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 nosso 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 nosso 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 aqui 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. Eu estou aqui 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 aí o que você passa é um objetivo, que é chamado de *goal*. Aí 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 aí compilou com sucesso.

[02:07] Se dermos uma olhada aqui 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 aqui 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` aqui. 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: “olhe, o Java 5 não é mais suportado, tem que ser pelo menos o Java 6”. Aí está um problema, ele está, por padrão, configurando que o Java

está na versão 5. Em nenhum lugar aqui do "pom.xml" dissemos qual é a versão do Java.

[03:04] Então tem um padrão aqui que ele pega como Java 5. Só que para configurar isso podemos adicionar aqui, eu vou colar aqui 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.

[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. Aqui 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 aqui esse *plugin* e com isso eu já ensinei ao Maven qual é a versão do Java utilizada na aplicação. Se rodarmos novamente, podemos dar um `mvn clean` e aí 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 aqui 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 aqui: “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 aqui em cima no terminal, ele falou que compilou um *source file*. O nosso projeto só tem uma única classe, para o diretório “loja/target”.

[04:44] Lá na nossa pasta, entramos aqui “eclipse-workspace > loja > target”, tem esse diretório *target*, que é onde ele joga as coisas, onde ele compila. Todo *build* vem para cá. Aqui 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*. Aqui poderíamos configurar outras informações também, tudo que for relacionado com *build* configuramos aqui 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 aqui em cima ele te dá um relatório, resultado.

[05:51] Ele rodou um teste e esse único teste falhou. Aqui ele te diz 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 aqui uma classe de teste de exemplo `ProdutoTest.java` que está com um *fail* aqui, `fail("Not yet implemented")`, então ele vai falhar. Vou só tirar isso daqui, 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 aqui do *build* e de *plugin*. De *plugin* pode ignorar, mas ele fala aqui. Aqui é que começou a brincadeira, "Building loja 1.0.0" ele começou a fazer o *build* do "loja". Aqui é aquela versão do projeto que tínhamos configurado e aí ele executou esse *plugin* aqui, `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!