01 Projeto da aula anterior

Caso queira, você pode baixar aqui o projeto do curso no ponto em que paramos na aula anterior.

02 Processo de build no Maven

Transcrição

[00:00] Na última aula aprendemos sobre um dos pilares do Maven, essa parte de gerenciamento de dependências. Aprendemos a adicionar dependências no projeto, a pesquisar por essas dependências no repositório central do Maven e aprendemos também essa questão de repositórios, o *cache* local, repositório local.

[00:20] Esse era um dos grandes pilares no Maven, gerenciamento de dependências, conforme tínhamos discutido. Um outro pilar essencial para uma aplicação é a questão do *build*, como que eu faço para buildar a aplicação, para compilar, modificar arquivos, criar diretórios e gerar o pacote do projeto para fazer o *deploy* e ele entrar em produção.

[00:41] Nessa aula o objetivo vai ser esse, aprender sobre o *build* do projeto. Na realidade, o Maven já tem embutido essa questão do *build*, podemos executar o *build* da aplicação pelo pela IDE, pelo Eclipse, ou pelo *prompt* de comando, no terminal.

[00:58] Vou fazer nos dois casos porque essa parte de rodar comandos para o *build* também é comum de ser executada pelo *prompt* de

comando. No prompt, eu estou no diretório da minha aplicação. Se eu digitar um `ls` para listar os arquivos, tem o `pom.xml`, o `src` e aqui lembra tem aquele `mvn`.

[01:17] Se você já estiver adicionado no *path* do sistema operacional como variável de ambiente, diretório onde descompactou o Maven, você já consegue rodar pelo `mvn` diretamente, sem passar o caminho completo da pasta `bin` do Maven.

[01:29] `mvn` e o que você passa é um objetivo, que é chamado de *goal*. Existem vários objetivos, várias tarefas que o Maven pode executar em cima do seu projeto relacionadas com o *build* da aplicação.

[01:40] Por exemplo: tem o `mvn compile`, serve para compilar. Eu estou falando "Maven compila" e aí ele vai procurar no diretório onde eu estiver, tem o `pom.xml`. Vai encontrar as configurações do projeto, vai encontrar os arquivos e fazer a compilação. Se eu usar a tecla “Enter”, ele começará o processo de compilação. Ele encontrou o nosso projeto “loja”, executou e compilou com sucesso.

[02:07] Se dermos uma analisada no `pom.xml`, em nenhum lugar que eu disse nada sobre *build*, sobre versão de Java e tudo mais. Lembra que tinha aquele comando que eu tinha comentado com vocês, que às vezes você precisa atualizar o projeto? Você clica com o botão direito do mouse no projeto “loja > Maven > Update Project” e nós podemos simular isso no *prompt* de comando.

[02:29] Se executarmos um `mvn clean`, que é um outro *goal* (objetivo) do Maven, que é só para limpar o diretório *target*, limpar os arquivos e

deixar o terreno preparado, se você quisesse compilar. Vamos executar um `clean`. É sempre importante limpar, você já tinha arquivos lá na *target*, talvez ele pule alguma etapa.

[02:49] Agora se eu executar um `mvn compile`, vamos ver o que vai acontecer. Como eu já havia executado antes, tinha dado certo. Agora deu um erro, que era o erro que eu estava esperando.

[03:00] Ele falou: "O Java 5 não é mais suportado, tem que ser pelo menos o Java 6". Temos um problema, ele está, por padrão, configurado que o Java está na versão 5. Em nenhum lugar do `pom.xml` dissemos qual é a versão do Java.

[03:04] Então tem um padrão que ele pega como Java 5. Só que para configurar isso podemos adicionar no `pom.xml`, eu vou colar que eu já tenho salvo só para não perder muito tempo. Existe essa *tag* `<build>`, que é justamente utilizada para configurarmos coisas do *build* e coisas opcionais.

