01 Projeto da aula anterior

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

02 Plugins

Transcrição

Errata

Para a aula funcionar é preciso adicionar a dependencia do *jacoco* como apresentado abaixo:

```
<dependencies>
    <dependency>
        <groupId>org.jacoco</groupId>
        <artifactId>jacoco-maven-plugin</artifactId>
        <version>0.8.8</version>
    </dependency>
</dependencies> COPIAR CÓDIGO
```

[00:00] Agora que já completamos os dois principais pilares, a parte de gerenciamento de dependências e de *build*; nessa aula, vamos conhecer alguns outros recursos adicionais do Maven. São alguns recursos que podem te ajudar em algumas situações. Um deles é justamente esse recurso de *plugin*.

[00:21] Eu tinha comentado brevemente quando precisamos configurar a versão do Java da nossa aplicação, que por padrão o

Maven estava considerando que era Java 5. Para você trabalhar com *plugins* na *tag* `<build>` em `pom.xml` tem essa *tag* chamada `<plugins>`.

[00:36] Um *plugin* nada mais é do que algo que vai modificar, manipular o *build* da aplicação. Durante o *build* do projeto você pode adicionar *plugins*, que farão alguma coisa. Existem diversos *plugins*, cada um com um objetivo.

[00:49] Tínhamos conhecido esse *plugin*, que é o `maven-compiler-plugin`. Éum *plugin* do próprio Maven, onde configuramos qual é a versão do Java, dos arquivos `.java` da aplicação e do `.class` na hora em que ele for fazer a compilação. Passamos que é a versão 11.

[01:08] Um *plugin* tem esse objetivo, de adicionar algum recurso para o *build* da aplicação. Se abrirmos o navegador e no Google pesquisarmos "maven plugins", existe aqui site do Maven neste link - que explica o que são os *plugins* e ele tem uma lista de *plugins* que são suportados por ele.

[01:28] Tem os *plugins* padrões do Maven para fazerem *clean*, *deploy*, *test*. Tem alguns *plugins* para empacotamento de acordo com algumas tecnologias - tipo `ear`, `ejb` e aplicações de JavaEE. *Plugins* para relatórios de `changeLog` de mudanças que foram feitas na aplicação, e documentação do site, `pmd` para testes e boas práticas de programação.

[01:54] *Plugins* para ferramentas do *Ant*. Tem vários *plugins*, alguns antigos estão aposentados, esses daqui em *"Retired"* não funcionam

mais. Tem alguns que estão fora da zona do Maven *("Outside The Maven Land")* e não foi Maven que desenvolveu, mas ele suporta e destaca.

[02:11] Alguns deles são interessantes. Por exemplo: esse do `jetty`, esse daqui é um *plugin* para você ter um servidor embutido na aplicação. Ao invés de ter um `tomcat` adicionado externo no Eclipse, você pode adicionar o `jetty` diretamente no Maven e executá-lo pelo Maven.

[02:32] No caso do `jetty`, como é que funciona? Se você adicionar esse *plugin* do `jetty` no *prompt* você poderia vir e executar `mvn jetty:run`. Esse "Jetty:" é o nome do *plugin*, o prefixo do *plugin* e o "run" seria o *goal*.

[02:47] Perceba que um dos objetivos dos *plugins* é eu posso adicionar novos *goals* ao Maven nas etapas de *build* do Maven. Não existe esse *goal* `run` no Maven, por isso que ele está com `jetty:`. Aí, seu eu tentar executar, vai falhar porque eu não adicionei esse *plugin* no `pom.xml` da aplicação.

[03:04] Essa é a ideia do *plugin*, ele serve para fazer alguma coisa no projeto - em especial na etapa de *build*. Tem vários *plugins* no site e um deles eu vou mostrar para vocês - é um relacionado com testes automatizados, com a parte de relatório e cobertura de testes.

[03:21] É o *plugin* chamado "Jacoco". É uma ferramenta que analisa o código-fonte da aplicação e te dá um relatório de cobertura de teste. A porcentagem de cada classe, de cada pacote está coberto por testes

automatizados. Para você encontrar classes que ainda não foram testadas e coisas do gênero.

