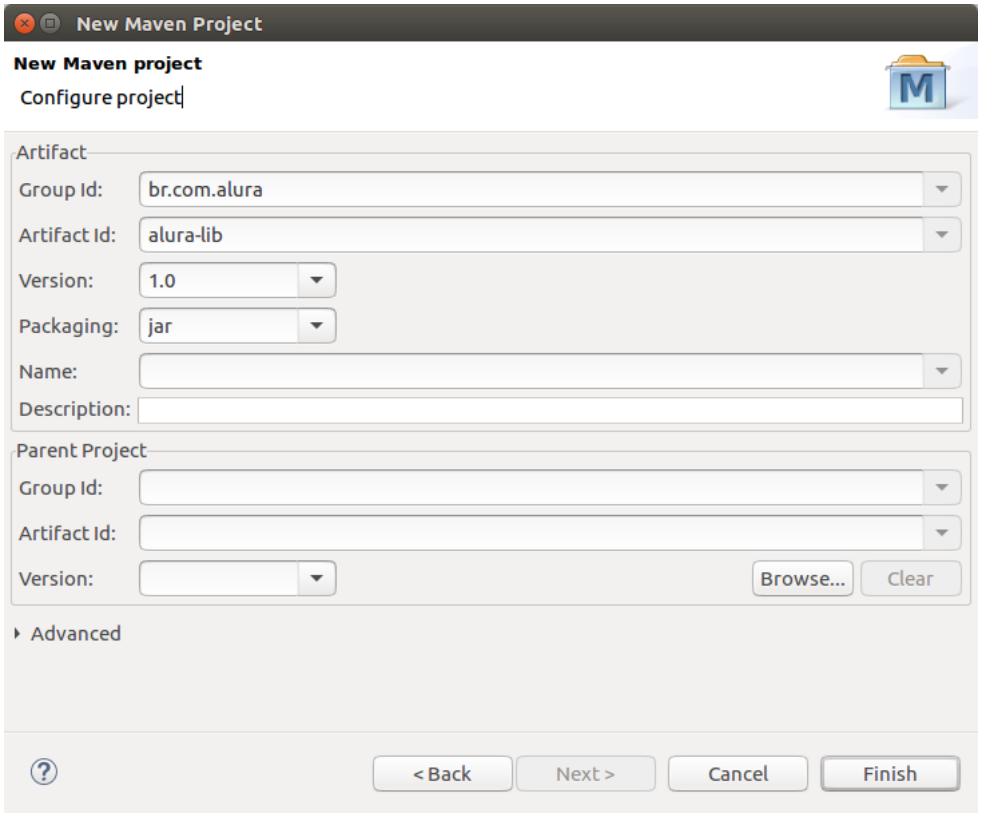
Vamos escolher Maven Project.



Na próxima tela, vamos marcar a opção "Create a simple project (skip archetype selection)".



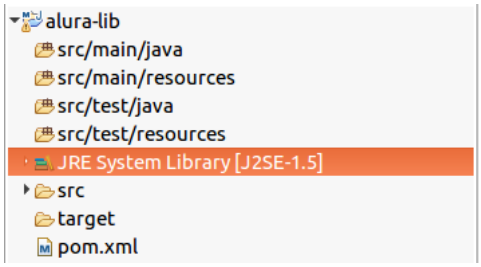
Vamos preencher as informações sobre o projeto e clicar em "finish".



Como o nosso projeto vai utilizar CDI e o `EntityManager` que é uma interface da JPA, vamos precisar dessa parte do Java EE. Na especificação são definidas várias interfaces e quem desejar pode implementar. Um exemplo disso é o Weld que faz a implementação do CDI, ou o EclipseLink, implementação da JPA.

A ideia é utilizar sempre essas interfaces, pois isso deixará nosso projeto flexível e funcional independente da implementação.