Código `pom.xml`:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"


//código omitido


<build>
        <plugins>
            <plugin>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
```

```
                    <source>11</source>
                    <target>11</target>
                </configuration>
            </plugin>
        </plugins>
</build>

</project> COPIAR CÓDIGO
```

[03:32] Tem `<plugins>`, que depois vamos ver com calma. Existe esse *plugin* chamado `maven-compiler-plugin`, que é para configurarmos qual é a versão do Java. Tem essa *tag* `<configuration>`, o código-fonte está na versão 11 e é para ele compilar e gerar os arquivos binários na versão 11 também.

[03:50] Agora, eu já adicionei esse *plugin* e com isso eu já ensinei ao Maven qual é a versão do Java utilizada na aplicação. Se rodarmos novamente no *prompt*, podemos dar um `mvn clean` e podemos passar mais de um *goal* ao mesmo tempo. É só escrever nome do *goal* + "Espaço" + nome do segundo *goal*.

[04:10] Eu posso passar um *compile*, `mvn clean compile` - ele vai executar um *clean* e se tudo der certo ele vai para o próximo *goal* - que no caso é o `compile`. Vai fazer um *clean* do projeto. E agora compilou tudo certo, detectou a versão do Java 11 e apareceu essa mensagem: “BUILD SUCCESS”. Gerou o *build* com sucesso, ele fez a tarefa que você mandou ele fazer que é dar um *clean* e um *compile*.

[04:29] Cadê as classes compiladas? Como é que eu verifico isso? Se você olhar em cima no terminal, informa que compilou um *source file*. O projeto só tem uma única classe, para o diretório “loja/target”.

[04:44] Lá na pasta, entramos em “eclipse-workspace > loja > target”, tem esse diretório *target*, que é onde ele compila. Todo *build* vem para cá. Tem `classes`, tem aquele arquivo “messages.properties”, que estava no diretório “source/main/resources”, que tinha criado de exemplo. Ele jogou para cá faz parte da compilação. Ele criou pastas ""“br” > “com” > “alura” > “loja”, está em "loja" o arquivo `.class`, ele compilou para cá.

[05:13] Dentro do diretório *target* é onde ele executa, joga os artefatos, tudo o que foi gerado no processo de *build*. Em `pom.xml` na parte do código que acabamos de colar poderíamos configurar outras informações também, tudo que for relacionado com *build* configuramos na *tag* `<build>`.
[05:27] Outros *goals* que existem e são importantes - além do *clean* e do *compile*, eu tenho testes automatizados na aplicação. Quero executar os meus testes, eu posso executar `mvn test`. Ele vai verificar se tem algum teste automatizado com `JUnit` na aplicação, vai executar os testes e dizer se passou ou se falhou.

[05:46] No caso falhou e em cima ele te dá um relatório, resultado, em *Results*.

[05:51] Ele rodou um teste e esse único teste falhou. Ele te informa qual foi o teste que falhou, foi na classe produto teste e a mensagem “Not yet implemented”.

[05:59] Isso foi porque tínhamos deixado de propósito uma classe de teste de exemplo em `ProdutoTest.java` que está com um *fail* , `fail("Not yet implemented")`, então ele vai falhar. Vou só remover essa parte de `fail`, não vou escrever um teste, só para emular. Tem um teste, vamos rodar um `mvn test` novamente para ver. Ele vai executar e agora foi com sucesso.

