05

Módulos

Transcrição

[00:00] Nesse vídeo vamos aprender mais um recurso do Maven, que é a ideia de você ter módulos. Às vezes, você tem aplicações que são maiores, são complexas e você quer modularizá-las, você quer dividir em alguns módulos com projetos separados. É possível fazer isso com o Maven.

[00:21] Hoje, como já temos o Java 9, o Java 9 já tem o sistema de módulos, você poderia usar o esquema do Java 9 para uma modularizar aplicação. Antes disso, você não tinha esse recurso no Java e uma das maneiras de fazer isso era utilizando, por exemplo, o Maven. O Maven tem um suporte para módulos onde você consegue criar uma aplicação que vai funcionar como pai de outras aplicações e você tem esses módulos e essas dependências entre módulos.

[00:47] Eu vou mostrar esse recurso para vocês agora. Como aquela aplicação que eu estava usando era uma aplicação bem simples, só de exemplo para nós focarmos no Maven, eu tenho uma outra aplicação que eu vou importar no Eclipse e ela já está utilizando esse esquema de módulos. Vamos entender com calma.

[01:02]No *package explorer* Vou clicar em “Import projects”, vou filtrar aqui pelo “maven”, “Existing Maven Projects > vou escolher o diretório”, e essa minha pasta chamada “clean-architecture”. Clico em “OK”. Perceba que ele encontrou um “pom.xml”, só que dentro tem mais três “pom.xml” - justamente porque ele detectou que esse daqui é um módulo, um projeto dividido em módulos.

[01:28] Vou clicar em “Finish”. Ele importou, só que olhe só que legal, ele importou quatro aplicações separadas.

[01:36] Você tem aqui esse “rh-domain”, “rh-persistence”, “rh-web” e tem esse “Clean-architecture”. Esse primeiro projeto é apenas o projeto principal, é o projeto pai (*parent*), ele serve apenas para agrupar os módulos. Perceba que dentro dele tem justamente as pastas dos três módulos de um “pom.xml”.

[01:59] Vamos dar uma olhada nessei nesse “pom.xml”. O que significa isso? Como que eu configuro um projeto para ser um pai, um *parent*, para agrupar outros projetos? Você tem o “pom.xml” tradicional, tem aqui `<modelVersion>4.0.0</modelVersion>`, `<groupId>`, `<artifactId>` e `<version>`.

[02:15] A diferença é o `<packaging>pom</packaging>`, ao invés de ser JAR e WAR, tem essa questão de ser um `pom`. Quando você tem um projeto que vai ser um *parent*, o *packaging* dele tem que ser `pom`. É para isso que serve esse empacotamento.

[02:28] Aqui tem umas propriedades para configurar o Java 8, configurar o projeto como UTF-8. Tem as dependências e aqui uma *tag* nova, `<modules>` - e nessa *tag* listamos quais são os módulos pertencentes a esse `pom`, a esse projeto principal.

[02:47] Aqui tem `<module>rh-domain</module>`, `<module>rh-web</module>` e `<module>rh-persistencia</module>`. Estamos declarando exatamente esses três módulos aqui e aqui("clean-architecture") é só o projeto principal.

[02:59] Ele está dando vários erros, é por conta da versão do Java e versão do Maven utilizada nesses projetos. A princípio isso não vai influenciar em nada aqui, não vamos perder muito tempo com isso.

[03:13] Agora vamos analisar esses módulos. Se olharmos, vamos pegar esse módulo "rh-domain". Aplicação Maven `src/main/java`, `src/test/resources`, vamos para o “pom.xml” que é o que interessa. Aqui no “pom.xml”, perceba

que no `<modelVersion>` e `<artifactId>` não tem `groupId` porque tem essa outra *tag* nova chamada `<parent>` e é ela que eu configuro quem é o módulo pai desse módulo onde estão as configurações principais.