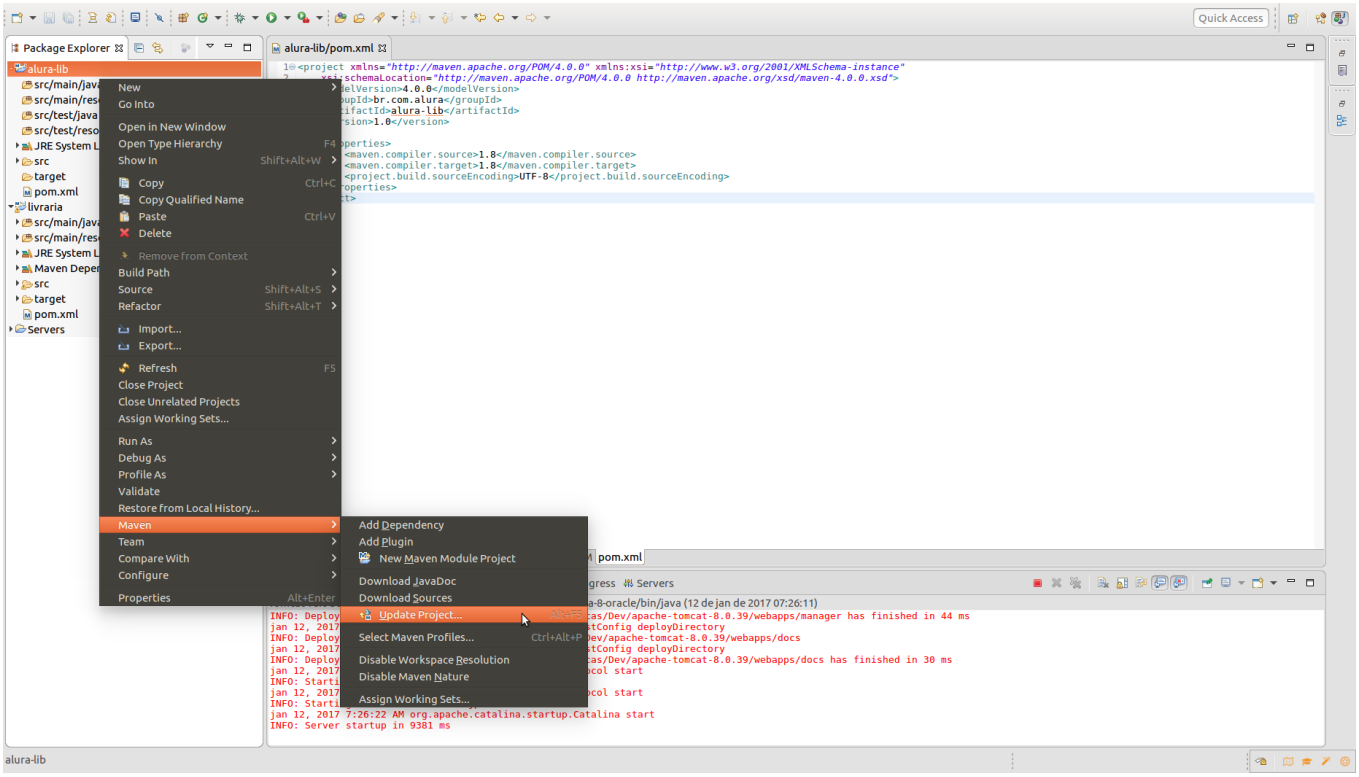
O primeiro ponto que devemos observar é que o projeto está com a versão do Java desatualizada.



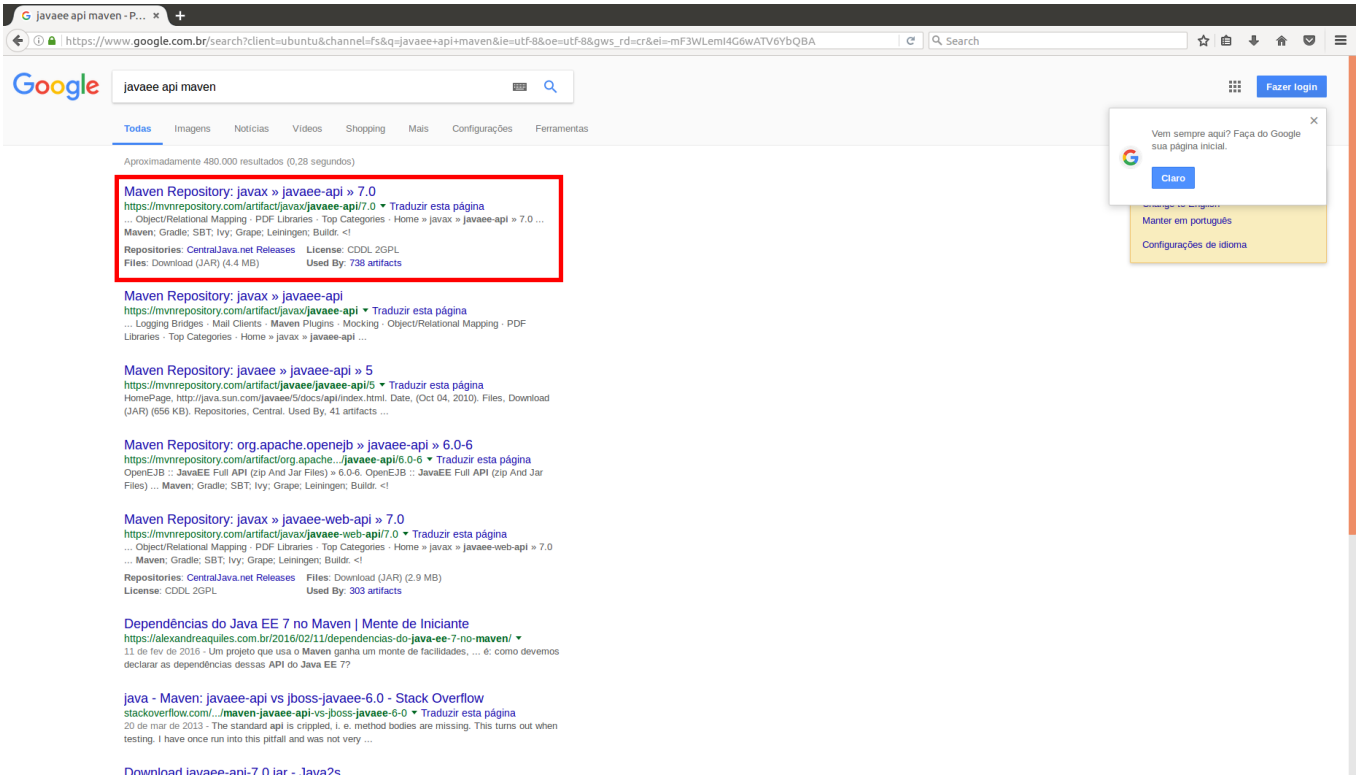
Para utilizar a última versão do Java, vamos adicionar a mesmo configuração que adicionamos no projeto `livraria`.