[03:37] Para adicionar um novo *plugin* na *tag* `<build>` tem a *tag* `<plugins>`. Já tem um *plugin*. Uso a tecla "Enter" e vou colar. Eu já tinha copiado é um *plugin* bem extenso. Agora vem um detalhe meio chato, cada *plugin* funciona de um jeito, cada *plugin* tem uma configuração diferente.

```
                        <plugin>
                                <groupId>org.jacoco</groupId>
                                <artifactId>jacoco-maven-
plugin</artifactId>
                                <version>0.8.2</version>
                                <executions>
                                        <execution>
                                                <goals>
                                                        <goal>prepare-
agent</goal>
                                                </goals>
                                        </execution>
                                        <execution>
                                                <id>report</id>
                                                <phase>test</phase>
                                                <goals>

<goal>report</goal>
                                                    </goals>
                                    </execution>
```

```
                                </executions>
                            </plugin>+
        </plugins> COPIAR CÓDIGO
```

[03:54] O *plugin* no `maven-compiler-plugin` era só declarar o `artifactId` e essa *tag* `<configuration>` com a versão do Java, `<source>` e o `<target>`. Simples assim!

[04:04] Agora esse do Jacoco não. Como ele não é do Maven, eu preciso do `<groupId>org.jacoco</groupId>` do `<artifactId>jacoco-maven-plugin</artifactId>`. Qual é a versão, `<version>0.8.2</version>`. Se ele tiver várias versões. Nesse têm uma *tag* chamada `<executions>` - já que é para ele executar algum *goal* em alguma fase do *build* do projeto.

[04:21] Tem uma execução que é exigida por ele `<goal>prepare-agent</goal>`, que é para preparar e fazer uma análise do projeto. Tem uma outra `<execution>` - que é a principal, que é a de *report*, `<id>report</phase>`. O *goal* se chama *report* e ele será executado na fase de teste.

[04:37] Durante o *goal* de teste, o Jacoco vai executar logo sequência esse *goal* de *report*, `<goal>report</goal>` - que é o que vai gerar o relatório em si. Essa é a configuração do Jacoco. Você tem que consultar a documentação do *plugin* para entender como você configura ele e como faz para executá-lo. No caso do Jacoco é dessa maneira.

[04:56] Vou salvar e vamos rodar pelo *prompt* mesmo. Para rodar o Jacoco não se cria um *goal* a mais - ele até cria aquele *goal report*, mas se você rodar o próprio `mvn test` já vai funcionar porque o *report* no código está sendo configurado para rodar na fase de teste.

[05:13] Perceba que ele rodou os testes e na sequência, `jacoco-maven-plugin:0.8.2:report`. Ele executou o Jacoco, o *goal* de *report*. Ele gerou um arquivo dizendo `target/jacoco.exec`.

[05:26] Se entrarmos lá no *target* nas pastas do computador mesmo, tem um "jacoco.exec", mas é um arquivo `.exec`, não tem como executarmos esse arquivo. Se você reparar tem uma pasta "site", que é onde o Maven gera a documentação da aplicação. No caso não estamos usando nenhum *plugin* para isso, por isso ele não tinha gerado.

[05:41] O Jacoco gera um *site* para mostrar esse relatório. Na pasta "site" tem uma pasta "jacoco" e dentro tem um "index.html". Ele gera uma página HTML com um relatório, se abrirmos está o relatório do Jacoco.

[05:56] É uma página bem feia, não tem CSS e não tem um visual bacana, mas para nós isso não importa muito. O que nos importa é o conteúdo. Ele lista todos os pacotes da aplicação, no caso a nossa aplicação só tem 1 pacote, o nível de cobertura se estiver bom ficaria verde. Está ruim, está vermelho, porque está com 0% de cobertura.

[06:15] Podemos detalhar. Quando você clica no pacote, ele te mostra todas as classes e em cada classe ele mostra a cobertura. Nós só temos

uma classe e ele mostra aqui métodos, construtores e o percentual de cobertura. Está tudo zerado, não tem nada sendo testado.

[06:30] Vamos tentar escrever um teste só para mudarmos isso daí. Nós até temos o `ProdutoTest`, só que tem um teste de vazio que não faz nada, `@Test public void test () {}`.

[06:37] Vamos instanciar um produto: `Produto p = new Produto()`.Lá na classe `produto`, se dermos uma olhada em `produto.java` tem um atributo de nome, um atributo de preço, um construtor que recebe os dois atributos e um *getter* para cada um dos dois atributos.

