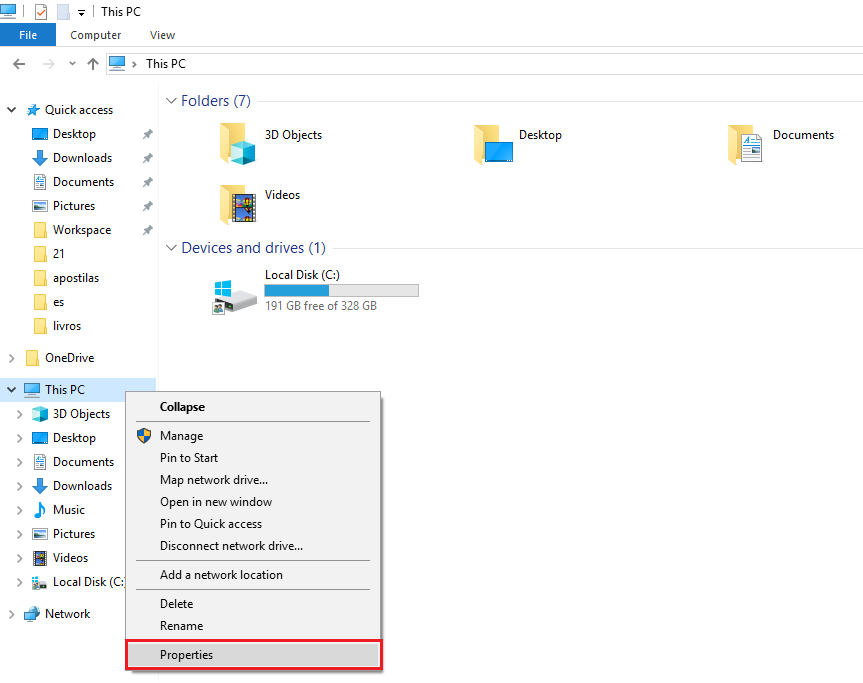
Maven: Build do zero a web

Vamos instalar o Maven no Windows, antes você deve baixá-lo no site do [Apache](https://maven.apache.org/download.cgi.). Como desejamos executar no Windows, baixe o arquivo zip, na opção "Binary zip archive".

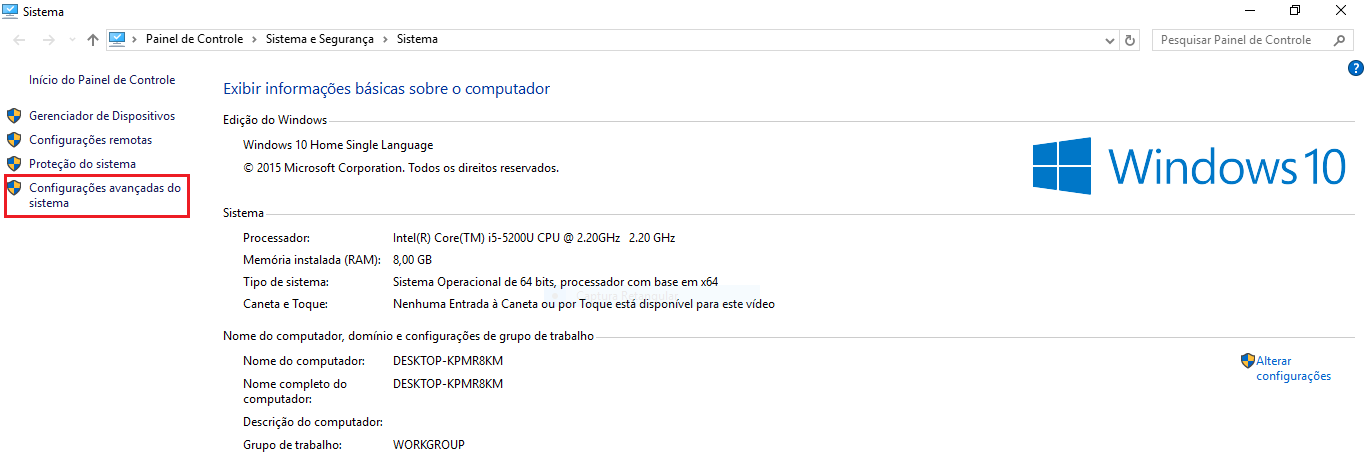
Após o Download, extraia o .zip em um diretório de sua preferência. O próximo passo é adicionar o Maven nas variáveis de ambiente, para que seja possível executar o comando mvn independente de qual seja o nosso diretório atual do cmd.

No Windows 10, acesse Meu Computador (Tecla do Windows + E):

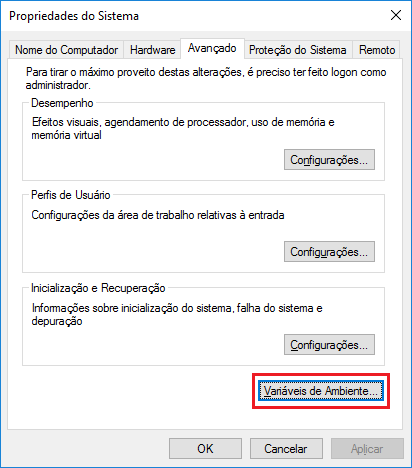


Se você estiver usando Windows 7, no menu iniciar, em "Computador", clique com o botão direito e escolha a opção "Propriedades".

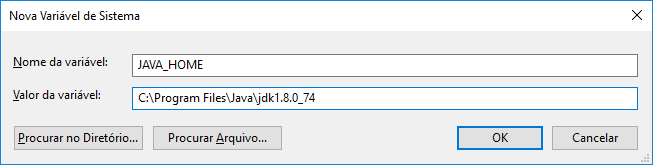
Outra forma de acessar a tela de propriedades é utilizando o atalho "Tecla do Windows + Pause Break". Na tela que será exibida, no lado esquerdo, escolha a opção "Configurações avançadas do sistema":



Na tela que será exibida, clique na aba "Avançado". Em seguida clique no botão "Variáveis de ambiente".

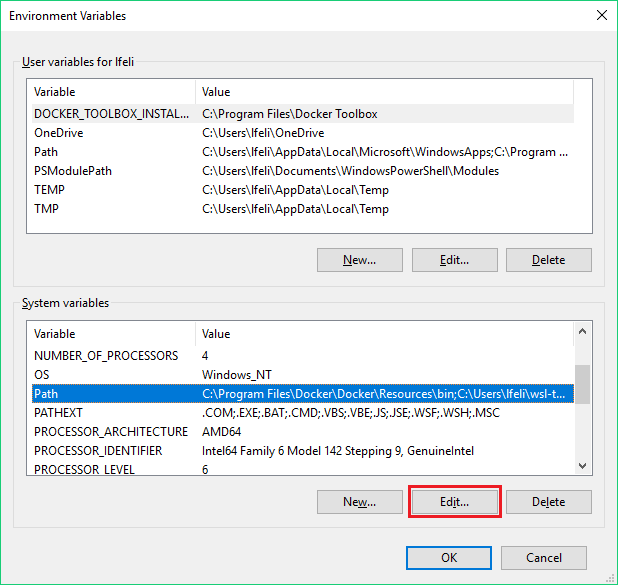


Na próxima tela, na área "Variáveis do sistema" clique no botão "Novo...". O maven necessita de uma variável de ambiente chamada JAVA\_HOME que aponta para o diretório onde está o seu JDK. Crie essa variável e lembre-se de alterar o caminho de acordo com sua instalação do JDK:

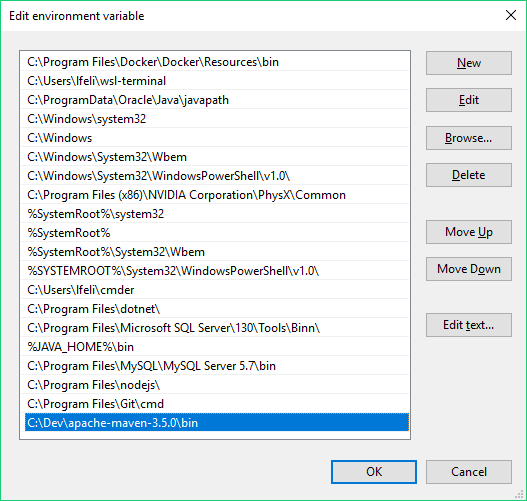


Clique no botão "OK" para confirmar a criação da variável de ambiente JAVA\_HOME.

Ainda na tela das variáveis de ambiente, em "Variáveis de Usuário", busque pela variável de ambiente "Path" e clique em "**Editar**". **Não** crie uma nova variável, altere a já existente:



Na próxima tela, clique no botão "Novo" e adicione o caminho completo até o diretório bin do seu Maven. Lembre-se de alterar de acordo com o diretório que você escolheu:



**Observação:** Em versões anteriores do Windows, os diretórios presentes na variável são separados por ";" (ponto e vírgula). Se for o seu caso, basta adicionar um ";" no final do valor já existente e adicionar o caminho para o diretório bin do Maven. **Não** remova o conteúdo já existente na variável PATH.

Clique no botão "OK" em todas as janelas abertas para confirmar as alterações. Abra um novo *Prompt* de Comando. Ao digitar o comando mvn -v informações sobre a instalação do Maven devem ser mostradas. Teste para ver se tudo funcionou como esperado.

Vamos criar um projeto simples com o Maven. O nome do projeto será produtos e o pacote principal será br.com.alura.maven.

Para isso você deve executar o seguinte comando no diretório em que deseja que o projeto seja criado:

mvn archetype:generate -DartifactId="produtos" -DgroupId="br.com.alura.maven" -DinteractiveMode="false" -DarchetypeArtifactId="maven-archetype-quickstart"

Após realizar download de bastante coisa, caso seja a primeira vez que você executou o comando, o maven deve mostrar uma mensagem indicando que o projeto foi criado com sucesso.

No lugar de executar o javac, vamos utilizar o Maven para compilar a classe App.java que o projeto produtos contém. Para isto, execute mvn compile. Verifique se dentro do diretório target/classes, o arquivo class foi devidamente gerado.

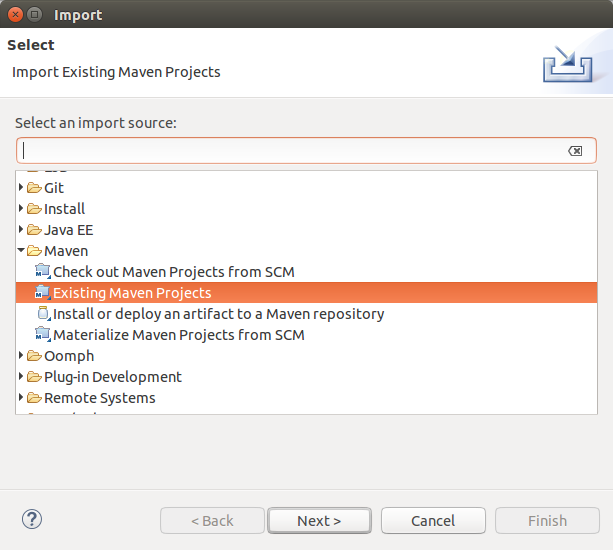
Utilize o Maven para rodar os testes do projeto. Para isto execute o comando mvn test. Além de verificar a saída do comando no terminal, verifique dentro do diretório target quais diretórios foram gerados e seus conteúdos.