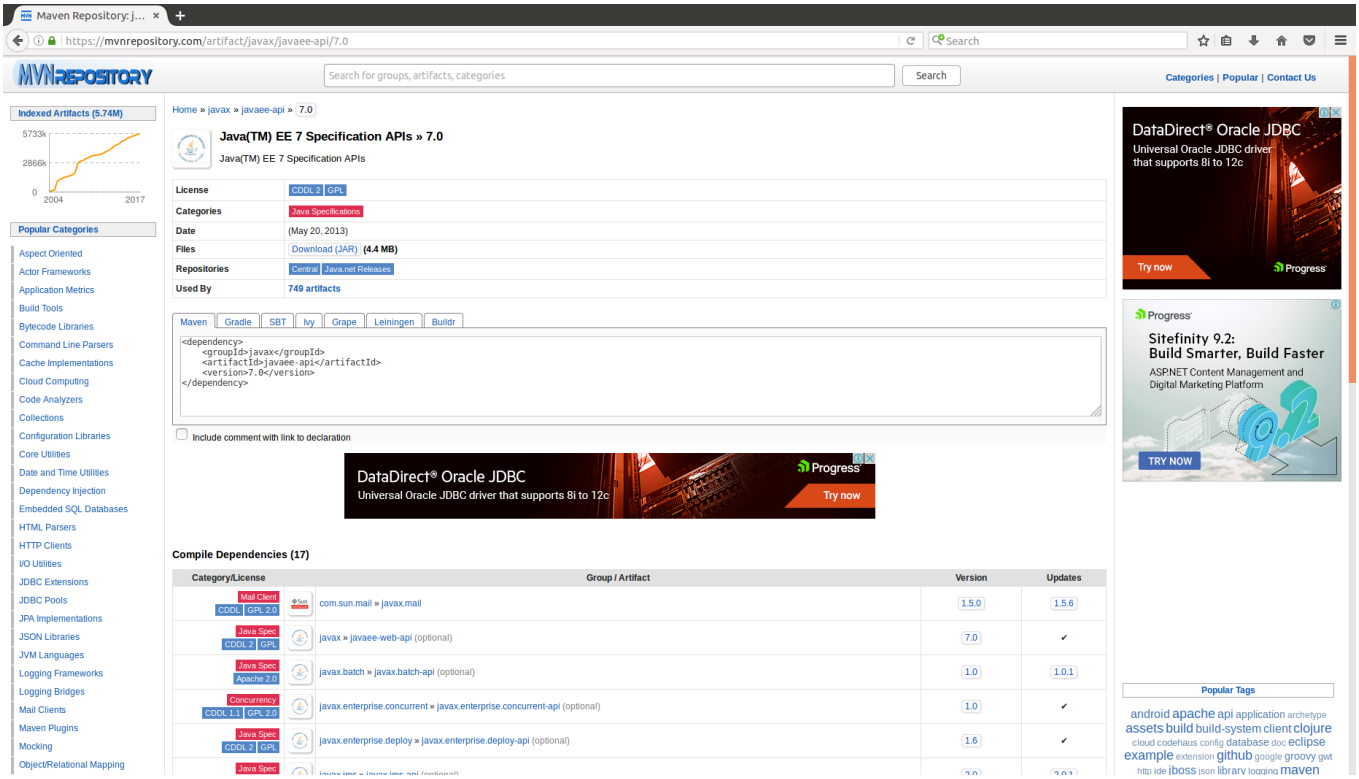
```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-insta
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xs
    <modelVersion>4.0.0</modelVersion>
    <groupId>br.com.alura</groupId>
    <artifactId>alura-lib</artifactId>
    <version>1.0</version>

    <properties>
        <maven.compiler.source>1.8</maven.compiler.source>
        <maven.compiler.target>1.8</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>
</project>
```

Para atualizar o projeto, clique com o botão direto no projeto e escolha a opção "Maven > Update Project".