[06:20] Só que perceba que antes de executar essa tarefa de testes, ele imprimiu um monte de coisas. Você não precisa ficar preocupado com tudo isso, mas ele fala algumas coisas do *build* e de *plugin*. De *plugin* pode ignorar, mas ele fala. Nessa parte é que começou a brincadeira, "Building loja 1.0.0" ele começou a fazer o *build* do "loja". É aquela versão do projeto que tínhamos configurado e aí ele executou esse *plugin*, `maven-resources-plugin:2.6:resources`, para pagar os *resources* do projeto.

[06:48] E rodou o Maven *compile*. Ele fez um *compile* antes, embora só tenha executado `mvn test`, o *compile* é pré-requisito do teste, para nós rodarmos os testes precisamos compilar o código-fonte primeiro. [06:58] As classes de testes vão utilizar as nossas classes, elas precisam estar compiladas. Depois é que ele veio aqui, gerou os *resources* de testes, compilou as classes de testes e executou os testes. Aí vem um relatório dizendo se passou ou se falhou.

[07:14] Agora no caso, rodou 1, não deu nenhuma falha: "Tests run: 1, Failures: 0, Errors: 0, Skipped: O, Time elapsed: 0. 094 sec". Tem um relatório dos testes ele te mostra em quantos segundos rodou e o *build* foi executado com sucesso, "BUILD SUCCESS".

[07:25] Já aprendemos como compilar o projeto, executar os testes e fazer um *clean*. E como é que eu faço para gerar o *build* de fato. Isso vai ser assunto do próximo vídeo, nós vamos aprender como gerar o JAR, o WAR da aplicação e fazer algumas customizações em relação ao *build* do projeto. Isso veremos no próximo vídeo. Vejo vocês lá, um abraço!

03 Goals do Maven

Vimos que podemos executar o build da aplicação com o comando `mvn goal`, sendo **goal** o nome da *tarefa* que queremos solicitar ao Maven para executar.

Qual o objetivo do goal **test**?

- Alternativa correta

  Testar se as dependências foram baixadas com sucesso

- Alternativa correta

  Testar se o projeto está compilando com sucesso

- Alternativa correta

  Executar os testes automatizados da aplicação

  Alternativa correta! O comando `mvn test` executa os testes automatizados da aplicação, apresentando ao final um relatório.

04 Gerando o artefato do build

Transcrição

[00:01] Agora que já aprendemos os comandos do Maven para fazermos o processo de *build*, de compilação, executar os testes e fazer o *clean* do projeto; chegou a hora de vermos essa questão do *build* em si, de gerar o pacote da aplicação.

[00:14] Para fazer isso existe um comando justamente chamado `mvn package` ele vai empacotar a aplicação. Ele vai analisar no `pom.xml` as configurações para ver se ele tem que gerar um JAR ou se tem que gerar um WAR. Baseado nisso, ele faz a criação do pacote. Vamos executar no *prompt* o `mvn package` e ver o que vai acontecer.

[00:34] Ele começou - e aí perceba que ele executou um monte de coisas no terminal. Ele copiou *resources*, fez a compilação, copiou as *resources* de testes, compilou os testes, executou os testes e fez o *plugin* para gerar o JAR, já que é uma aplicação Java *standalone* e ele gerou o JAR da aplicação. E retornou "BUILD SUCCESS", ou seja, ele gerou com sucesso esse JAR.

[01:01] Já sabemos, ele joga naquele diretório *target*, por padrão. É nesse diretório que ele vai jogar os arquivos do *build*. Se entramos naquela pasta, estará lá no "workspace > loja > target". Tem um monte de diretórios das coisas que ele gera no meio do caminho, mas está aqui `loja-1.0.0.jar`. Ele gerou o JAR da aplicação corretamente.

[01:26] Bastaria pegarmos esse JAR agora e passarmos para equipe de infraestrutura para fazer o *deploy* e colocarmos essa aplicação em produção. Perceba que é bem simples de fazer o *build* da aplicação, é só executar esse comando no terminal.