Para finalizar, utilize o comando mvn clean para limpar o diretório target e manter apenas os arquivos fonte.

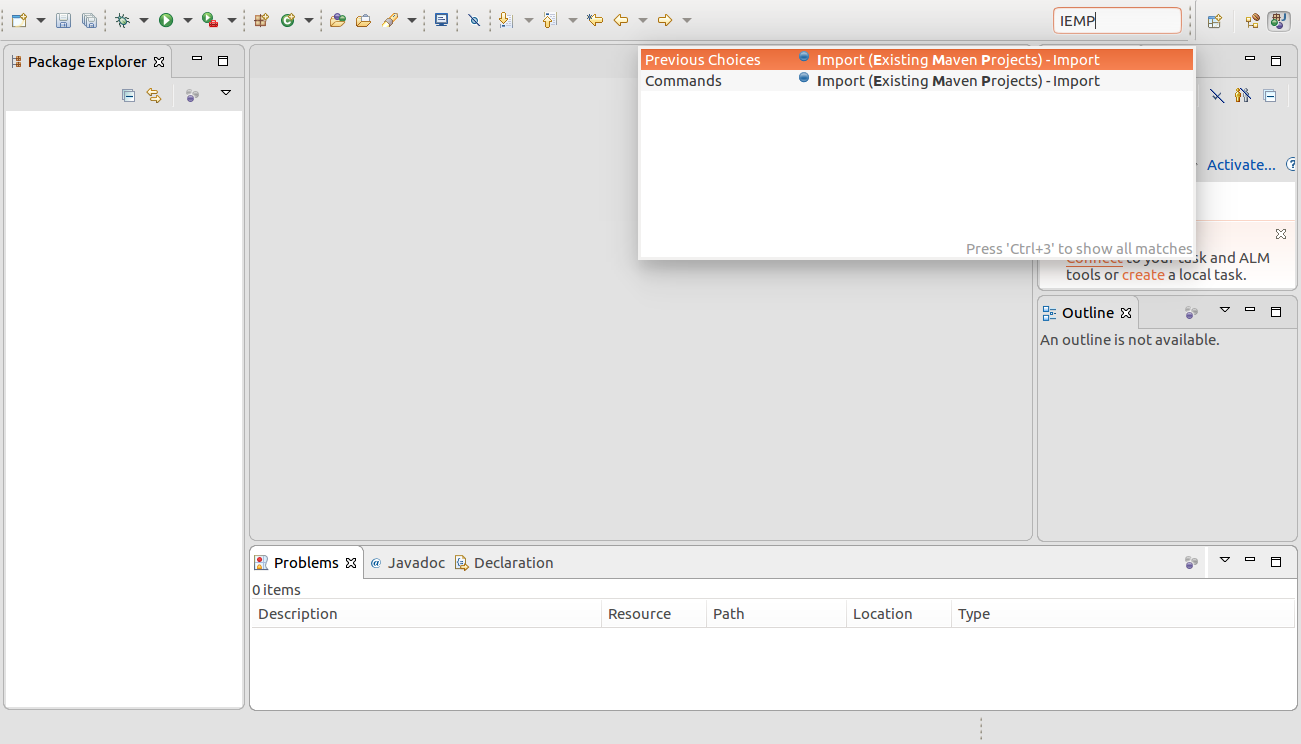
**Importando o projeto no Eclipse**

Vamos importar, no Eclipse, o projeto Maven criado no capítulo anterior. De preferência, abra a versão mais recente do Eclipse, o Oxygen, que pode ser encontrada para download [aqui](http://www.eclipse.org/downloads/). É importante que você tenha a versão do Eclipse com suporte a Java EE, pois mais à frente neste curso criaremos um projeto *web*.

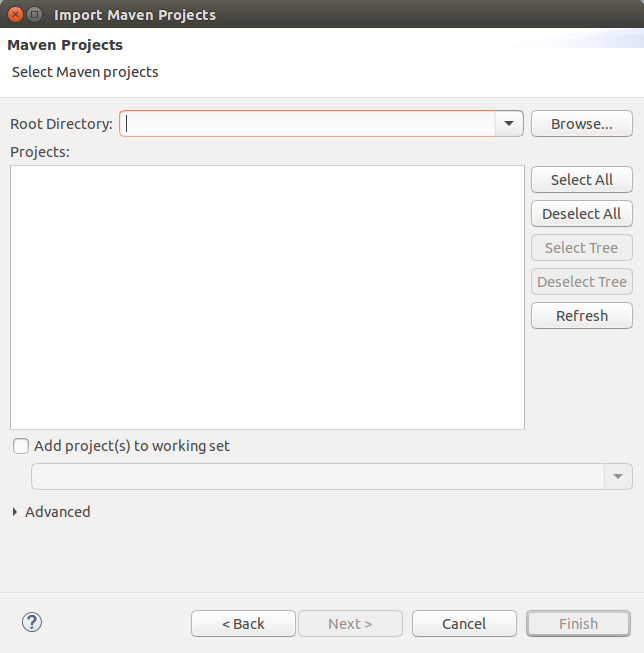
Importar um projeto Maven no Eclipse é simples: ao acessar o menu File > Import, aparecerá uma janela. Nessa janela, escolha a opção Maven > Existing Maven Projects.

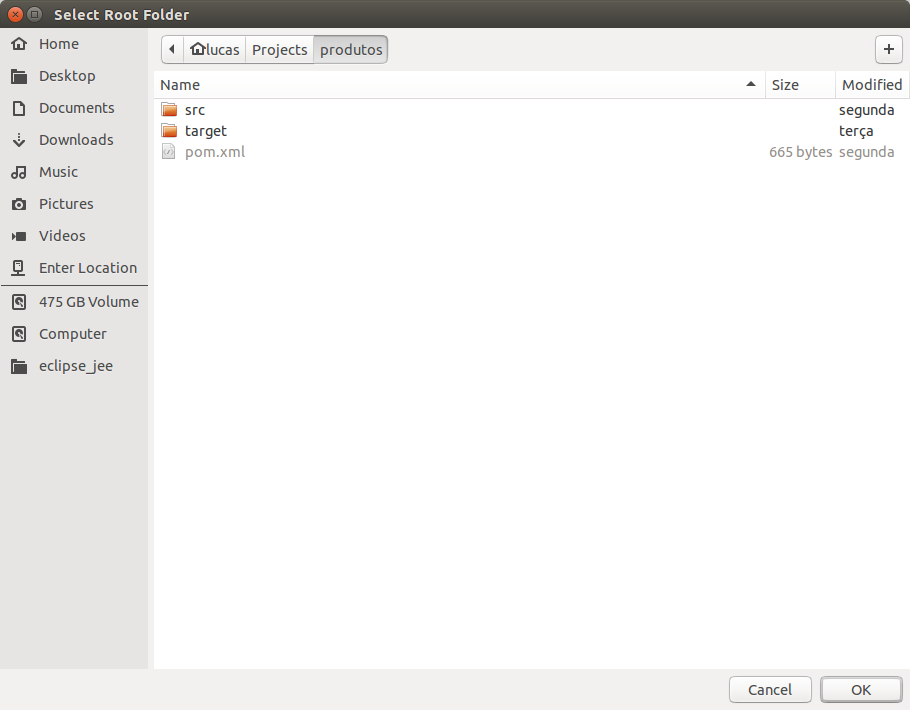


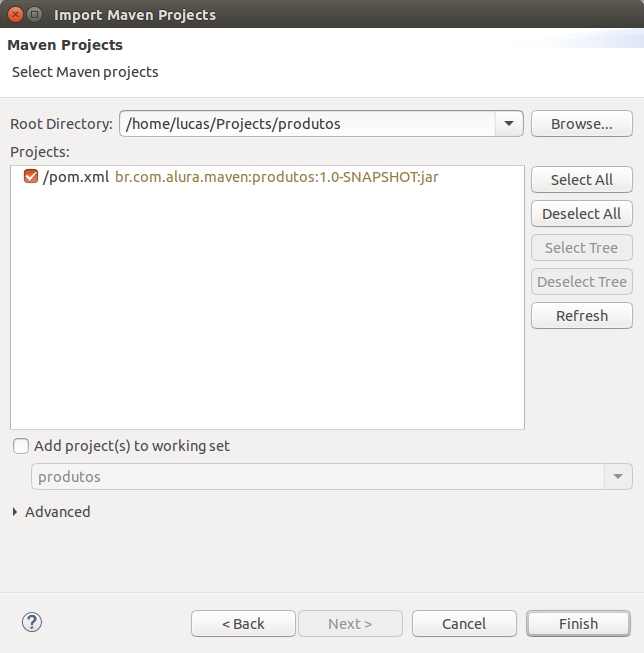
Uma alternativa é utilizar o atalho "Ctrl + 3" , o que abrirá o "Quick Access do Eclipse". Digite "IEMP" (*Import Existing Maven Projects*) e pressione "Enter". Você será direcionado diretamente para próxima tela, que solicita o diretório onde se encontra o projeto.



Na próxima janela, selecione o diretório produtos. O Eclipse deve identificar o arquivo pom.xml e reconhecer o projeto Maven.







Finalize a importação do Projeto clicando em "Finish".

O [PMD é um analisador de código-fonte](https://pmd.github.io/) que consegue encontrar algumas falhas no nosso código, como por exemplo, variáveis que não estão sendo utilizadas.

Para gerar o relatório, você deve utilizar o comando mvn pmd:pmd. Caso existam violações no código-fonte, o arquivo pmd.html será criado em target/site, indicando quais são as violações.