[06:51] Vou criar um produto `Produto p = new Produto("teste", BigDecimal.TEN)` e aí vou fazer um *assert*: `Assert.assertEquals("teste", p.getNome())`. É um teste não recomendado. Testar o método 'get' é algo bizarro. É só para vermos o relatório.

[07:24] Eu vou fazer um outro teste para ver se ele pega o preço aqui como sendo `(BigDecimal.TEN)`. Está dando um erro aqui. Foi o `Import`, não é desse pacote, é `assert` do "org Junit".

Código em `ProdutoTest.Java`:

```
package br.com.alura.loja;


import java.math.BigDecimal;


import org.junit.Test;
```

```
import org.junit.Test;


public class ProdutoTest {


        @Test

        public void test() {

             Produto p =  new Produto("teste", BigDecimal.TEN);

                        Assert.assertEqual("teste",p.getNome());


Assert.assertEqual(BigDecimal.TEN,p.getPreco());

        }
} COPIAR CÓDIGO
```

[07:38] É um teste bem meia boca, só para simular. Vamos rodar novamente no *prompt*, `mvn test`. Ele vai recompilar tudo, retestar o relatório geral. Se usar a tecla “F5” na tabela, olhe só que legal, agora já muda a figura!

[07:55] Vou mudar para “Home”. 100% de cobertura de teste. A barra verde dizendo que está boa a sua cobertura. Detalhou a classe produto. Tudo coberto, o construtor, os *getters*. É um *plugin* para cobertura de testes.

[08:11] Essa é que é a ideia dos *plugins*. Existem vários *plugins* na comunidade para relatório de testes - para modificar o *build* do projeto, para executar um servidor, para gerar um PDF e para rodar SQL. Tem *plugins* para tudo que você possa imaginar. Inclusive, você pode criar um *plugin*. Você pode criar um *plugin* que vai fazer alguma

coisa durante o *build* de uma aplicação. É totalmente possível fazer isso!

[08:37] Essa é uma das vantagens do Maven. Ele é extensível, não é engessado, só tem aquelas funcionalidades "x", "y", "z" e acabou; você só pode usar aquilo. Ele te permite estendê-lo, adicionar *plugins* para ter novos comportamentos e novas funcionalidades. Isso é algo extremamente importante em uma aplicação, essa capacidade de extensão para você adicionar novos comportamentos.

[08:59] Espero que tenham gostado de entender melhor como funciona esses *plugins* e essa que é ideia de um *plugin* que está associado com o *build* do projeto e que ele vai estender o Maven, adicionando novas características.

[09:12] Vejo vocês na próxima aula onde vamos conhecer outros recursos do Maven. Até lá!

03 Plugins do Maven

Vimos no último vídeo como adicionar plugins ao projeto.

Qual o objetivo de se utilizar plugins?

- Alternativa correta

  Acelerar o processo de build

- Alternativa correta

  Adicionar novas funcionalidades ao Maven

  Alternativa correta! Plugins servem para **extender** os comportamentos do Maven.

- Alternativa correta

Baixar dependências de outros repositórios remotos

04 Proxy

Transcrição

[00:00] Outro recurso bem bacana do Maven é a configuração de *proxy*. Às vezes, isso vai ser extremamente útil para você.

[00:10] Se você usar o Maven no seu trabalho ou em alguma rede que tem *proxy* para você configurar o acesso à internet, se você for usar o Maven nessa situação, você vai ter algum problema porque na hora dele baixar as dependências da internet não vai conseguir, porque precisa de um *proxy*. Sem um *proxy* você não vai conseguir navegar. O Maven iria gerar um erro para você.

[00:31] Como eu faço para configurar um *proxy* no Maven? A princípio, você pensaria: "basta abrir o `pom.xml`, deve ter alguma propriedade em algum lugar chamada *proxy* ou algo do gênero! " Só que na verdade não tem. Como isso não é algo do projeto. É um *proxy*, é uma configuração do Maven para o Maven acessar a internet não é algo específico de um projeto. Não fica no `pom.xml`.