[03:44] Aqui eu passo o `groupId` e o `artifactId` do módulo pai. Ele mostrou aqui que o `groupId` é esse, `<groupId>br.com.caleum.fj91</groupId>`. O `artifactId` é `<artifactId>rh</artifactId>`, que é justamente o `groupId` e `artifactId` do meu "pom.xml", que estão declaradas no "pom.xml" do projeto *parent*. Aqui a versão especifica desse projeto, `<version>1.0</version>`.

[04:04] Como eu estou declarando que eu tenho um projeto *parent*, é como se fosse a herança do Java. No projeto *parent*, como temos umas dependências JUnit e Mockito, automaticamente esse projeto herda essas dependências. Se eu quiser usar o JUnit nesse projeto *domain*, eu não preciso declarar aqui a independência porque ele já está herdando.

[04:23] Aqui eu já sei quais são os problemas que estão acontecendo. Como eu acabei de importar esses projetos, eles não estão instalados no meu repositório local. A *tag* `<parent>` não encontrou o projeto `br.com.caleum.fj91` e como no último vídeo eu tinha configurado esse *proxy* só de exemplo, vou desabilitá-lo aqui no arquivo "settings.xml".

[04:42] Vou colocar `active>false</active`, porque ele deve está tentando baixar da internet, baixar desse *proxy* e não existe esse *proxy*. Vou ocultar esse *proxy*. Agora vou só rodar esse projeto - "clean-architeturie > Run As > Maven Install", aí ele vai compilar e vai instalar no meu repositório local. Deve resolver esses problemas aqui.

[05:05] Como é a primeira vez que eu estou executando o projeto aqui no computador, demora um pouco porque ele vai baixar as dependências e tudo mais. Já baixou e começou a rodar.

[05:16] Perceba, olhe só que interessante: eu mandei ele fazer o *build* do projeto `clean-architeture`, que é o projeto pai, mas como ele tem três

módulos, também faz o *build* dos três projetos, dos três módulos. Para cada módulo que ele tiver vai fazer o *build* de cada um dos módulos, vai baixar as dependências específicas de cada um dos módulos. Quando você tem um projeto modularizado, na hora de gerar o *build* é diferente o projeto.

[05:40] "Rodrigo, eu vou ter que gerar o *build* de cada um desses projetos separados, um por um?" Não! Você vai fazer tudo pelo *parent*. É no *parent* que você faz o *build*, porque o *parent* já contém todos esses módulos. Ele já vai fazer o *build* de tudo ao mesmo tempo.

[05:54] Até aquela vantagem da herança temos aqui. "Eu preciso adicionar uma dependência que é comum para todos os projetos". É só você adicionar no "pom.xml" do projeto *parent* - como é o caso do JUnit e do Mockito.

[06:05] Se eu quiser escrever testes nesses três projetos, vou precisar do JUnit. Ao invés de declará-los separadamente, repetindo aqui em cada um dos projetos, eu adiciono ele no "pom.xml" do projeto *parent* e pronto. Projetos filhos, os módulos já automaticamente vão herdar por dependência.

[06:23] Aqui no console do Maven, vemos que ele está baixando as bibliotecas porque cada um dos projetos tem suas dependências específicas. Por exemplo: esse projeto `rh-web`, se olharmos o pom dele, também depende do projeto *parent* e ele tem as dependências do *Spring boot*, dos outros módulos. Olhe que legal, um módulo pode depender do outro, eu posso adicionar como dependência.

[06:50] Esse módulo `rh-web` eu quero que dependa do módulo de persistência. É só eu declarar `groupId`, `artifactId`, qual é versão, como uma dependência e ele vai baixar do repositório local - no caso, essa dependência.

[07:04] Como esses projetos estão executando apenas na minha máquina, a dependência só existe dentro da pasta ".m2". Se eu quiser, eu posso enviar esses projetos para o Mvn Repository.

