07

Entendendo o escopo das dependências

Transcrição

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`.

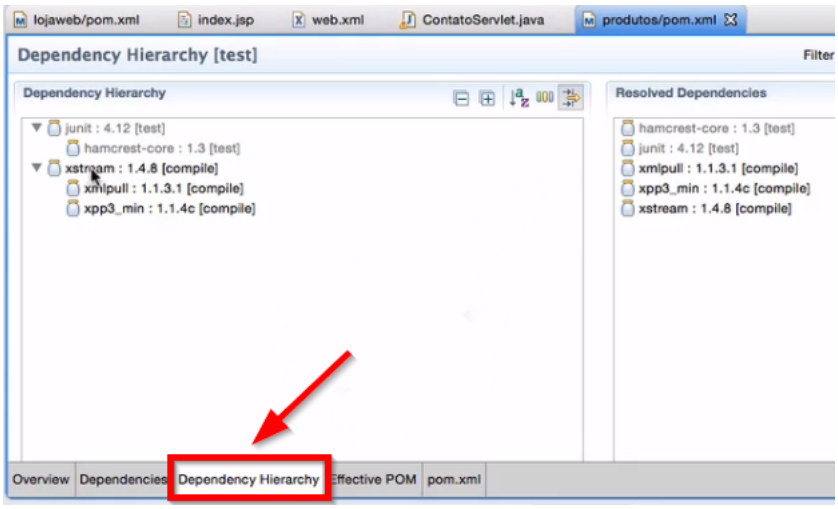
Ao verificarmos a pasta `Maven Dependencies` nos depararemos com o arquivo `xstream-1.4.8.jar`, o que nos leva a refletir: por que além das dependências temos o XStream adicionado?

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.



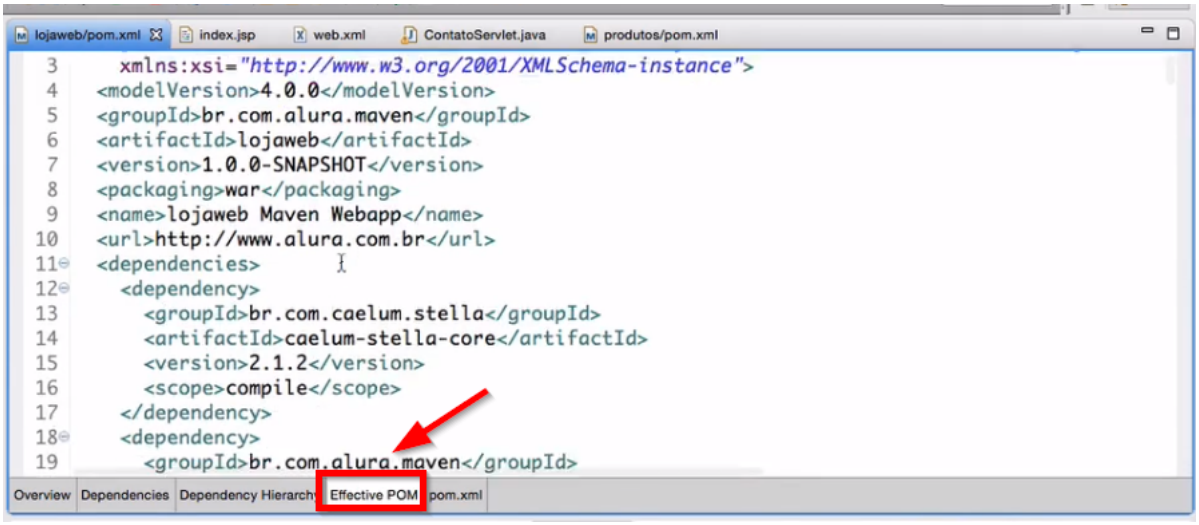
Ao acessarmos o projeto `lojaweb` veremos que as dependências são `caelum-stella-core`, `produtos`, `junit` e `javax.servlet-api`. No caso de `produtos`, temos a dependência de `xstream`, que por sua vez depende de `xmlpull` e `xpp3_min`.

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.