[00:57] Existem algumas configurações do Maven e essas configurações que são externas ficam em um arquivo específico. Lembra daquela pasta ".m2"? Eu estou nela, dentro dela tem aquele diretório "repository", que tem os JAR, as dependências e dentro se você quiser configurar alguma coisa, tem que criar um arquivo chamado "settings.xml".

[01:22] Eu já criei esse arquivo e está vazio. Temos a *tag* `</settings>`, a *tag* raiz do arquivo, tem os *namespaces* do Maven. Você pega um desse exemplo na internet - não vai ficar memorizando isso - e aqui dentro vem configurações globais, configurações específicas do Maven e não necessariamente dizem respeito a um único projeto.

[01:45] Dentre essas configurações vem a configuração de *proxy*. Eu já tenho ela copiada vou só colar para não perdermos muito tempo. É dessa maneira que configuramos um *proxy*.

Código em `setting.xml`:

```
<settings xmlns="http://maven.apache.org/SETTINGS¹1.0.0"


//código omitido


    <proxies>
        <proxy>
                    <id>genproxy</id>
                    <active>true</active>
                    <protocol>http</protocol>
                    <host>proxyHost</host>
                    <port>3128</port>
                    <username>username</username>
                    <password>password</password>
        </proxy>
    </proxies>


    </settings> COPIAR CÓDIGO
```

[01:56] Dentro da *tag* raiz `<settings>` você coloca a *tag* `<proxies>` e dentro dela você coloca os *proxys* que você tiver, geralmente é só um. Como podem ter vários, você coloca um `<id>meu-proxy</id>`, `active>true</active>` para dizer que esse *proxy* está ativo. Qual é o protocolo? `protocol>http</protocol>`.

[02:13] Aí você vai ter que conhecer o *proxy* que você utiliza. É HTTP? É HTTPS? O *host*. Qual é o endereço do *proxy*? Provavelmente vai ser algo assim `<host>http://ip</host>`, esse é o IP interno da essa sua empresa que funciona como um *host*.

[02:33] Qual é a porta? `<port>3128</port>`. Vai ter que conhecer. E se tiver *login* e senha, qual é? `<username>username</username>`. E qual é a senha? `<password>password</password>`. Se não tiver *login* e senha, o *proxy* não exigir você pode ocultar essa *tag username* e *password*. No geral, quando tem *proxy* tem usuário e senha, deve ser teu *login* da rede, a sua senha da rede.

[02:49] Dessa maneira você configura o *proxy*. Vou salvar e pronto, basta salvar isso daqui nesse arquivo `settings.xml` dentro do diretório ".m2"! A partir de agora, o Maven, quando for baixar as dependências e quando precisar acessar a internet, vai levar em consideração esses *proxys*.

[03:09] Outra coisa que daria para fazer: lembra no projeto que configuramos um repositório em `pom.xml`, além de usarmos o repositório central do Maven? Dá para fazermos isso no `pom.xml` para ser um repositório específico desse projeto, mas dá para configurar

repositório também no `settings.xml`, que ele fica valendo para todos os projetos.

[03:28] Só que nesse caso, como não dá para configurar no `settings.xml` e no `pom.xml`, se tiver um no `pom.xml` ele tem prioridade, ele vai meio que sobrescrever o que está configurado no `settings.xml`. O que é específico do projeto tem precedência em relação ao `settings.xml` - que é mais global, digamos assim.

[03:50] Mais um recurso, às vezes a galera quebra a cabeça por conta disso: você vai usar o Maven em casa. Funciona e quando usar no trabalho tem alguma dor de cabeça porque tem um *proxy* e para configurar não é no `pom.xml`, é nesse arquivo de configurações.

[04:04] Esse que era o objetivo do vídeo de hoje, mostrar para vocês como configurar um *proxy* no Maven. É bem simples e bem tranquilo, nada demais. No próximo vídeo continuaremos vendo outros recursos do Maven.

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, quer dividi-las em alguns módulos com projetos separados. É possível fazer isso com o Maven.

[00:21] Hoje, como já temos o Java 9 que já tem o sistema de módulos, você poderia usar o esquema do Java 9 para uma modularização de 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 pelo, digitando “maven” em *"Select an import wizard"* e selecionar “Existing Maven Projects", clicar no botão "Next", vou escolher o diretório em "Browser" que vai ser minha pasta chamada “clean-architecture”. Clico em “OK”. Perceba que ele encontrou um `pom.xml`, só que dentro tem mais três `pom.xml`, "rh-domain/pom.xml", "rh-web/pom.xml" e "rh-persistencia/pom.xml" - justamente porque ele detectou que esse 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 do lado esquerdo da tela.