Agora vamos adicionar a dependência do Java EE ao nosso projeto. Uma pesquisa no Google por "javaee api maven" é suficiente para encontrarmos a dependência no repositório do Maven.

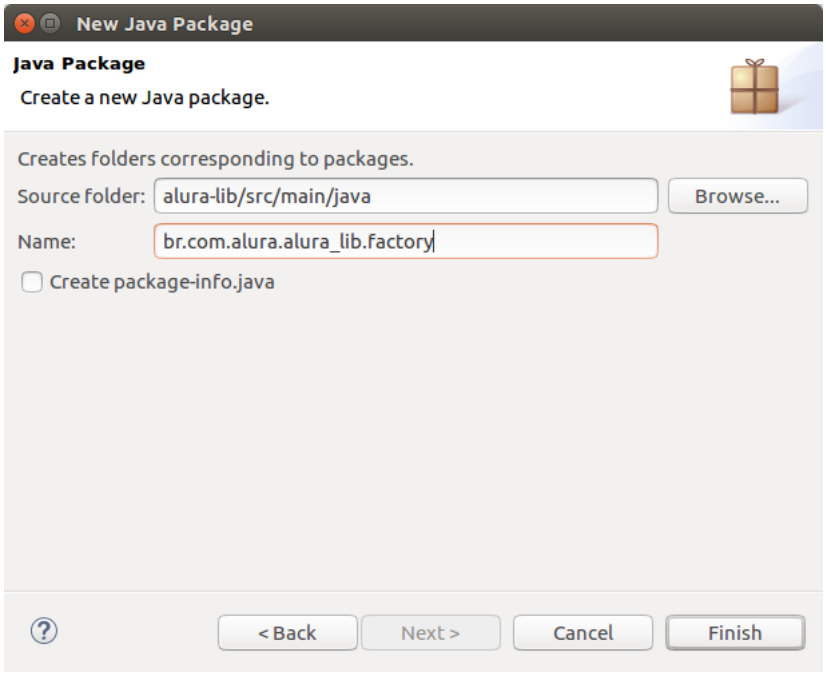




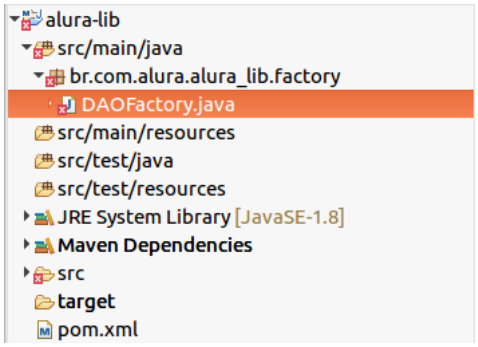
Vamos copiar o código e adicionar no `pom.xml`. Essa dependência será necessária apenas no momento em que estivermos compilando, sendo útil no momento em que formos distribuir o `jar`. Quem utilizar o `jar` que será gerado, deve possuir as interfaces e implementações. Por esse motivo, vamos adicionar o `scope provided`.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-insta
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xs
    <modelVersion>4.0.0</modelVersion>
    <groupId>br.com.alura</groupId>
    <artifactId>alura-lib</artifactId>
    <version>1.0</version>

    <properties>
        <maven.compiler.source>1.8</maven.compiler.source>

        <maven.compiler.target>1.8</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

    <dependencies>
        <dependency>
            <groupId>javax</groupId>
            <artifactId>javaee-api</artifactId>
            <version>7.0</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>
</project>
```

Dentro do diretório `src/main/java`, vamos criar um novo pacote. Todas as nossas *factories* ficarão dentro deste pacote.



Agora vamos copiar o arquivo `DaoFactory.java`, presente no projeto `livraria`, para o nosso projeto `alura-lib`, no pacote que acabamos de criar.



Como está faltando a classe `DAO` o arquivo não compila. Já que o `DAO` é genérico, e podemos utilizá-lo em outro projeto, vamos criar um pacote chamado `br.com.alura.alura_lib.dao`. Vamos copiar o arquivo `DAO.java` para o pacote recém-criado.

Na classe `DAOFactory`, basta utilizar o atalho "Ctrl + Shift + O" para organizar os imports. E o `DAOFactory` já funciona. O que falta agora é o `JPAUtil`, que é quem produz o `EntityManager`. Vamos copiar a classe `JPAUtil` para o projeto `alura-lib`, dentro do pacote `br.com.alura.alura_lib.factory`.

Apenas por questão de organização, vamos renomear a classe `JPAUtil` para `JPAFactory`.

Selecionamos o nome da classe e em seguida utilizamos o atalho "Shift + Alt + R". Digitamos o novo nome da classe (`JPAFactory`) e pressionamos "Enter".

JPAUtil.java

```
1  package br.com.alura.alura_lib.factory;
2
3  import javax.enterprise.context.RequestScoped;
9
10 public class JPAUtil {
11
12     private static EntityManagerFactory emf = Persistence
13             .createEntityManagerFactory("livraria");
14
15     @Produces
16     @RequestScoped
17     public EntityManager getEntityManager() {
18         return emf.createEntityManager();
19     }
20
21     public void close(@Disposes EntityManager em) {
22         if(em.isOpen()) {
23             em.close();
24         }
25     }
26 }
27
```

alura-lib/pom.xml | DAOFactory.java | JPAUtil.java

