03

Build e dependências em uma aplicação Java

Transcrição

[00:00] Agora que já vimos o que será abordado aqui no nosso treinamento de **Maven**, vamos entender um pouco sobre qual é a motivação da utilização do Maven, o que ele veio para resolver.

[00:11] Eu estou com a minha IDE aqui aberta, estou usando o eclipse, mas você pode usar a IDE de sua preferência. Eu tenho um projeto de exemplo, uma aplicação Java web tradicional. Vamos considerar que essa aplicação está pronta. Aqui tem várias classes que foram criadas e estão lá os "controllers", "dao". Toda a parte implementada.

[00:35] Agora, eu preciso gerar um *build* desse projeto, gerar um pacote para passar para a galera da infraestrutura para fazer o *deploy* e colocar em produção, colocar no ar essa aplicação.

[00:46] Vem aquela questão: “como é que eu faço para gerar um WAR? Já que é aplicação web. Como que eu gero um WAR dessa aplicação?” Como não tem nenhuma ferramenta, podemos fazer isso diretamente pela IDE, diretamente pelo Eclipse. Eu poderia simplesmente chegar aqui e clicar com botão direito no projeto “controle-products > Export >” filtrar “WAR file” e ir clicando em “Next” em “Destination” escolher onde quero salvar o WAR e epois "Next > Finish”.

[01:09] Porém, nem sempre é tão simples e fácil assim de você gerar um *build* do projeto. Às vezes, você tem que fazer algumas coisas antes disso, você tem um passo a passo, tem várias tarefas para serem executadas antes de gerar o WAR ou JAR da sua aplicação.

[01:27] Por exemplo: essa aplicação tem testes automatizados. Antes de gerar o JAR ou WAR da aplicação, eu preciso rodar esses testes, ver se está tudo passando, se está tudo funcionando, e não gerar o JAR se tiver algum teste falhando. Antes de gerar o WAR, eu tenho que lembrar de rodar os testes.

[01:45] Eu teria que vir aqui, clicar com o botão direito do mouse em "test > Run As > JUnit test" e rodar os testes aplicação. Só se estiver tudo passando, eu gero o WAR da aplicação. Além disso, há outra coisa que eu tenho que fazer também. Aqui tem umas propriedades do Hibernate, a biblioteca para fazer integração com banco de dados.

[02:04] Só que essas configurações estão apontando para o banco de dados local. Para gerar o WAR, para gerar o pacote da aplicação, eu tenho que trocar `localhost`, eu tenho que colocar o IP do servidor, as propriedades do servidor do banco de dados de produção ou do ambiente de homologação. Eu tenho que trocar essas propriedades.

[02:23] Como é que eu faço para trocar essas propriedades? Eu vou ter que lembrar também de abrir o arquivo e modificar `localhost`, colocar o IP do servidor. Outra coisa para eu ter que lembrar, não tem como fazer isso de uma maneira automática - tem dois arquivos aqui, um para produção, um para cada ambiente. Fica mais um trabalho para fazermos. Percebe?

[02:43] Se eu tivesse outras atividades aqui também teria que renomear um arquivo, criar um diretório, jogar um arquivo para aquele diretório e apagar um determinado arquivo. Todo esse passo a passo faz parte do processo de *build* da aplicação.

[02:58] Antes de você gerar o pacote, o JAR ou WAR, você tem que lembrar de executar o passo a passo - e provavelmente é comum as pessoas esquecerem isso. É comum criarmos um documento, um passo a passo, um tutorial explicando como fazer o *build* do projeto. Isso é um negócio muito chato e trabalhoso, é sujeito a falhas. Seria melhor se automatizarmos isso, se tivesse uma ferramenta para automatizar isso.

[03:21] O pessoal do Java há muito tempo criou uma ferramenta bem famosa, que é o Apache Ant. Talvez você já tenha trabalhado com Apache Ant. Para você automatizar esse processo de compilar as classes do projeto, compilar as classes de teste, rodar os testes, criar um arquivo, mover um arquivo, apagar um diretório, apagar um arquivo e renomear coisas, você poderia utilizar o Apache Ant para fazer isso.