[01:36] Você tem em *"Package Explorer"* esse “rh-domain”, “rh-persistencia”, “rh-web” e tem esse “Clean-architecture”. Esse primeiro projeto “Clean-architecture” 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 e um `pom.xml`.

[01:59] Vamos dar uma olhada 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, temos `<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] Temos umas propriedades `<properties>` para configurar o Java 8, configurar o projeto como UTF-8. Tem as dependências `<dependencies>` e uma *tag* nova, `<modules>` - e nessa *tag* listamos quais são os módulos pertencentes a esse `pom`, a esse projeto principal.

[02:47] Em `<modules>` temos `<module>rh-domain</module>`, `<module>rh-web</module>` e `<module>rh-persistencia</module>`. Estamos declarando exatamente esses três módulos que apareceram em *"Package explorer"* e *"clean-architecture"* é só o projeto principal.

Código *clean-architecture-pom.xml*:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>br.com.caelum.fj91</groupId>
    <artifactId>rh</artifactId>
    <version>1.0</version>
    <packaging>pom</packaging>

    <properties>
        <project.build.sourceEncoding>UTF-
8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-
8</project.reporting.outputEncoding>
        <java.version>1.8</java.version>
        <maven.compiler.target>1.8</maven.compiler.target>
        <maven.compiler.source>1.8</maven.compiler.source>
    </properties>

    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
            <scope>test</scope>
        </dependency>

        <dependency>
            <groupId>org.mockito</groupId>
            <artifactId>mockito-all</artifactId>
```

```
            <version>1.10.19</version>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <modules>
        <module>rh-domain</module>
        <module>rh-web</module>
        <module>rh-persistencia</module>
    </modules>

</project> COPIAR CÓDIGO
```

[02:59] Ele está retornando vários erros, é por conta da versão do Java e versão do Maven utilizada nesses projetos. A princípio isso não vai influenciar em nada, 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. No `rh-domain/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] Passo o `groupId` e o `artifactId` do módulo pai. Ele mostrou 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*. A versão específica desse projeto, `<version>1.0</version>`.

Código em `rh-domain/pom.xml`:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <artifactId>rh-domain</artifactId>

    <parent>
        <groupId>br.com.caelum.fj91</groupId>
        <artifactId>rh</artifactId>
        <version>1.0</version>
    </parent>

</project> COPIAR CÓDIGO
```

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

[04:23] 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 tinha configurado esse *proxy* só de exemplo, vou desabilitá-lo 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.

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

[05:16] Perceba 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. "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 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] 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/pom.xml`, se olharmos o *pom* dele, percebemos que também depende do projeto *parent* e ele tem as dependências do *Spring boot*, dos outros módulos. Um módulo pode depender do outro, eu posso adicionar como dependência.

[06:50] Esse módulo `rh-web` 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, 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.

[07:30] Ele vai continuar baixando as dependências e vamos continuar.

[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 a 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 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 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] No `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. Dentro de `clean-architeture` 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. 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.

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

[10:45] Apresentei para vocês 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!

06 Modularização de projetos

Vimos no último vídeo como utilizar módulos no Maven.

Como o Maven identifica os módulos de um projeto?

- Alternativa correta
  O Maven detecta os módulos do projeto automaticamente
- Alternativa correta
  Pela tag `<sub-projects>` adicionada no `pom.xml`
- Alternativa correta
  Pela tag `<modules>` adicionada no `pom.xml`
  Alternativa correta! Essa é a tag que deve ser utilizada para declarar os módulos de um projeto.

07 Projeto separado em módulos

O projeto feito com o recurso de módulos do Maven, que foi mostrado no último vídeo, pode ser encontrado aqui: https://github.com/rcaneppele/fj91-clean-architecture

08 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 os próximos cursos que tenham este como pré-requisito.

Opinião do instrutor

-
-

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

09 O que aprendemos?

Nesta aula, aprendemos:

- A utilizar plugins do Maven;
- A como configurar um *proxy* no Maven;
- A como configurar uma aplicação modularizada no Maven.