```
1  package br.com.alura.alura_lib.factory;
2
3  import javax.enterprise.context.RequestScoped;
9
10
11 public class JPAUtil {
12
13     private  Enter new name, press ↵ to refactor  = Persistence
14             .createEntityManagerFactory("livraria");
15
16     @Produces
17     @RequestScoped
18     public EntityManager getEntityManager() {
19         return emf.createEntityManager();
20     }
21
22     public void close(@Disposes EntityManager em) {
23         if (em.isOpen()) {
24             em.close();
25         }
26     }
27 }
```

A estrutura final ficará da seguinte forma:

No projeto `livraria`, podemos excluir os arquivo `JPAUtil.java` e `DAO.java`, bem como o `DAOFactory`.

A estrutura ficará da seguinte forma:

Agora é necessário adicionar o `alura-lib` dentro do projeto `livraria`. Já que os dois são projetos Maven, podemos adicionar a dependência no `pom.xml` do projeto `livraria`.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-insta
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.x
    <modelVersion>4.0.0</modelVersion>
    <groupId>br.com.alura</groupId>
    <artifactId>livraria</artifactId>
    <version>1.0.0-SNAPSHOT</version>
    <packaging>war</packaging>

    <properties>
        <!--  properties-->
    </properties>



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

        <dependency>
            <groupId>br.com.alura</groupId>
            <artifactId>alura-lib</artifactId>
            <version>1.0</version>
        </dependency>
    </dependencies>

    <repositories>
        <!repositorios>
    </repositories>

    <build>
        <finalName>livraria</finalName>
        <!-- plugins -->
    </build>
```

```
</project>
```

Precisaremos atualizar os pacotes para que agora sejam referenciados os pacotes do `alura-lib` e tudo volte a funcionar. Basta abrir as classes que não compilam - `AutorBean`, `LivroBean` e `VendasBean`, e pressionar "Ctrl + Shift + O" em cada uma delas.

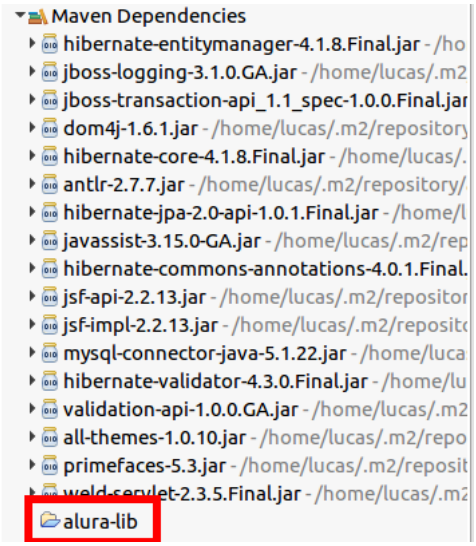
A classe `UsuarioDao` não funciona por conta da seguinte linha dentro do método `existe()`:

```
EntityManager em = new JPAUtil().getEntityManager();
```

Isso não é mais necessário. Podemos receber o `EntityManager` via injeção de dependências:

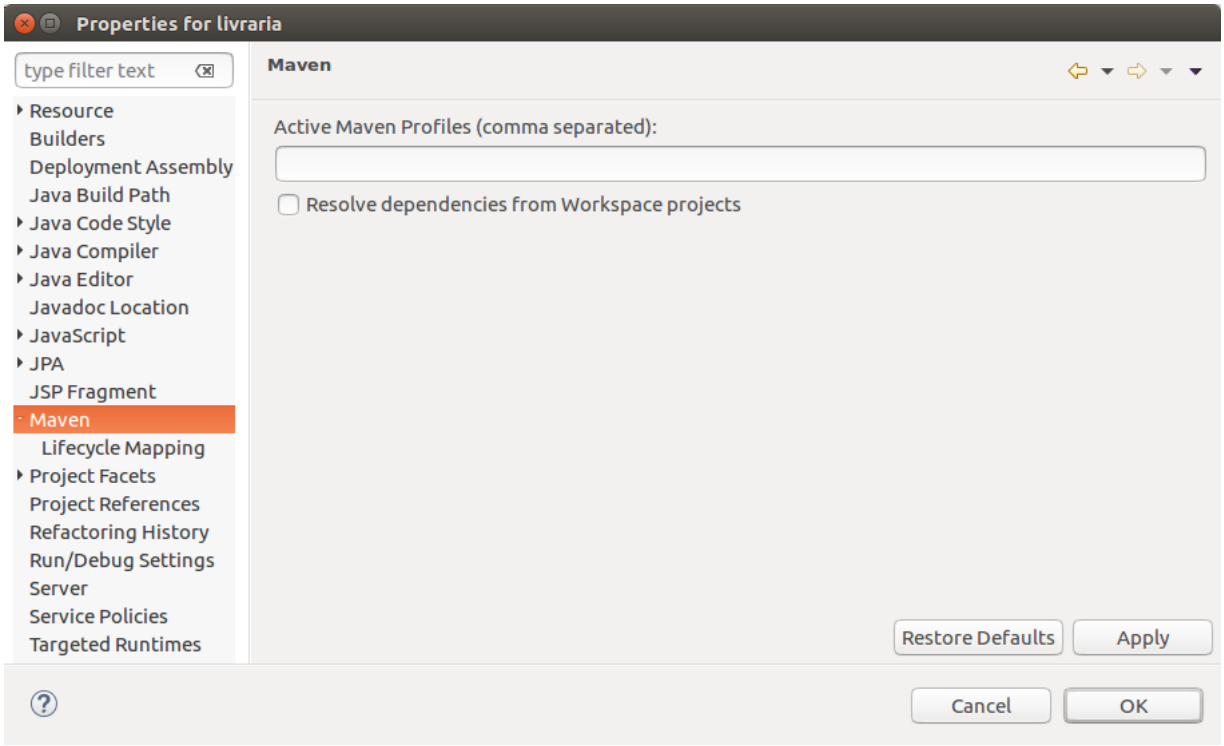
```
public class UsuarioDao {

    private EntityManager em;

    @Inject
    public UsuarioDao(EntityManager em) {
        this.em = em;
    }

    public boolean existe(Usuario usuario) {
        TypedQuery<Usuario> query = em.createQuery(
                  " select u from Usuario u "
                + " where u.email = :pEmail and u.senha = :pSenha", Usuario.class);

        query.setParameter("pEmail", usuario.getEmail());
        query.setParameter("pSenha", usuario.getSenha());
        try {
            query.getSingleResult();
        } catch (NoResultException ex) {
            return false;
        }

        em.close();

        return true;
    }

}
```

Pronto! Nosso projeto está funcionando. Se olharmos no projeto `livraria`, em "Maven Dependencies", temos os `jars` das outras dependências, mas do `alura-lib` não temos um `jar`, e sim um projeto.



Isso acontece porque o Eclipse tenta resolver isso, se temos projetos no mesmo Workspace, em vez de incluir o `jar` (que estará no repositório local, ou em um repositório remoto que será baixado para o repositório local), ele inclui o projeto. Queremos que ele inclua o `jar`, então vamos mudar essa configuração.

Vamos clicar com o botão direito em cima do projeto `livraria` e escolher a opção "Properties". Em "Maven", vamos desabilitar a opção "Resolve dependencies from Workspace projects":

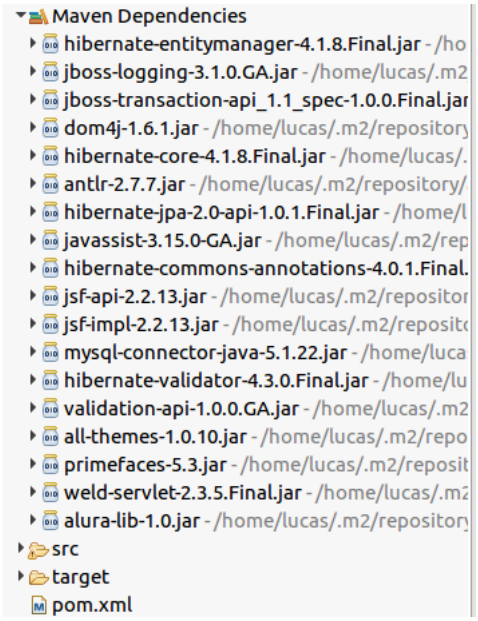


Clique em "Apply", "Yes", no diálogo de confirmação, e em seguida clique no botão "Ok".

O projeto `livraria` irá quebrar porque não temos essa dependência no repositório local. Para instalar, vamos clicar com o botão direito em no projeto `alura-lib`, escolher a opção "Run As > Maven Install". Isto fará com que o Maven instale a dependência no repositório local.

Após isso, o Maven conseguiu instalar, mas o projeto `livraria` continua com erro. Precisamos atualizar as referências. Vamos clicar com o botão direito em cima do projeto `livraria`, e escolher a opção "Maven > Update Project" (O atalho é "Alt + F5").