A primeira coisa a se fazer é adicionar o plugin do PMD ao pom.xml. Nós adicionamos depedências na tag <dependencies>, mas como aqui queremos alterar o *build* do projeto, os *plugins* ficam dentro da tag build:

<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/maven-v4\_0\_0.xsd">

<!-- informações sobre o projeto -->

<dependencies>

<!-- dependências do junit e xstream -->

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-pmd-plugin</artifactId>

<version>3.6</version>

<executions>

<execution>

<phase>verify</phase>

<goals>

<goal>check</goal>

</goals>

</execution>

</executions>

</plugin>

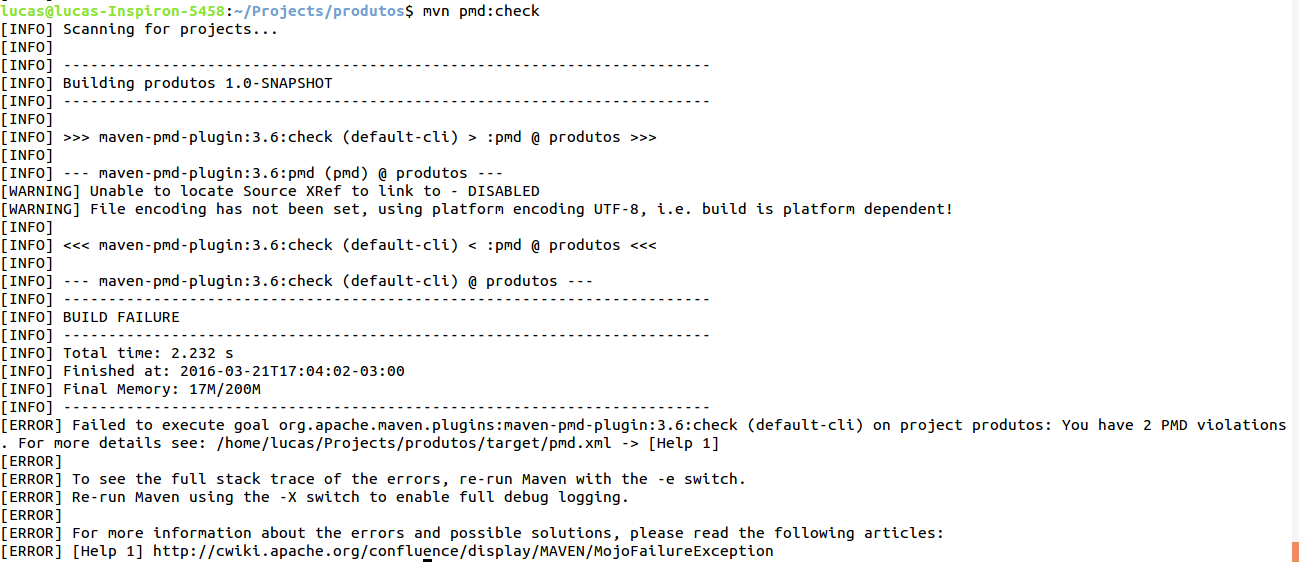
</plugins>

</build>

</project>

Vimos que um dos goals do plugin PMD, é o pmd:check. Esse goal [falha o build se existir alguma violação das regras do PMD no código-fonte](https://maven.apache.org/plugins/maven-pmd-plugin/check-mojo.html).

Teste o comando mvn pmd:check. Perceba que nesse momento o build falhará, pois temos violações nas validações:



Já vimos que se executarmos o comando mvn pmd:check, o *build* falhará, pois estamos violando as validações ao manter variáveis não utilizadas no código.

Porém se executarmos a fase verify, o *build* será concluído com sucesso. Não é o que desejamos. Queremos que o *build* falhe caso o relatório do PMD contenha erros.

Adicione o plugin do PMD ao seu pom.xml e faça com que o *goal* pmd:check seja executado durante a fase verify do ciclo de vida do build.

Para alterar o *goal* na fase de verify do nosso projeto, utilizamos *plugins*. O pom.xml ficará com o seguinte código:

<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/maven-v4\_0\_0.xsd">

<!-- informações sobre o projeto -->

<dependencies>

<!-- dependências -->

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-pmd-plugin</artifactId>

<version>3.6</version>

<executions>

<execution>

<phase>verify</phase>

<goals>

<goal>check</goal>

</goals>

</execution>

</executions>

</plugin>

</plugins>

</build>

</project>

Dessa forma alteramos o ciclo de vida do nosso projeto. Indicamos que o *goal* check do plugin pmd deve ser executado durante a fase de verify. Se você executar o comando mvn verify agora verá o *build* falhar.

**Utilizando o plugin do JaCoCo**

Adicione o plugin do JaCoCo ao projeto para que seja possível gerar relatórios sobre a cobertura de testes do projeto. Esse tipo de relatório é interessante pois é possível ver de uma forma fácil o que não testamos em nossa aplicação.

Faça com que os *goals* prepare-agent e report sejam adicionados ao ciclo de vida. Mantenha a execução do *goal* na fase padrão do plugin, não é necessário deixar a fase explícita no pom.xml.

Execute o comando mvn verify, e no diretório target/site/jacoco você encontrará a página HTML index.html. Abra o arquivo e veja o relatório detalhado sobre o código não coberto por testes. No momento, nenhum código do nosso projeto está testado.

No arquivo pom.xml, basta adicionar o plugin.

Lembre-se que caso a versão do *plugin* não seja definida, o Maven irá utilizar a última versão. Aqui vamos utilizar a versão mais recente no momento, encontrada no *[Maven Repository](http://mvnrepository.com/artifact/org.jacoco/jacoco-maven-plugin" \t "_blank)*:

<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/maven-v4\_0\_0.xsd">

<!-- informacoes -->

<dependencies>

<!-- depedências -->

</dependencies>

<build>

<plugins>

<!-- PMD plugin -->

<plugin>

<groupId>org.jacoco</groupId>

<artifactId>jacoco-maven-plugin</artifactId>

<version>0.7.6.201602180812</version>

<executions>

<execution>

<goals>

<goal>prepare-agent</goal>

<goal>report</goal>

</goals>

</execution>

</executions>

</plugin>

</plugins>

</build>

</project>

**Atualizando as dependências do projeto**

No próximo exercício, vamos criar um teste de unidade com uma versão mais nova do JUnit do que a declarada no pom.xml.

Você pode sempre encontrar as informações sobre as bibliotecas do mundo Java e como adicioná-las ao seu projeto no mvnrepository. No caso do JUnit, as informações encontram-se [aqui](http://mvnrepository.com/artifact/junit/junit).

Em projetos com muitas dependências, com o passar do tempo elas podem ficar desatualizadas, caso você queira atualizar as dependências para suas versões mais atuais, poderá utilizar o seguinte comando:

mvn versions:use-latest-versions

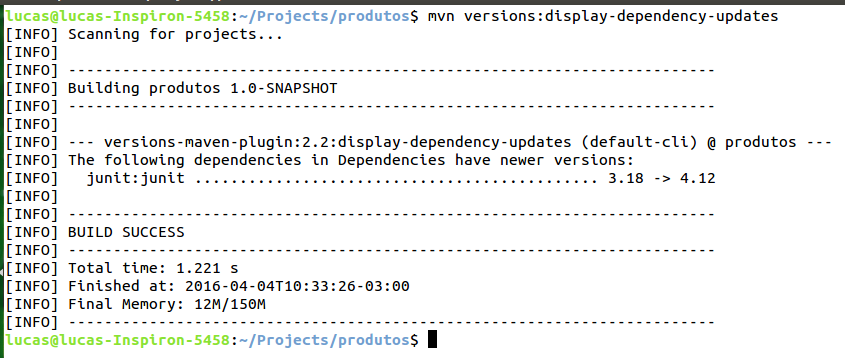
Você pode encontrar a documentação desse goal [aqui](http://www.mojohaus.org/versions-maven-plugin/use-latest-versions-mojo.html). Tente executar esse comando no seu projeto e perceba que o JUnit foi atualizado para a versão mais recente no pom.xml.

Perceba que o comando modifica o pom.xml atualizando as versões de todas as dependências. Isso pode ser perigoso, especialmente se no projeto não existirem bons [testes de unidade](https://www.alura.com.br/curso-online-tdd) e [integração](https://www.alura.com.br/curso-online-selenium), pois se algo deixar de funcionar isso pode ser descoberto apenas em produção.

Existe um outro goal que verifica por atualizações sem de fato alterar o pom.xml:

mvn versions:display-dependency-updates

Ele mostrará as dependências que possuem novas versões, mas não atualizará para você:



Os dois goals fazem parte do plugin Versions, cuja documentação pode ser encontrada [aqui](http://www.mojohaus.org/versions-maven-plugin/).

Escolha uma das opções e atualize a versão do JUnit para que no próximo exercício seja possível criar um teste de unidade utilizando os recursos dessa nova versão.

No fim, o pom.xml deve conter a versão 4.x do JUnit:

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

</dependency>

**Criando um teste de unidade**

Você pode excluir a classe AppTest, que apenas é criada de exemplo e utiliza uma versão antiga do JUnit.

Na pasta de teste (src/test/java), crie a classe ProdutoTest. Vamos testar se o método getPrecoComImposto() em Produto, realiza o cálculo correto:

package br.com.alura.maven;

import org.junit.Test;

import static org.junit.Assert.assertEquals;

public class ProdutoTest {

@Test

public void verificaPrecoComImposto() {

Produto bala = new Produto("juquinha", 0.10);

assertEquals(0.11, bala.getPrecoComImposto(), 0.00001);

}

}

Na classe Produto adicione o método getPrecoComImposto():

public class Produto {

private final String nome;

private final double preco;

// restante do código

public double getPrecoComImposto() {

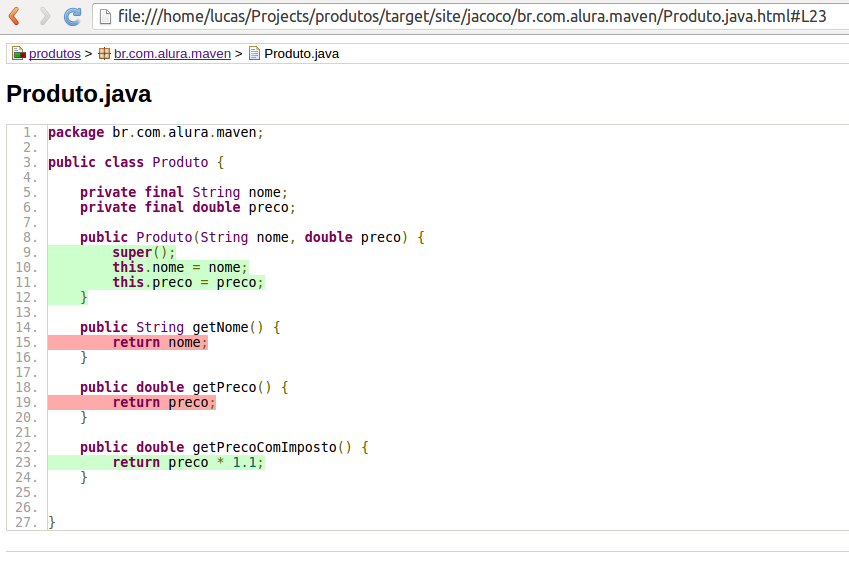
return preco \* 1.1;

}

}

Execute novamente o comando mvn verify e veja o novo relatório gerado pelo plugin do JaCoCo.