10 Conclusão

Transcrição

[00:00] Chegamos ao final do treinamento sobre **Maven**, sobre como funciona o Maven. Durante esse treinamento, aprendemos o que é o Maven, quais são os dois pilares principais do Maven - que é a gestão de dependências e *build* do projeto. Fomos aprendendo esses recursos e algumas aplicações de exemplo.

[00:21] Aprendemos como se faz para instalar o Maven. Você pode baixar o ZIP, descompactar e executar pelo *prompt* de comandos ou através da sua IDE. No caso do Eclipse, ele já tem uma integração nativa com o Maven. Aprendemos como se faz para criar uma aplicação Maven usando aquela opção do Eclipse, "New > Maven Project".

[00:39] Vimos a questão do *archetype*, que podemos pular ou escolher um *archetype* específico que já vem com uma estrutura específica. Entendemos essa estrutura de diretórios de uma aplicação Maven, o que é cada um desses diretórios - "src/main/java", "src/main/resources" , "src/test/java" e "src/test/resources".

[00:57] Aprendemos a trabalhar no `pom.xml`, aquele arquivo XML de configurações do Maven para esse projeto. Vimos a declaração do `groupId`, `artifactId`, as dependências. Uma das principais utilizações do Maven, que é como declaramos dependências, como pesquisamos por essas dependências.

[01:15] Aprendemos sobre o repositório central do Maven, como adicionar outros repositórios no Maven. Aprendemos sobre o segundo pilar do Maven - que é essa parte de *build*, como é que eu faço para fazer o *build* da minha aplicação, para compilar, para rodar testes, gerar o JAR, o WAR e fazer o *deploy*.

[01:31] Aprendemos como customizar esse *build*, trocando o nome do arquivo, do artefato gerado. Aprendemos a como adicionar*plugins*, como o *plugin* do compilador do Maven para trocar versão do Java, o *plugin* do Jacoco para ver o relatório de cobertura de testes.

[01:45] Aprendemos a mexer naquele arquivo `settings.xml` para declararmos um *proxy*, por exemplo - dentre outras coisas que podem ser declaradas.

[01:54] Por fim, aprendemos sobre essa questão de modularização. Como ter uma aplicação modularizada, uma aplicação cujo o *packaging* é "pom" e ela declara dependências que são compartilhadas entre os seus módulos e faz essa declaração de quais módulos pertencem a essa aplicação.

[02:10] Você tem o projeto *parent* - o projeto raiz que agrupa esses subprojetos e cada projeto filho declara essa *tag* `<parent>`, então “pom.xml” fica alterado nessa situação. Você está herdando do “pom.xml”, daquele projeto e você herda todas as dependências e configurações que estão descritas. Com isso, você pode quebrar a sua aplicação e ter um reaproveitamento, evitar código duplicado de dependências e configurações.

[02:38] Um módulo pode depender do outro. Por exemplo: em `rh-web/pom.xml` temos esse módulo `rh-web`, que depende do módulo `rh-persistencia`. Basta declarar como sendo uma dependência. O módulo nada mais é do que uma dependência, é um projeto que pode ser adicionado como dependência a outro projeto.

[02:56] Aprendemos sobre essa questão de modularização e que pode ser usada para criar aplicações que seguem aquela arquitetura de microsserviços.

[03:04] Esses foram os recursos que aprendemos durante esse treinamento sobre o Maven para te dar esse *overview*, essa noção do que é o Maven, para que ele serve, como funciona e como eu crio uma aplicação.

[03:15] O Maven, como tinha dito, é a principal ferramenta para gerenciamento de dependências e *build* de aplicações em Java. Muito provavelmente você vai trabalhar em aplicações que utilizam o Maven. É extremamente importante e recomendado que você utilize alguma ferramenta como o Maven.

[03:29] Tem o *Gradle* também, que é uma outra ferramenta bem popular e concorrente forte do Maven. O Maven acaba sendo o mais utilizado no mercado, o "mais padrão", digamos assim, no caso do Java.

[03:41] Espero que vocês tenham gostado desse treinamento e que tenham trabalhado com o Maven. Vejo vocês em outros treinamentos da Alura. Um abraço e até lá!