Agora se olharmos nas dependências, podemos ver que temos o `jar` do `alura-lib` :

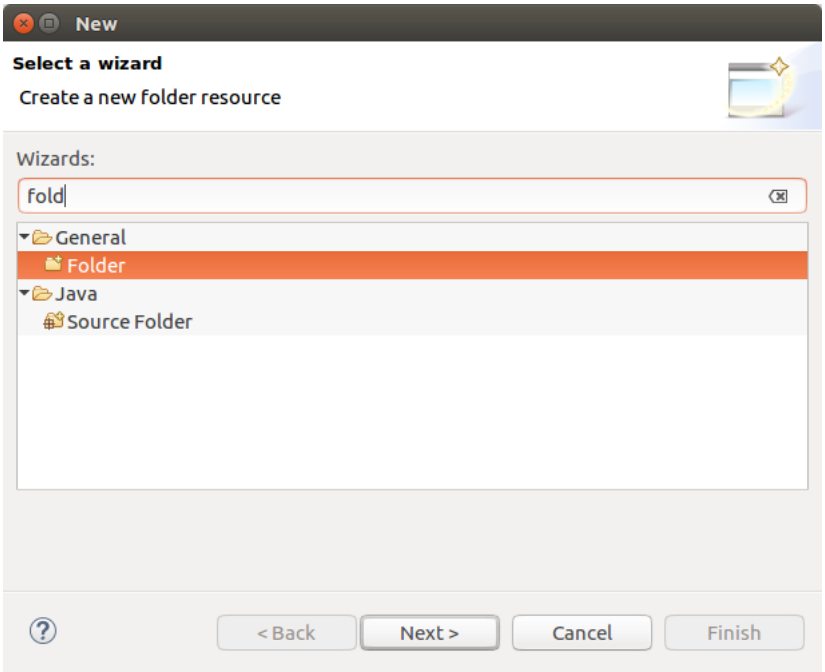


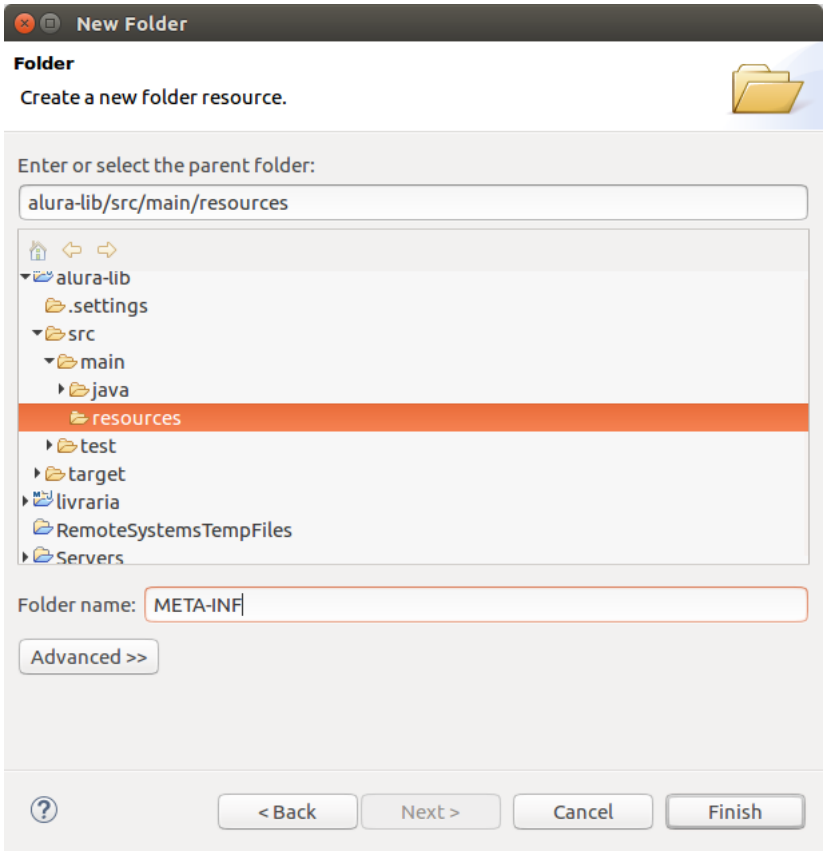
Aparentemente tudo está funcionando. Vamos iniciar o servidor e testar. Mas ao tentar iniciar o projeto, recebemos um erro:

```
Unsatisfied dependencies for type DAO<Autor> with qualifiers @Default
```