Agora deve aparecer que uma porcentagem do nosso código está coberta por testes. Se você for navegando até chegar no código da classe Produto poderá ver qual é exatamente o código que é coberto:

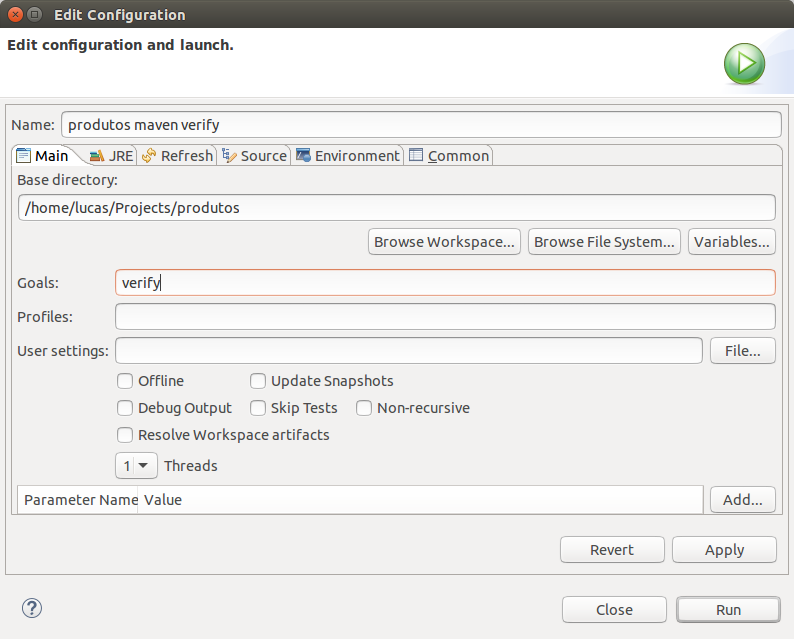


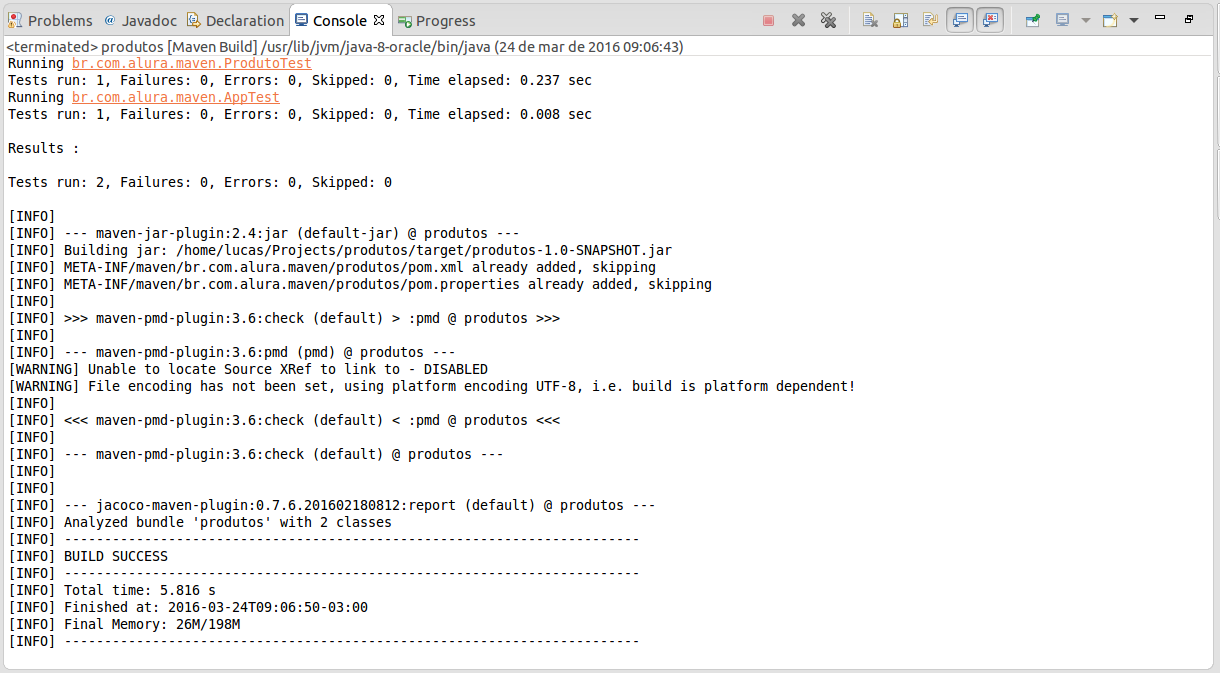
**Executando os comandos através do Eclipse**

Faça as configurações necessárias para executar o mvn verify através do Eclipse. Dessa forma quando você executar o projeto será possível escolher a fase verify sem a necessidade de executar no terminal.

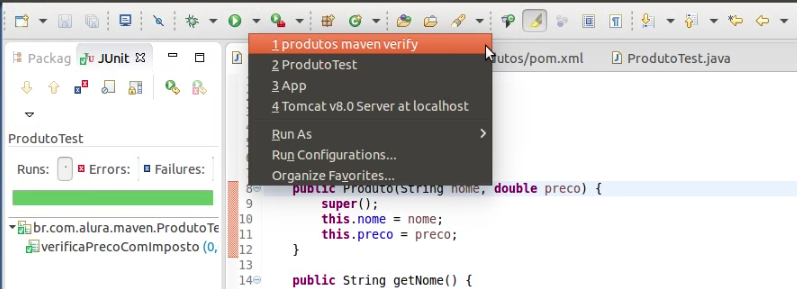
Run As -> 5 Maven build

Na janela que irá ser exibida em "Goals", adicione verify. Clique em "Run". Você verá o comando ser executado no Console do Eclipse.



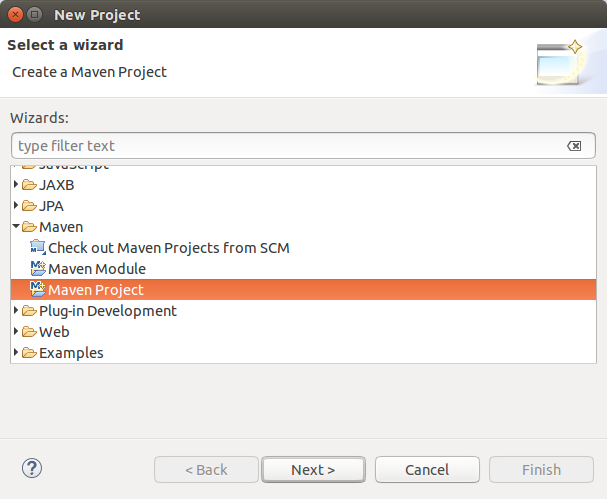


A partir desse momento, a opção irá aparecer nas opção de "Run" do Eclipse:

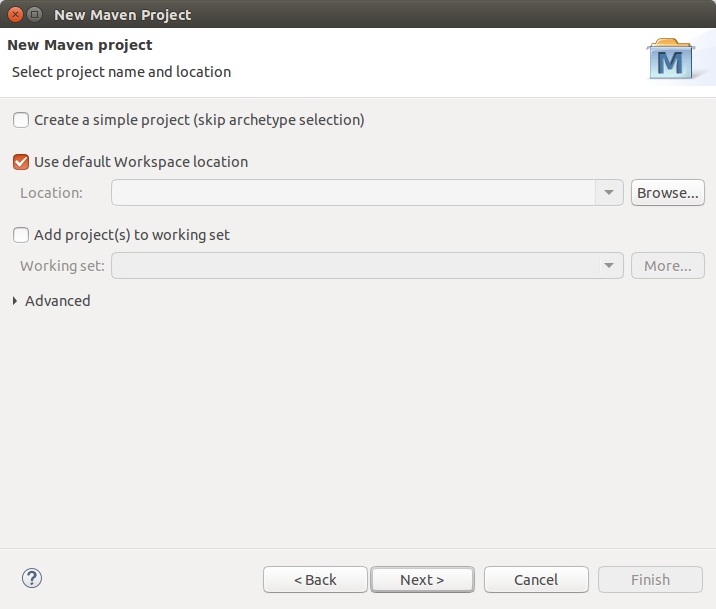


**Criando um projeto Maven no Eclipse**

Para abrir o *wizard* de criação do projeto podemos utilizar o menu File > New > Project.... Na janela que será mostrada você deve escolher "Maven Project":

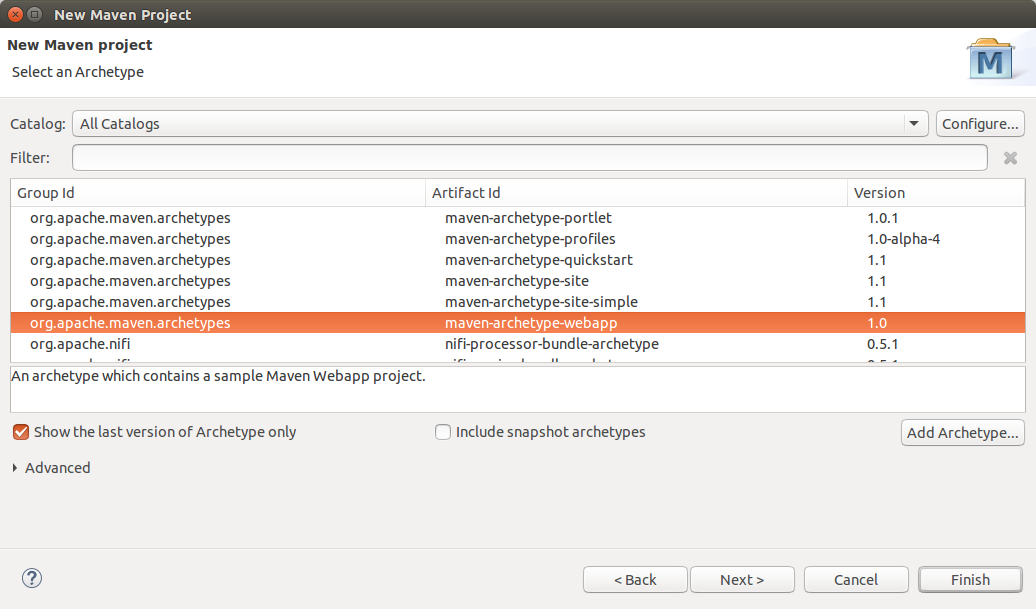


Na próxima tela, não é necessário realizar modificações a não ser que queira mudar o local onde o projeto será criado.