[01:39] E tem os *goals*, `clean` e tem o `compile` para compilar o código-fonte, `test` para compilar e executar as classes de testes, `package`. Se for o caso, se você quiser ao invés de gerar só no diretório *target*, tem um outro comando também, um outro *goal* - que é o `*install`. Ele depende de todos esses outros.

[02:01] O `install` vai fazer todo esse passo a passo, vai gerar o JAR e ele vai instalar. “Instalar” significa que ele vai jogar naquele repositório local, naquele nosso *cache* local.

[02:14] Vamos verificar isso. Lembra que no diretório “Home” do seu usuário, basta utilizar o atalho “Ctrl + H” para exibir os diretórios ocultos, pasta “.m2 > repository”. Aí ele criou “br > com > alura > loja > 1.0.0" e está aqui o JAR `loja1.0.0.jar`. O `install`, ele instala localmente no seu repositório local.

[02:30] Tem também um outro comando, que é o *Deploy*. Poderíamos rodar no *prompt*, `mvn deploy`. Ele vai fazer algo parecido, vai executar todo esse passo a passo, só que o `deploy` não vai jogar no nosso repositório local, ele vai tentar jogar em algum repositório remoto. Só que ele deu erro porque não temos configurado um repositório local. Precisamos passar alguns parâmetros e tudo mais.

[02:54] Isso é feito lá no arquivo `pom.xml`, aqui naquela *tag* `<repositories>`. Só que é uma configuração mais avançada, que você tem que passar outros parâmetros e precisa ter permissão também para jogar um arquivo nesse diretório. É um pouco mais complicado. Não é o caso de um repositório remoto só para baixar as dependências.

[03:13] Seria possível fazer o *deploy* implantar o pacote que foi gerado em um repositório remoto. Por exemplo: se você usa o Nexus, que eu tinha comentado, a ferramenta para gerenciamento de artefatos, poderia usar o Nexus e esse *deploy* seria feito no repositório do Nexus

e isso ficaria disponibilizado esse projeto para outros times acessarem. Mas vai fugir do escopo do treinamento.

[03:37] Perceba que é bem simples. Lá no Eclipse na *tag* `build` além de ter essa questão de *plugins*, que vamos ver mais para frente, tem algumas outras *tags* e algumas outras configurações que podemos fazer.

[03:50] Por exemplo: tem uma *tag* chamada "finalName". Vocês viram, vamos voltar para o diretório *target* nas pastas do computador, “workspace > loja > target”. Por padrão o nome do projeto é “loja”. De onde que ele pegou essa "loja"? É o *name* do `artifactId` no arquivo `pom.xml` e ele coloca `loja-1.0.0.jar`.

[04:13] Nessa parte do código eu não disse qual é o tipo de empacotamento. Por padrão é JAR, tem a *tag* chamada "packaging". Por padrão, se você não colocar é JAR. Se fosse uma aplicação web bastaria colocarmos WAR aqui, `<packaging>war</packaging>`.

[04:24] Eu quero trocar esse nome, não quero colocar o nome da versão na pasta. Tem a versão 1.0, tem o versionamento, mas na hora de gerar o JAR não quero que tenha versão. Eu quero que seja simplesmente “loja”. Basta você colocar essa *tag*: `<finalName>loja</finalName>`.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"


//código omitido
```

```
<build>
        <finalName>loja</finalName>
        <plugins>
            <plugin>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <source>11</source>
                    <target>11</target>
                </configuration>
            </plugin>
        </plugins>
</build>

</project> COPIAR CÓDIGO
```