[07:15] É possível você criar um projeto seu e enviá-lo para o Mvn Repository para você baixar da internet e outras pessoas conseguirem baixar da internet normalmente. Não precisa ser apenas local e envio por e-mail, ou algo do gênero. Tudo bem?

[07:30] Aqui ele vai continuar baixando as dependências e vamos continuar aqui.

[07:35] Esse é um recurso extremamente útil quando trabalhamos com Maven e com aplicações modularizadas. Por exemplo: hoje em dia é bem comum o pessoal desenvolver aplicações seguindo naquela arquitetura de microsserviços. Aqui seria um exemplo de como você poderia organizar esses microsserviços.

[07:57] Como que eu faço para quebrar o meu projeto, separar a parte web, a parte da persistência, a parte de domínio, os subdomínios? Como é que eu faço para esses projetos conversarem entre si, se interligarem? Como é que eu vou gerar o *build* desses 10, 5, 20 projetos quebrados, separados?

[08:15] Um exemplo: seria utilizar o Maven com essa estrutura de módulos. Você poderia criar um projeto, criar um projeto do tipo `pom`. Na hora que criamos o projeto Maven daqui dar para colher qual que o *packaging*. Se é JAR, se é WAR ou se é "pom.xml". Você adiciona aqui os módulos, adiciona as dependências comuns para todos eles e vai criando cada um desses módulos.

[08:38] Os módulos são aplicações Maven tradicionais. Se você criar ali um *new Maven Project*, normal, JAR ou WAR - dependendo da tecnologia que você usar no projeto, desenvolve o projeto, leve o código-fonte e as classes de teste.

[08:54] O "pom.xml", você declara as dependências específicas de cada um desses projetos. Esse projeto tem essas dependências - Spring boot, Tomcat, JSTL, cada um tem suas dependências. Se um projeto depende de um dos módulos, você declara como uma dependência.

[09:14] O módulo nada mais é do que uma dependência. Pense em um módulo como sendo um projeto. Um módulo é um projeto e é considerado uma dependência, então eu posso ter um projeto que uma das suas dependências é um dos módulos. Não tem o menor problema.

[09:29] A única diferença especial mesmo é no projeto raiz. Além de ter os três módulos aqui, você precisa ter um projeto raiz, que é só o projeto que vai agrupar tudo. Aqui dentro dele só tem um atalho para cada um dos seus módulos e o "pom.xml" dele tem essa diferença de ter o `<packaging>pom</packaging>`. Fora isso, nada de mais. Dependências e módulos aqui. Fora isso nada demais, é um projeto como outro qualquer.

[09:58] Agora finalmente terminou! Três minutos para rodar, baixar e instalar tudo. Ele buildou e perceba que no *build* mostra o resumo.

[10:10] Ele fez o *build*, o `rh` (que é o projeto principal) e buildou o `domain`, o `persistencia` e o `web`.

[10:16] Para você gerar o *build* de um projeto modularizado você gera o *build* só do projeto *parent* e ele gera cada um dos filhos específicos. Certo?

[10:25] Vou dar um *clean* " Maven > Update Project > OK". O problema aqui é algo específico do Maven, era alguma configuração dessa que estava faltando fazer o *build* dos projetos e dar um *clean* geral aqui no projeto.

[10:45] Aqui, um exemplo de projeto utilizando módulos do Maven. Existem várias maneiras de desenvolver aplicações modularizadas, o Maven é uma delas e é bem interessante principalmente por conta das dependências que você consegue reaproveitar dependências e os módulos conseguem se comunicar - um módulo pode depender do outro módulo.

[11:07] Esse é um recurso bem bacana. Avalie se você trabalha em um projeto que usa arquitetura de microsserviços. Se não faz sentido separar os

microsserviços em módulos do Maven, talvez seja interessante para o seu projeto.

[11:21] Espero que vocês tenham gostado. Eu vejo vocês no próximo vídeo. Um abraço e até lá!