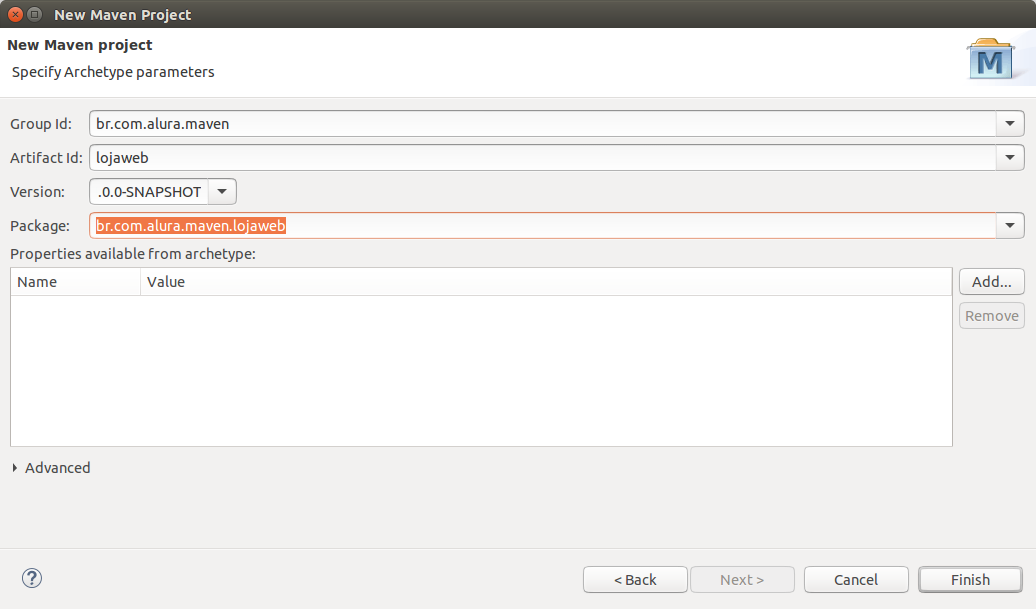


Na próxima tela selecione o *archetype* maven-archetype-webapp. Dessa forma indicamos que desejamos criar um projeto Java Web, e que o pacote gerado é do tipo .war.

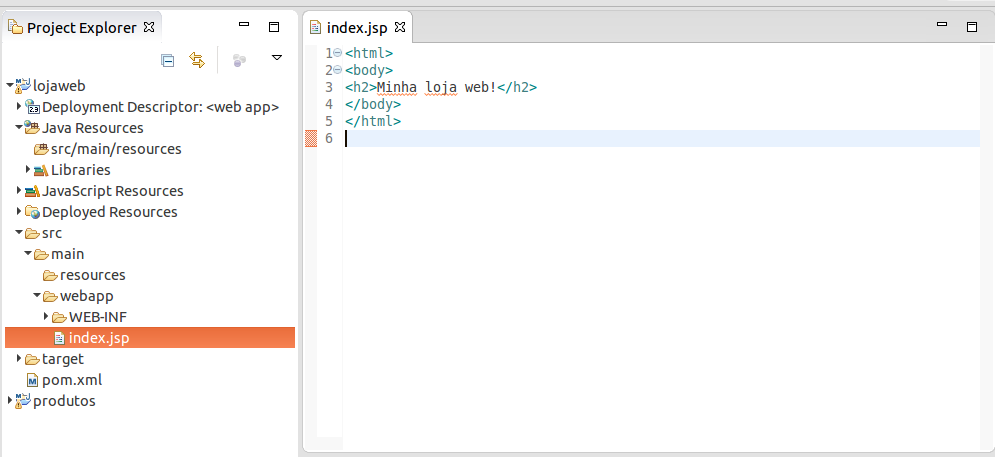
É normal o Eclipse demorar um pouco para mostrar as opções, pois como de costume, na primeira vez serão baixadas as informações da internet.



Insira as informações sobre o projeto:



Para finalizar, clique no botão "Finish". O projeto será exibido na aba "Project Explorer" do Eclipse:



Pronto! O projeto web está criado. Se você abrir o pom.xml verá que, diferentemente do projeto anterior, aqui o pacote gerado será do tipo .war em vez de .jar.

Nesse momento, apesar de ser executado com sucesso no Jetty, o Eclipse está indicando que existe um erro no projeto, mais precisamente na página index.jsp. Por que isso ocorre? Como resolver?

O problema ocorre porque o .jsp, por debaixo dos panos, é uma Servlet, e precisamos que a classe HttpServlet esteja disponível no classpath. O mesmo problema ocorreria se tentássemos criar uma Servlet: não conseguiríamos herdar de HttpServlet.

O projeto funciona bem no Jetty pelo fato de que o servidor já possui uma implementação da API de Servlets e a disponibiliza para o nosso arquivo .war.

Porém, como o Eclipse não consegue encontrar a classe nesse momento, ele indica o erro. Para resolver isso, e inclusive sermos capazes de criar uma Servlet, adicione a API nas dependências do 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/maven-v4\_0\_0.xsd">

<!-- informações sobre o projeto -->

<dependencies>

<!-- junit -->

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>javax.servlet-api</artifactId>

<version>3.1.0</version>

</dependency>

</dependencies>

<!-- build -->

</project>

Feito isso, perceba que o erro deixa de existir, porém, o nosso arquivo web.xml, que você encontra em src/main/webapp/WEB-INF/, está com as configurações da versão 2.3. Você pode encontrar o conteúdo da versão 3.1 [aqui](http://goo.gl/KNu4sj), e substituir o conteúdo do arquivo web.xml pelo seguinte:

<web-app 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/web-app\_3\_1.xsd"

version="3.1">

</web-app>

Na prática, é interessante trabalhar com as versões mais atuais da API de Servlet. Por isso atualizamos essas configurações.

Tente executar um mvn jetty:run e veja se tudo continua funcionando.

**Adicionando o plugin do jetty**

Para adicionar o plugin, basta adicionar as informações no pom.xml. Com o tempo, a versão na documentação pode ser mais atual:

<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/maven-v4\_0\_0.xsd">

<!-- outras infos -->

<dependencies>

<!-- depedências -->

</dependencies>

<build>

<finalName>lojaweb</finalName>

<plugins>

<plugin>

<groupId>org.eclipse.jetty</groupId>

<artifactId>jetty-maven-plugin</artifactId>

<version>9.3.7.v20160115</version>

</plugin>

</plugins>

</build>

</project>

Para executar o goal (como de costume, da primeira vez pode ser que demore um pouco):

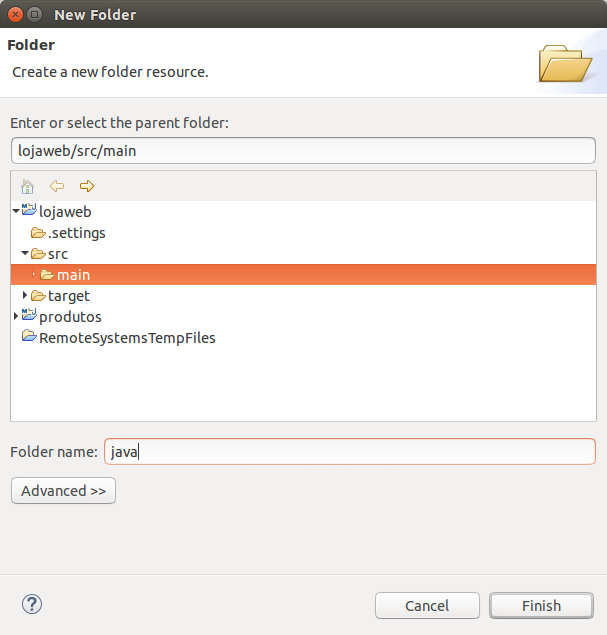
mvn jetty:run

Ao acessar [http://localhost:8080](http://localhost:8080/), você deverá ver a mensagem "Hello World!", que foi definida dentro do arquivo src/main/webapp/index.jsp, do nosso projeto.

**Testando o hot deploy**

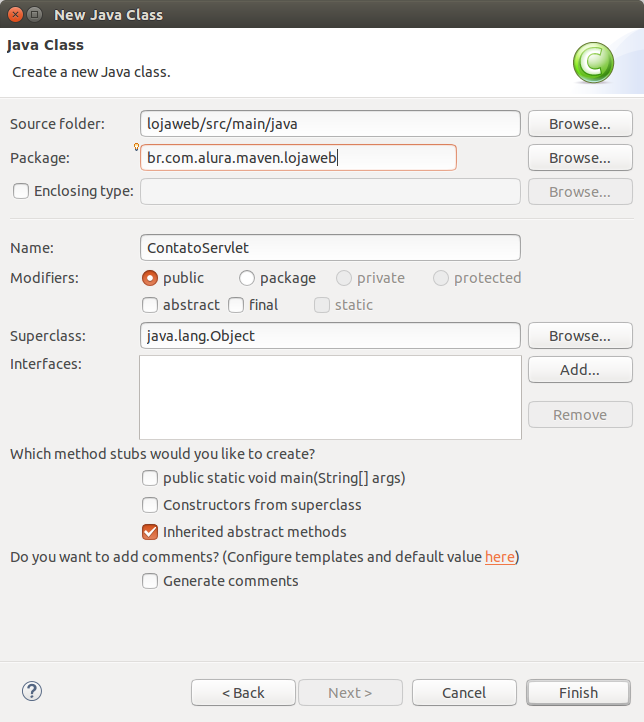
Agora vamos criar uma Servlet para que no próximo exercício seja possível testar algumas outras configurações do *plugin* do Jetty.

Como vimos no projeto anterior (produtos), os códigos das classes ficam no diretório src/main/java. Dentro de src/main, crie java. Clique com o botão direito no diretório main, escolha as opções New > Folder. A seguinte janela será exibida, na qual digitaremos o nome do diretório:



Clique no projeto, no Explorer do Eclipse e utilize a tecla "F5" para atualizar o projeto. A pasta criada deve ser reconhecida como uma source folder dentro de "Java Resources", na perspectiva Java EE do Eclipse.

Na source folder src/main/java, dentro do pacote br.com.alura.maven.lojaweb, crie a Servlet ContatoServlet (Botão direito em src/main/java, "New > Class").



Ela deverá conter o seguinte conteúdo:

package br.com.alura.maven.lojaweb;

import java.io.IOException;

import java.io.PrintWriter;

import javax.servlet.ServletException;

import javax.servlet.annotation.WebServlet;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

@WebServlet(urlPatterns={"/contato"})

public class ContatoServlet extends HttpServlet {

@Override

protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {

PrintWriter writer = resp.getWriter();

writer.println("<html><h2>Entre em contato</h2></html>");

writer.close();

}

}

Caso esteja com o Jetty "startado", encerre utilizando "Ctrl + C". No Windows, após o atalho, ele perguntará se você deseja finalizar o arquivo em lotes. Digite "S" e pressione "Enter".

Configure o Jetty para que as alterações nas classes sejam carregadas automaticamente de tempo em tempo, para evitar que pausemos e iniciemos o servidor sempre que fizermos alguma alteração nas classes. Configure o tempo em 10 segundos. Você pode obter ajuda [aqui](http://www.eclipse.org/jetty/documentation/current/jetty-maven-plugin.html#jetty-run-goal).

Após realizar as configurações, inicie o servidor, acesse localhost:8080/contato e veja a mensagem Entre em contato aparecer. Troque a mensagem na classe e aguarde uns dez segundos. Atualize a página e veja que a mensagem mudou.

Para configurar o *hot deploy* no Jetty, basta inserirmos scanIntervalSeconds no *plugin*. O número representa o intervalo em segundos para o Jetty checar por mudanças.

