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 nossa aplicação. Ele vai olhar 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 aqui o `mvn package` e ver o que vai acontecer.

[00:34] Ele começou - e aí perceba que ele executou um monte de coisas aqui 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 aqui 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 aqui no diretório *target*, tem um outro comando também, um outro *goal* - que é o `*install`. Ele depende de todos esses outros.

[02:01] O que o `install` vai fazer? Ele vai fazer todo esse passo a passo, vai gerar o JAR e ele vai instalar. "Instalar" significa que ele vai jogar naquele nosso 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 aqui, `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 aqui 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 aqui 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 aqui do escopo do treinamento.

[03:37] Perceba que é bem simples. Lá no Eclipse aqui 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 aqui para o nosso diretório *target*, "workspace > loja > target". Por padrão o nome do projeto é "loja". De onde que ele pegou esse "loja"? É o *name* aqui do `artifactId` no arquivo "pom.xml" e ele coloca `loja-1.0.0.jar`.

[04:13] Aqui 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 aqui. 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>`.

[04:42] Vamos executar novamente o Maven e rodar o *build*. Só que aí vamos executar um *clean* para ele limpar aqui 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 lno diretório *target*, está lá "loja.jar". Não tem mais a versão. Percebe?

[05:07] Aqui eu 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 aqui no seu “pom.xml”.

[05:38] Com isso aprendemos aqui como fazemos o *build*, como funciona essa parte aqui do Maven que tem esses comandos aqui do `mvn` . Só que eu fiz aqui 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 aqui no “loja”. Ele está dando algum erro aqui agora. Às vezes acontece esse erro aqui no Maven, já esteja ciente. Um erro na linha 1.

[06:07] Você passa o mouse aqui 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. Aqui 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* aqui. 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". Aí ele executa de dentro do Eclipse. Você pode abrir o console do Eclipse 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.