[04:42] Vamos executar novamente o Maven e rodar o *build* no *prompt*. Só que aí vamos executar um *clean* para ele limpar esse diretório *target*, `mvn clean package`. Ele vai rodar o *clean*, vai recompilar tudo, rodar os testes e gerar o pacote da aplicação.

[05:00] Se entrarmos no diretório *target* nas pastas, está lá "loja.jar". Não tem mais a versão.

[05:07] Nessa parte do código em `pom.xml` consigo personalizar algumas coisas - tem algumas *tags* do diretório, não serem o diretório *target*. Configurações caso você não esteja seguindo aquela convenção do Maven do "src/main/java", "src/main/resource", você consegue passar onde que é *source directory* , onde que é o diretório de teste.

[05:27] Não vamos precisar disso porque estamos seguindo as convenções do Maven. Por isso que é uma boa prática. Se você seguir as convenções do Maven, é menos coisas para configurar no seu `pom.xml`.

[05:38] Com isso aprendemos como fazemos o *build*, como funciona essa parte do Maven que tem esses comandos aqui do `mvn`. Só que eu fiz no *prompt* de comandos porque essa parte é mais comum você executar no *prompt* de comandos. Mas dá para fazer pelo Eclipse também. Vou mostrar para vocês.

[05:57] Poderíamos vir na "loja". Ele está retornando algum erro agora. Às vezes acontece esse erro aqui no Maven, já esteja ciente. Um erro na linha 1.

[06:07] Você passa o mouse na linha 1 ele fala que está faltando um arquivo, às vezes informa esse erro nada a ver. É bom sempre executar aquela atualização “loja > Maven > Update Project > OK”, só para ver se vai limpar, se vai parar com esse erro. No caso, resolveu. Às vezes acontece um erro na linha 1, é só executar um "Update Project", que provavelmente vai resolver.

[06:25] Como é que eu faço para rodar o *build* pelo Eclipse: clique com o botão direito no projeto: “loja > Run As” e ele já te mostra algumas opções: “Maven clean”, “Maven install”, “Maven test” ou “Maven build...” etc.

[06:41] Eu quero executar um *compile*, não tem um *compile*. Você clica no “Maven build...” e ele vai abrir uma janela e nela tem esse campo "Goals" e você diz qual é o *goal* que você quer executar.

[06:51] Vou executar um “clean compile" e em seguida clico no botão "Run". Ele executa dentro do Eclipse. Você pode abrir o console do Eclipse do lado direito e ele fará aquele mesmo processo que vimos no terminal, vai imprimir todo o passo a passo. Aí aparece “BUILD SUCCESS”.

[07:09] Você pode acessar a pasta *target* no *package explorer*. Gerou todos os arquivos. Dá para fazer isso de dentro do Eclipse também, embora seja mais prático fazer pelo *prompt* de comandos.

[07:21] Esse era o objetivo dessa aula. Aprendemos como geramos o artefato, o *build* do projeto e como configuramos. Tem a parte de *plugins*, que poderiam personalizar esse *build* - que depois vamos ver - e personalizar o nome do artefato.

[07:37] O *build*, o empacotamento é bem simples e tranquilo de realizar. Você não precisa executar passo a passo cada um daqueles *goals*, você pode rodar simplesmente um `mvn install` e pronto. Ele já vai executar tudo. Se o teste tiver teste automatizado falhar ele interrompe o *build*. Tem tudo isso aí configurado.

[07:55] Assim nós fechamos mais um pilar do Maven. São os dois pilares principais que eu considero do Maven: gerenciamento de dependências, que já vimos na aula anterior; e nessa aula *build* do projeto, como executar essas tarefas de compilar, executar testes e gerar o pacote da aplicação.

[8:13] Na próxima aula, vamos aprender outros recursos do Maven, algumas coisas adicionais e opcionais, porque o principal já vimos. Vejo vocês no próximo vídeo para aprender esses outros recursos.

05 Empacotando a aplicação

Vimos que o comando `mvn package` solicita ao Maven que compile o projeto, execute os testes automatizados e então gere o artefato de build da aplicação.

Onde podemos encontrar, por padrão, o artefato gerado pelo Maven?

- Alternativa correta

  No diretório *.m2* do computador

- Alternativa correta

  No diretório *target* da aplicação

  Alternativa correta! Esse é o diretório padrão onde o Maven gera o artefato.

- Alternativa correta

  No repositório central do Maven

06 Faça como eu fiz

Chegou a hora de você seguir todos os passos realizados por mim durante esta aula. Caso já tenha feito, excelente. Se ainda não, é importante que você execute o que foi visto nos vídeos para poder continuar com a próxima aula.

Opinião do instrutor

-
-

Continue com os seus estudos, e se houver dúvidas, não hesite em recorrer ao nosso fórum!

07 O que aprendemos?

Nesta aula, aprendemos:

- A realizar o build da aplicação com Maven;
- A utilizar o comando `mvn nome-do-goal` para realizar o build;
- Alguns *goals* do Maven, como `compile`, `test` e `package`;
- A personalizar o artefato de build gerado pelo Maven.