<plugin>

<groupId>org.eclipse.jetty</groupId>

<artifactId>jetty-maven-plugin</artifactId>

<version>9.3.7.v20160115</version>

<configuration>

<scanIntervalSeconds>10</scanIntervalSeconds>

</configuration>

</plugin>

**Alterando o contexto da aplicação**

Configure o plugin do Jetty para que o contexto da aplicação agora seja /loja. Dessa forma conseguimos ter várias aplicações no mesmo servidor.

Agora, em vez de acessar [http:localhost:8080/contato](http://localhost:8080/contato), você deve acessar [http:localhost:8080/loja/contato](http://localhost:8080/loja/contato).

**Observação:** Você deve parar o servidor ("Ctrl + C") e reiniciá-lo para que a nova configuração seja carregada.

Após testarmos a configuração, você pode removê-lo, assim mantemos o acesso da nossa aplicação através da URL [http:localhost:8080](http://localhost:8080/).

Para alterar o contexto da aplicação, basta adicionar a configuração contextPath:

<plugin>

<groupId>org.eclipse.jetty</groupId>

<artifactId>jetty-maven-plugin</artifactId>

<version>9.3.7.v20160115</version>

<configuration>

<scanIntervalSeconds>10</scanIntervalSeconds>

<webApp>

<contextPath>/loja</contextPath>

</webApp>

</configuration>

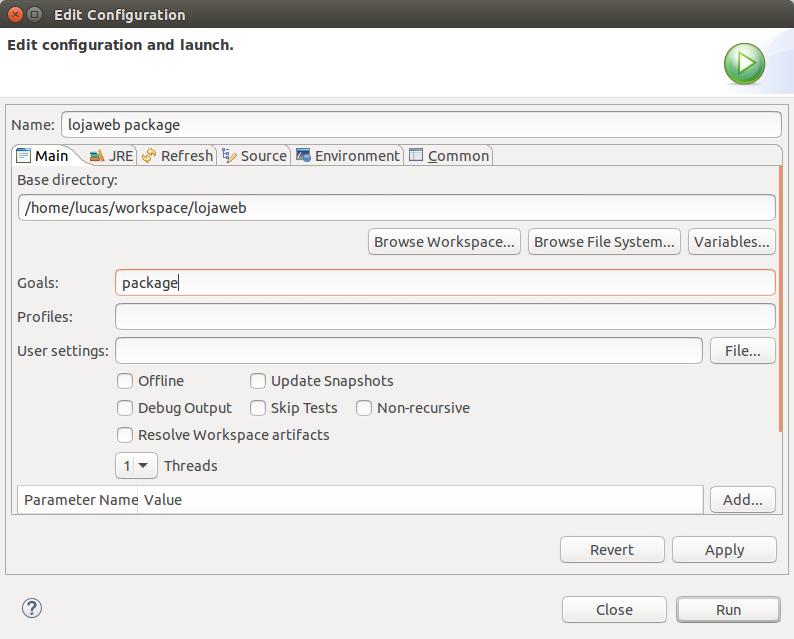
</plugin>

Deste modo não acessamos mais a aplicação a partir do [http://localhost:8080](http://localhost:8080/), que é o padrão caso não alteremos as configurações.

**Adicionando a dependência do Caelum Stella**

Adicione a dependência do Caelum Stella ao projeto lojaweb. As informações podem ser obtidas [aqui](http://mvnrepository.com/artifact/br.com.caelum.stella/caelum-stella-core).

Gere o pacote do projeto por meio do Eclipse clicando com o botão direito no projeto, e em "Run As > Maven build...". Após finalizar as configurações, clique no botão "Run":



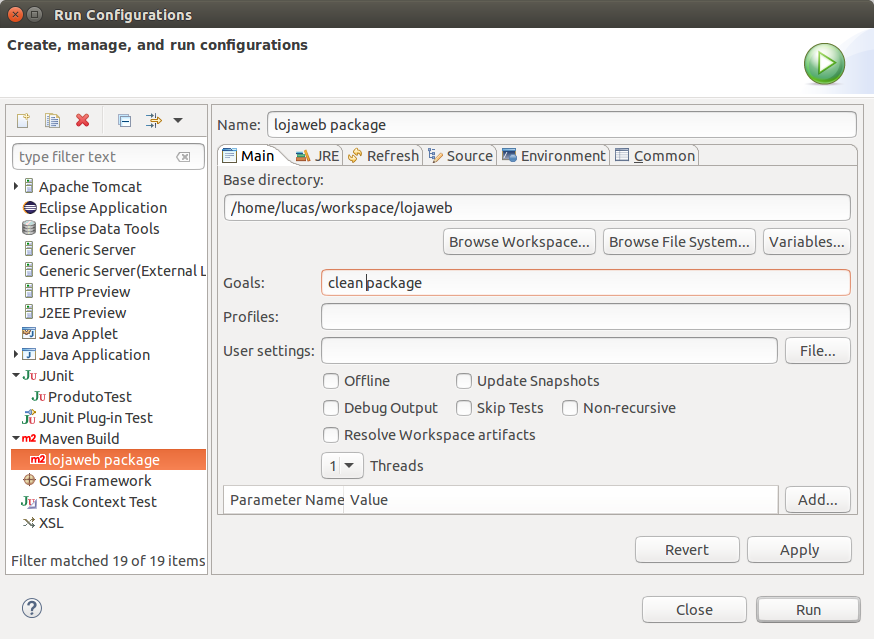
Na pasta do target do projeto, encontraremos o arquivo .war. É possível verificar o conteúdo executando o seguinte comando, a partir do diretório do projeto:

unzip -l target/lojaweb.war

Perceba que nesse momento temos as dependências do Caelum Stella e da API de Servlet no diretório WEB-INF/lib. O .jar do JUnit não se encontra aqui, pois o escopo dele é test, indisponível no artefato final:



Altere o escopo da API de Servlet para provided, gere novamente o pacote e veja quais bibliotecas passam a ficar disponíveis no .war gerado. É provável que a API de Servlet ainda esteja lá. Limpe o diretório target utilizando o clean. Para isso, é possível executar duas fases no mesmo comando. No menu do Eclipse, escolha "Run > Run Configurations". Em "Maven Build", altere o "lojaweb package" e clique no botão "Run":



Vamos adicionar a dependência do Stella, e utilizar a *tag* <scope> para adicionar o escopo provided na API de Servlet:

<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/maven-v4\_0\_0.xsd">

<!-- informações sobre o projeto -->

<dependencies>

<!-- junit -->

<dependency>

<groupId>br.com.caelum.stella</groupId>

<artifactId>caelum-stella-core</artifactId>

<version>2.1.2</version>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>javax.servlet-api</artifactId>

<version>3.1.0</version>

<scope>provided</scope>

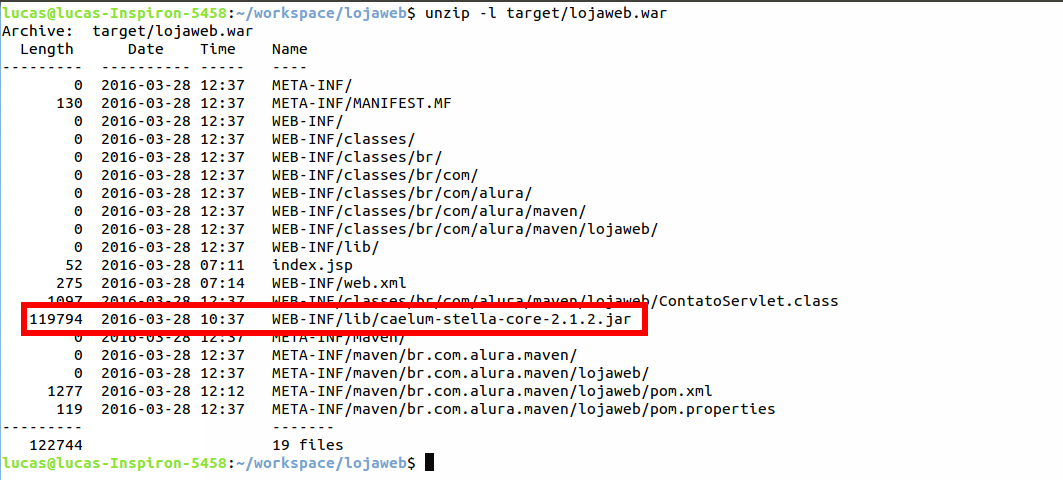
</dependency>

</dependencies>

<!-- build -->

</project>

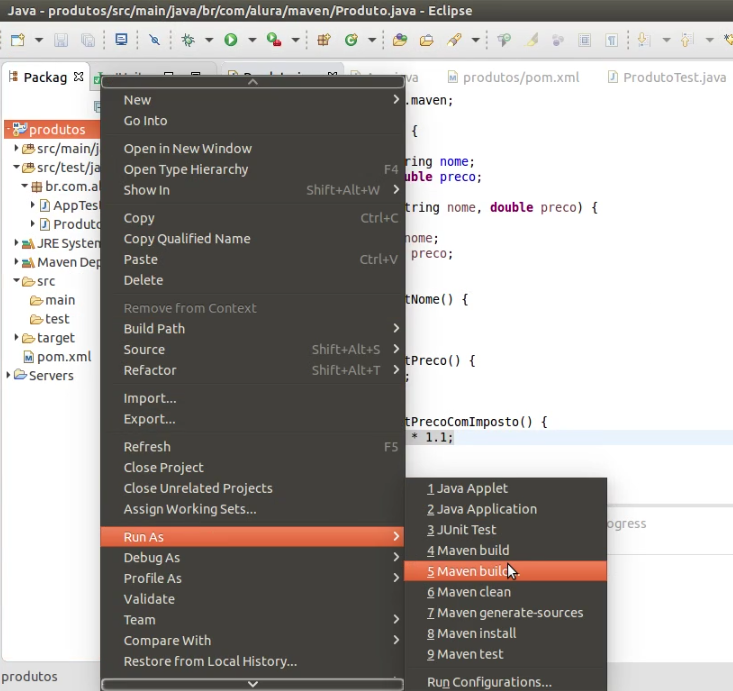
Após a alteração, a API de Servlet não é mais disponibilizada no artefato final, e agora temos apenas a biblioteca Caelum Stella:



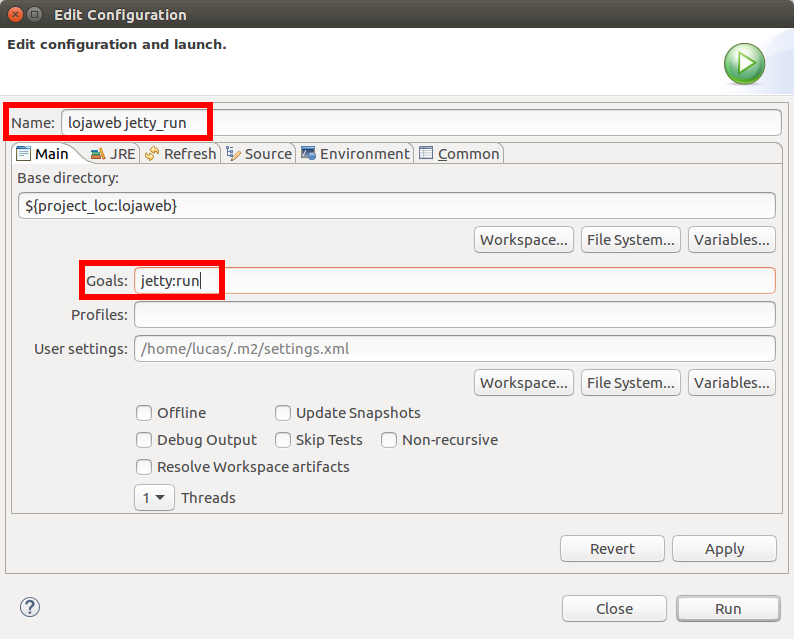
**Executando o Jetty a partir do Eclipse**

Agora vamos realizar as configurações para que seja possível executarmos o Jetty a partir do Eclipse. Lembre-se de que para isso funcionar é necessário que tenhamos executado o comando mvn install no projeto lojaweb.