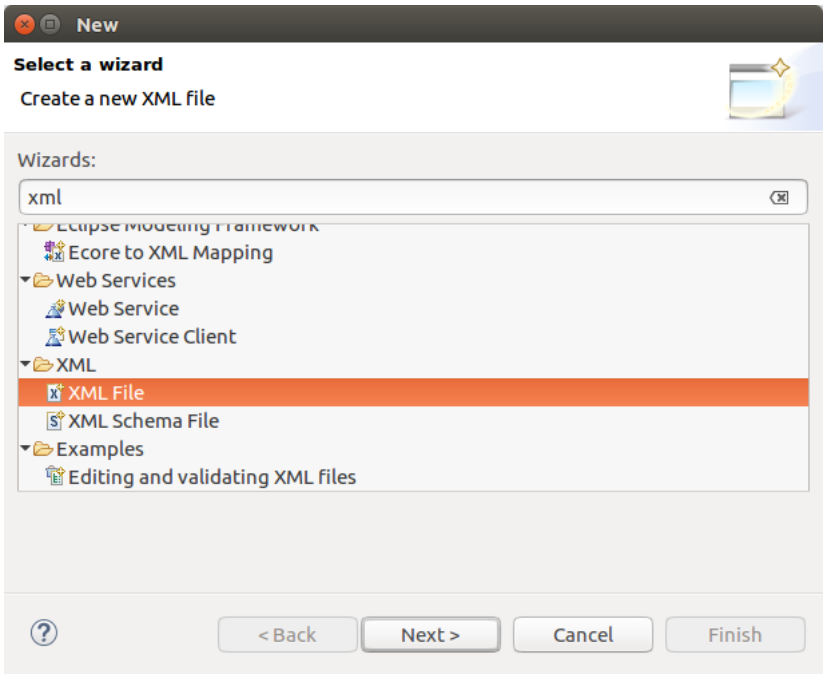
O CDI não encontrou alguém que satisfaça a dependência do `DAO` . Mas temos o `DAO` ! Ele está no nosso projeto. Lembre-se que no inicio falamos que o CDI exige que o nosso projeto tenha o arquivo de configuração `beans.xml` . Ele não consegue identificar o `alura-lib` como sendo um *bean package* porque ele não tem arquivo `beans.xml` .

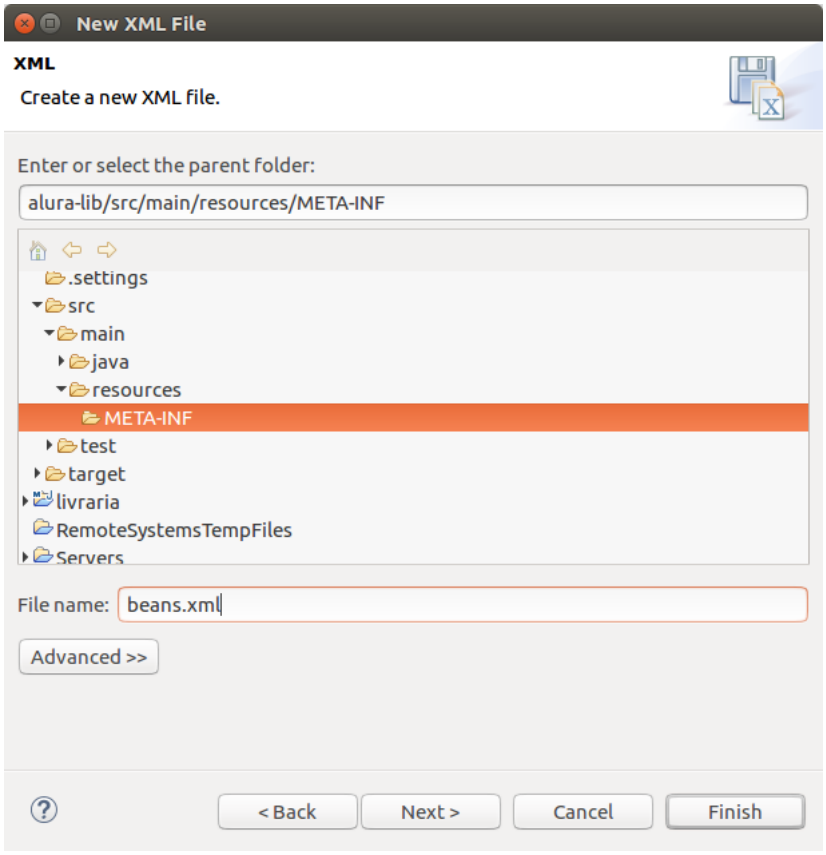
No projeto `alura-lib` , clicando de `src/main/resources` , e depois utilizando o atalho "Ctrl + N", vamos adicionar uma nova pasta chamada `META-INF` .





Após isso, vamos selecionar a pasta que acabamos de criar, pressionar novamente "Ctrl + N" e criar um arquivo chamado `beans.xml` . Após definir o nome do arquivo, basta clicar em "Finish"





Vamos copiar o conteúdo do arquivo `beans.xml` do projeto `livraria` para o arquivo `beans.xml` do projeto `alura-lib`.

```
<?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/bea
       version="1.2" bean-discovery-mode="all">

    <!-- some content -->
</beans>
```

Como fizemos uma alteração no `alura-lib`, precisamos instalar novamente o `jar` no repositório local. Para isso clicamos com o botão direito em cima do projeto e em seguida nas opções "Run As > Maven Install".

No projeto `livraria`, temos que clicar com o botão direito e em seguida "Maven > Update Project". Se aparecer um tela, basta clicar em "Ok".

Como a biblioteca não usou a versão, ele irá manter o mesmo nome do `jar` sempre, no projeto livraria. O tomcat irá entender que é o mesmo `jar`, porque possui o mesmo nome e não atualizará o projeto.

Por isso precisamos estar sempre limpando o servidor antes de *startar*. Para isso temos que clicar com o botão direito e selecionar "Clean". Na janela que irá aparecer, basta clicar em "Ok".