[03:46] Bastaria você vir no projeto, criar um arquivo "build.xml" e configurar o Ant para executar o *build* do seu projeto. Agora, você não faz todo esse passo a passo manualmente, você executa o Ant. Resolveria o nosso caso.

[04:00] Porém, tem outro detalhe nessa aplicação: e quanto às bibliotecas, os *framework* e as dependências dessa aplicação? Como é uma aplicação Java web, aqui no "Web-Content", dentro da pasta "WEB-INF" tem o diretório "lib" e todos os JARs da aplicação estão aqui.

[04:16] Tem aqui, essa aplicação está usando o `Spring framework`, está usando Hibernate, está usando biblioteca de log. Tem um monte de bibliotecas sendo utilizadas aqui.

[04:27] Sempre que eu precisar atualizar uma biblioteca dessa, atualizar o Hibernate por exemplo, vou ter que ir lá, baixar os JARs do Hibernate e substituir os JARs do Hibernate. É "hibernate-annotations.jar", "hibernate-commons-annotations.jar" e "hibernate3.jar" - mas será que são só esses três mesmo?

[04:43] E as dependências do Hibernate? O Hibernate depende desta biblioteca "cglib-nodep-2.1_3.jar", "commons-fileupload-1.2.1.jar" e "commons-logging.jar".

[04:48] Vai ter que saber de cabeça quais são os JARs que você vai ter que substituir. Se uma dessas dependências também é utilizada pelo *Spring*, agora eu vou ter que atualizar também o *Spring*?

[04:59] Baixar esses JARs, ficar gerenciando esses JARs manualmente é um negócio muito trabalhoso, é muito complicado e sujeito às falhas também. Para resolver esse problema, mais uma ferramenta que foi criada para o Java, o Apache Ivy. Talvez você já tenha ouvido falar, tenha utilizado, talvez não. Não é uma ferramenta tão popular assim.

[05:19] O Ivy era outra ferramenta que você poderia utilizar em conjunto até com o Ant para resolver esses problemas da sua aplicação. Você tem o Ant fazendo a parte do *build* e o Ivy cuidando especificamente das dependências.

[05:35] Você criaria aqui um arquivo chamado "Ivy.xml" e você só declara qual é a dependência. Quero o Hibernate na versão 3, quero o *Spring* na versão X. Você declara as dependências e o Ivy baixa essas dependências, resolve todas as dependências necessárias para essa biblioteca e configura tudo no seu projeto.

[05:55] Uma solução comum era utilizar essas duas ferramentas em conjunto, o Apache Ant e o Apache Ivy. Uma para cuidar da parte de *build*, de compilar, rodar testes, criar diretórios e mover arquivos; e a outra para gerenciar as dependências do projeto. Isso era bem comum. Porém são duas ferramentas separadas que se integram entre si, mas não seria melhor juntar essas duas ferramentas? Esse é justamente o objetivo do Maven.

[06:20] O Maven já cuida dessas duas etapas para você, ele já gerencia o *build* da aplicação, todo o processo, todo passo a passo para você gerar o *build* no pacote da sua aplicação - com todo esse passo a passo de compilar, de executar testes, de interromper o *build* se algum teste falhar, renomear arquivos e mover diretórios.

[06:49] Ele usa o Ant por baixo dos panos e ele também faz o trabalho lá que o Ivy fazia com a gestão das dependências das bibliotecas e *frameworks* que você quiser usar no seu projeto. O Maven veio justamente para substituir essas duas ferramentas. Ao invés de você usar duas ferramentas separadas, você poderia simplesmente utilizar o Maven.

[06:58] Tem outras ferramentas que são concorrentes, que são similares ao Maven - como o Gradle, o SBT, dentre outros que fazem algo parecido; mas o Maven é o mais popular, o mais famoso no mundo Java.

[07:12] Vai ser justamente o foco aqui do nosso treinamento. Vamos aprender a utilizar o Maven para resolver esses problemas. Problema de *build* de aplicação e de gerenciamento de dependências em Java. O Maven resolve isso e esse vai ser o foco aqui do curso. Vamos aprender como utilizar o Maven para resolvermos esses dois problemas clássicos em aplicações Java.

[07:33] No próximo vídeo, continuamos discutindo como funciona o Maven.