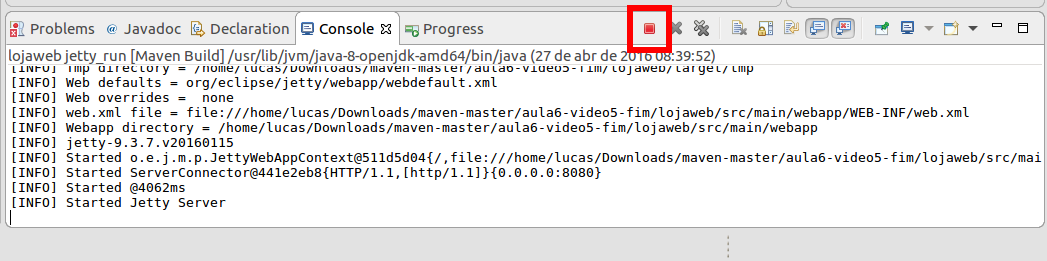
Como já vimos, para configurarmos um *goal* no Eclipse, o primeiro passo é clicar com o botão direito no projeto, e escolher "Run As > Maven build...". Não podemos esquecer que o projeto em que estamos trabalhando agora é o lojaweb.



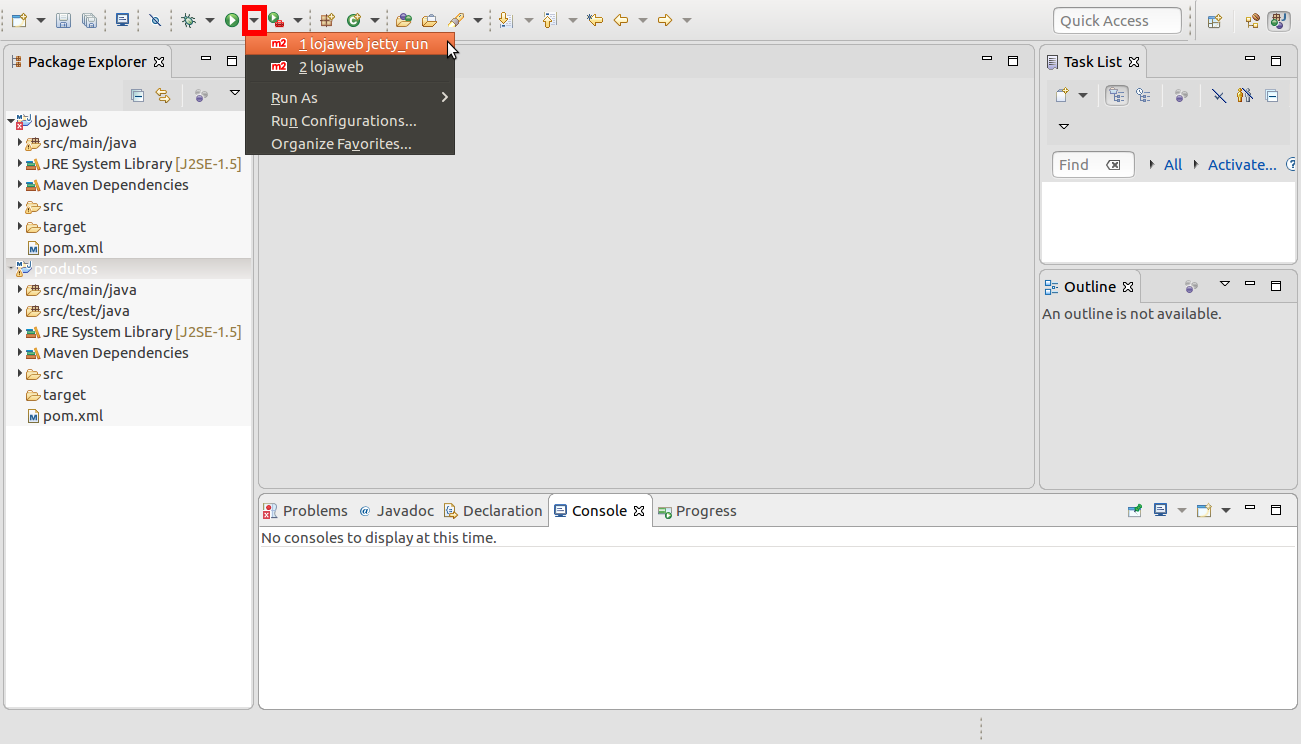
Na próxima tela, vamos configurar o *goals*. Em "Name", vamos escolher um nome significativo para utilizarmos depois. O *goal* será jetty:run.



Finalizada a configuração, basta clicarmos em "Run". Isso irá iniciar o servidor. Para pará-lo, é só clicar no ícone que representa o "Stop", no menu superior do painel que contém a aba "Console".



Criada a configuração, para executá-la novamente, acessamos as opções no botão "Run" e selecionamos o "lojaweb jetty\_run".



**Adicionando o projeto produtos como dependência**

Adicione o projeto produtos como dependência para o projeto lojaweb.

<dependency>

<groupId>br.com.alura.maven</groupId>

<artifactId>produtos</artifactId>

<version>1.0-SNAPSHOT</version>

</dependency>

O problema ocorre porque o artefato não está instalado localmente e não pode ser encontrado nos repositórios remotos. O Eclipse é esperto o suficiente para reconhecer o projeto produtos como um projeto aberto, por isso funciona na IDE. Para resolver o problema, devemos executar o comando mvn install no projeto produtos, que instalará o artefato no repositório local do Maven.

A vantagem de se utilizar essa abordagem do Eclipse é o fato de que quando atualizarmos o projeto produtos, as alterações serão refletidas automaticamente no projeto lojaweb. Normalmente seria necessário instalar novamente o projeto no repositório local para poder utilizá-lo.

A desvantagem é que temos acesso a todas as classes do projeto produtos, inclusive classes de teste que não estarão disponíveis por padrão no artefato final. Podemos acabar nos confundindo e tendo problemas.

Caso o projeto não estivesse aberto na IDE, o processo seria o padrão: o artefato seria buscado localmente ou nos repositórios remotos.

**Entendendo o escopo das dependências**

O próximo passo em nosso curso é entender melhor as dependências e seus escopos. Em nosso arquivo pom.xml do projeto lojaweb, temos a dependência com relação ao Stella, ao projeto produtos, junit, e javax.servlet.

Ao verificarmos a pasta Maven Dependencies nos depararemos com o arquivo xstream-1.4.8.jar, o que nos leva a refletir: por que além das dependências temos o XStream adicionado?

Devemos lembrar que o Maven constrói uma "árvore de dependências", detectando que a dependência produtos precisa do XStream. O arquivo pom.xml de produtos se encontra da seguinte maneira:

<dependency>

<groupId>com.thoughtworks.xstream</groupId>

<artifactId>xstream</artifactId>

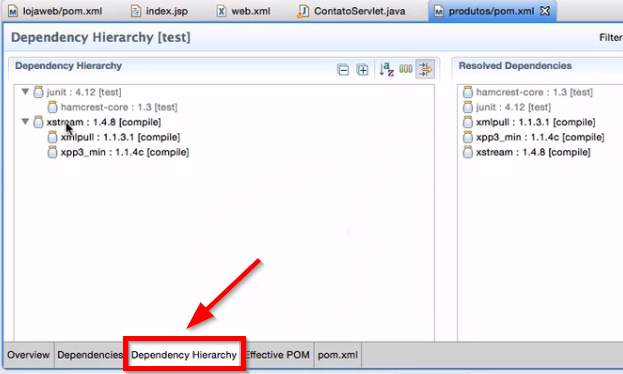
<version>1.4.8</version>

<scope>compile</scope>

</dependency>

O padrão de escopo (<scope>compile</scope>) indica que "será usado para compilar", e essa dependência será repassada para outros projetos interdependentes, por isso o XStream aparece no arquivo Maven Dependecies do projeto lojaweb. Caso modifiquemos a versão do XStream a ser utilizada, essa mudança se refletirá no projeto lojaweb de forma automática.

Podemos observar a árvore de dependências selecionando a opção "Dependency Hierarchy", localizada na parte inferior da área de edição do Eclipse. Conseguiremos visualizar a dependência do JUnit para a realização de testes, e do XStream para compilação.



Ao acessarmos o projeto lojaweb veremos que as dependências são caelum-stella-core, produtos, junit e javax.servlet-api. No caso de produtos, temos a dependência de xstream, que por sua vez depende de xmlpull e xpp3\_min.

Quando acionamos o comando mvn package, será gerado um arquivo .war que conterá **quase** todos estes arquivos citados, com exceção dos de teste. Podemos verificar nosso arquivo .war por meio da linha de comando:

pwd

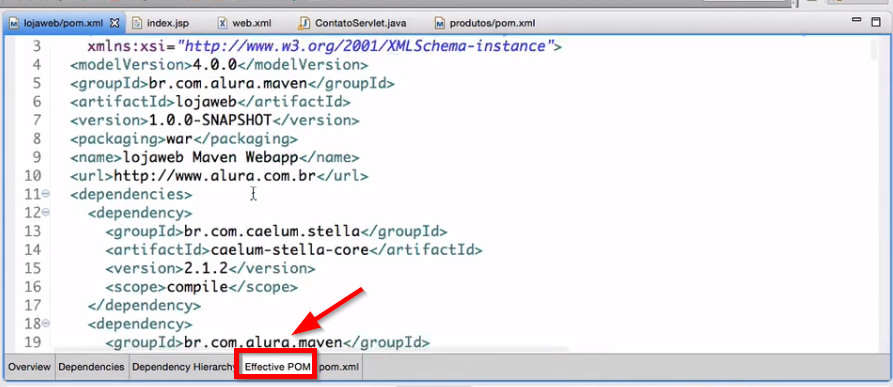
cd ..

cd lojaweb/

unzip -l target/lojaweb.war

Com isso teremos acesso a todas as dependências do projeto, exceto pelos arquivos de teste. O esquema de árvore de dependências do Eclipse é bem interessante, mas o Maven também possui algo semelhante.

Usaremos o comando mvn dependency:tree para baixarmos o plugin de dependência. Uma vez que download for concluído, veremos em nossa linha de comando a árvore de dependências. A organização visual desta árvore no Eclipse é mais interessante, mas o Maven também possui esse função, ainda que simplificada. No Eclipse podemos acessar o pom.xml efetivo (*Effective POM*), opção disponível na parte inferior da área de edição, que exibirá como o arquivo pom.xml de um determinado projeto ficaria caso englobasse todas as dependências possíveis.



Na prática, utilizamos essa versão para compreender algum erro que possa estar ocorrendo em nosso projeto.

Voltemos ao arquivo pom.xml do projeto produtos. Como sabemos, possuímos duas dependências: XStream e JUnit.

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>com.thoughtworks.xstream</groupId>

<artifactId>xstream</artifactId>

<version>1.4.8</version>

<scope>compile</scope>

</dependency>

Faremos uma alteração: modificaremos o <scope> do XStream de compile para test. Lembrando que testes não são incluídos no *package*, portanto devem desaparecer. De fato, os testes não são passados para frente. Voltaremos para o padrão compile e seguiremos explorando nosso projeto. Temos outras opções de escopo; no caso de javax.servlet, veremos que ele está armazenado em lib:

WEB-INF/lib/javax.servlet-api-3.1.0.jar

De acordo com a especificação do Java, não devemos inserir servlet-api em lib, ou seja, ela não deveria estar neste local como dependência, portanto retiraremos o seguinte trecho de código do arquivo pom.xml do projeto lojaweb.

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>javax.servelt-api</artifactId>

<version>3.1.0</version>

</dependency>

Ao fazermos essa subtração, teremos alguns erros em nosso projeto, afinal o Eclipse irá detectar a falta desse elemento. Temos um dilema: queremos a Servlet para compilar, mas em produção sabemos que ela proverá o arquivo .jar. Logo, para a compilação usaremos a Servlet, mas o arquivo .jar deve estar previamente disponível. Para isso usaremos o provided em <scope>.

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>javax.servelt-api</artifactId>

<version>3.1.0</version>

<scope>provided</scope>

</dependency>

Na linha de comando, acionaremos o mvn clean para limpar nosso diretório de trabalho, caso contrário a Servlet ainda aparecerá como dependência. Assim feito, executaremos o *build*. Ao analisarmos o resultado, veremos que o javax.servlet não se encontra mais nas dependências.

Quando removemos dependências, é muito importante utilizarmos o comando mvn clean, evitando resíduos em nosso código que podem gerar problemas na execução.

Conheceremos outra variação de escopo. Podemos querer que o Stella compile apenas para execução, isto é, o inverso do que fizemos com a Servlet. Para isso utilizaremos o runtime, que roda apenas em tempo de execução:

<dependency>

<groupId>br.com.caelum.stella</groupId>

<artifactId>caelum-stella-core</artifactId>

<version>2.1.2</version>

<scope>runtime</scope>

</dependency>

Ao analisarmos a hierarquia de dependências (*Dependency Hierarchy*) veremos caelum-stella-core: 2.1.2 [runtime], isto é, "apenas em execução". Contudo, ele continua visível no arquivo Maven Dependencies — por que isso acontece?

No Eclipse podemos compilar **e** executar códigos, assim como executamos um teste anteriormente. Quando é detectado um *runtime*, e Eclipse o insere no *classpath*, não por motivos de compilação, mas por execução.

Podemos encontrar problemas nesse processo e ter acessos indevidos. Por exemplo, ao tentarmos acessar new CPF(22222222222) em ContatoServlet, esse código será compilado no Eclipse, mas ao acessarmos a linha de comando — depois de acionarmos mvn clean — e utilizarmos o comando mvn compile, teremos que a classe CPF não será encontrada.

O escopo *runtime* funciona perfeitamente na linha de comando, assim como no Eclipse, porém ele cria a possibilidade de escrevermos um código com acessos indevidos. Assim, quando formos rodar o *build* no setor de integração teremos problemas e a compilação não ocorrerá.

Aprenderemos um último caso de dependências, quando temos o XStream como uma dependência do projeto produtos na versão 1.4.1.

<dependency>

<groupId>com.thoughtworks.xstream</groupId>

<artifactId>xstream</artifactId>

<version>1.4.1</version>

<scope>compile</scope>

</dependency>

Ao copiarmos essa dependência para inseri-la no projeto lojaweb, veremos que esse projeto também depende do XStream, mas de uma versão mais recente (1.4.8). Neste caso, será mantida a versão 1.4.8, pois foi um dado definido explicitamente. **Isto é, será dada a preferência para a definição explícita do código.** Essa preferência pode gerar problemas caso estejamos utilizando versões diferentes em outro projeto, por isso precisamos pensar sobre a decisão de declararmos explicitamente.

Outro caso comum: se temos uma biblioteca utilizando um xmlpull e xpp3\_mim mas sabemos que elas não serão necessárias, ou seja, se queremos utilizar o XStream, mas não essas partes. Para excluirmos essas partes que não serão utilizadas, iremos ao arquivo pom.xml e adicionaremos a tag <exclusions>, que abarcará os elementos que desejamos eliminar da biblioteca.

Iremos definir uma exclusão por vez, e no caso deletaremos xmlpull e manteremos xpp3\_mim:

<dependency>

<groupId>com.thoughtworks.xstream</groupId>

<artifactId>xstream</artifactId>

<version>1.4.1</version>

<exclusions>

<exclusion>

<groupId>xmlpull</groupId>

<artifactId>xmlpull</artifactId>

</exclusion>

</exclusions>

</dependency>

O item foi excluído da árvore de dependências, dessa forma podemos ter maior controle do que se passa no projeto. Podemos utilizar dependências de outros projetos, defini-las apenas para execução ou compilação, excluir uma dependência intermediária, utilizar versões mais antigas de uma biblioteca, e assim por diante.

**Escopo provided**

O escopo provided indica que precisamos da dependência para compilar e testar, mas não se faz necessário incluir a dependência no artefato final, pois o JDK ou o *container* irão disponibilizar a dependência.

Um exemplo disso é a nossa API de Servlet, declarada como dependência na lojaweb. O *container* (Jetty) irá disponibilizar a implementação dessa API, portanto não é necessário que a biblioteca esteja disponível junto com o .war da projeto.

**Escopo compile**

O escopo compile indica que a dependência estará disponível em todos os *classpaths*: de compilação de teste e execução. Esse é o escopo padrão utilizado quando não declaramos um escopo na dependência.

Nesse momento, a dependência javax.servlet-api, por exemplo, não tem nenhum escopo declarado explicitamente, portanto ele é compile.

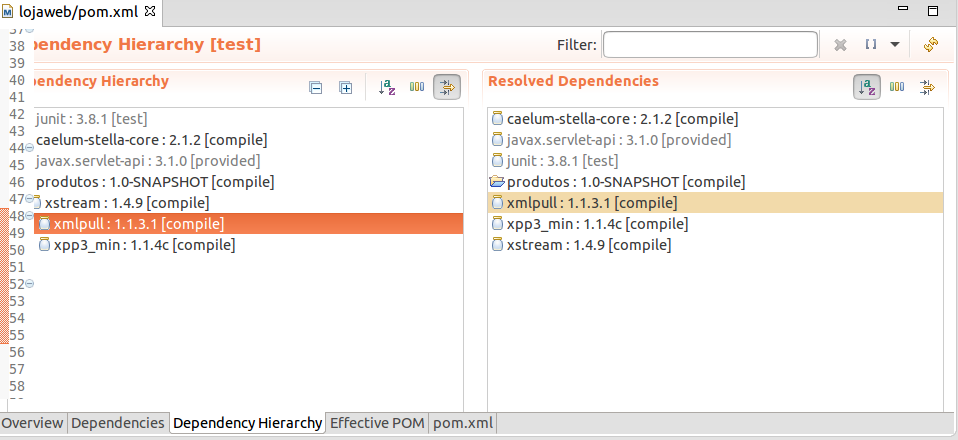
**Escopo runtime**

O escopo de runtime indica que **não** necessitamos da dependência para compilar. A dependência estará disponível no *classpath* de *runtime*, o que significa que utilizaremos a dependência na execução do projeto. A dependência também estará disponível no *classpath* de testes.

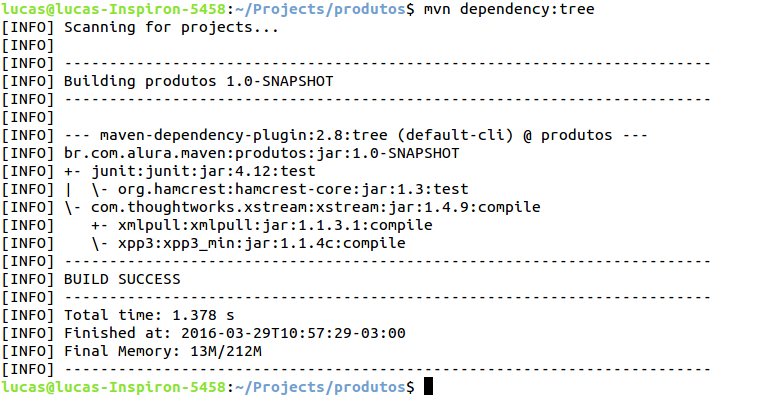
**Excluindo dependências**

Em "Java Resources > Libraries > Maven Dependencies", há algumas bibliotecas que não declaramos como dependências no pom.xml. Elas são resolvidas pelo Maven, pois cada dependência que adicionamos pode conter outras.

No Eclipse, é possível consultar a hierarquia de dependências clicando na aba "Dependency Hierarchy". Podemos ver, por exemplo, que o nosso projeto depende de produtos, e que este, por sua vez, depende do xstream. O xstream depende do xmlpull e do xpp3\_min:



Também é possível analisar a hierarquia de dependências utilizando o comando mvn dependency:tree:



Em alguns momentos, podemos querer excluir uma determinada biblioteca que seja dependência de uma biblioteca do nosso projeto. Um motivo é que as dependências em uma das que estamos utilizando pode ter sido declarada errada, por exemplo.

Exclua o xstream do produtos no projeto lojaweb.

Para excluir uma dependência que foi incluída em algum projeto que utilizamos, basta utilizar a *tag* exclusion:

<dependency>

<groupId>br.com.alura.maven</groupId>

<artifactId>produtos</artifactId>

<version>1.0-SNAPSHOT</version>

<exclusions>

<exclusion>

<groupId>com.thoughtworks.xstream</groupId>

<artifactId>xstream</artifactId>

</exclusion>

</exclusions>

</dependency>

Com isso você percebe-se que o xstream deixa de ser uma biblioteca do projeto lojaweb.

Nesse caso em específico, em que temos acesso ao código fonte do projeto produtos, o melhor a se fazer seria remover o xtream do pom.xml caso não estivéssemos de fato utilizando a biblioteca.