03

# Um gostinho do Maven

### Transcrição

Vamos buscar o Maven para que possamos realizar seu download. Acessaremos a página oficial (https://maven.apache.org/) do software e escolheremos a opção "Download" no menu principal.

Baixaremos a última versão disponível, de acordo com o sistema operacional em uso. No nosso caso, salvaremos o arquivo `apache-maven-3.3.9` no diretório `workspace`, mas você pode inseri-lo no diretório que faz mais sentido para o seu projeto.

Descompactaremos o aquivo baixado, e um dos diretórios existentes é o `bin`, no qual clicaremos e teremos acesso ao arquivo `mvn`. Caso você clique sobre ele, nada acontecerá, porque utilizamos o Maven no terminal. Assim, pelo terminal, acessaremos o diretório `apache-maven-3.3.9`, e dentro dele o `bin`. Utilizamos `pwd` para confirmar o diretório.

```
cd Documents/guilherme/workspace/
ls
cd apache-maven-3.3.9
cd bin
pwd
```

Em `bin` temos o `mvn`, e para executarmos este arquivo, iremos acionar `./mvn`. Dessa forma rodaremos um script desse diretório. O script é executado, mas recebemos uma mensagem de erro, pois não especificamos nenhum comando. Declaremos `mvn` e, dessa vez, passaremos o seguinte comando, com que as informações de ajuda são exibidas.

```
./mvn --help
```

O Maven é executado corretamente, entretanto não é prático entrarmos neste diretório toda vez que precisarmos acioná-lo. Para solucionarmos esse problema, adicionaremos o diretório atual em nosso *path*, que no Windows e no Linux são as variáveis de ambiente. Neste caso, voltaremos ao diretório original ( `cd` ), que conferiremos por meio do comando `pwd`.

```
pwd
/Users/alura/Documents/guilherme/workspace/apache-maven-3.3.9/bin
cd
pwd
/Users/alura
```

No diretório original queremos editar um arquivo para configurar que o diretório `/Users/alura/Documents/guilherme/workspace/apache-maven-3.3.9/bin` irá para o *path* global de execução de arquivos. No Linux, fazemos isso por meio da variável de ambiente que podemos configurar diretamente neste diretório, utilizando o comando `vi .bash_profile`.

Estamos prontos para editar o arquivo, portanto pressionaremos a tecla "A" para realizar a inserção de conteúdo. Escreveremos o comando `export` para exportar a configuração de `PATH` como sendo o atual ( `$PATH` ), e ( `:` ) o diretório

`:/Users/alura/Documents/guilherme/workspace/apache-maven-3.3.9/bin` . Estamos afirmando que o novo *path* é composto pelo atual, **e** pelo novo diretório.

```
export PATH=$PATH:/Users/alura/Documents/guilherme/workspace/apache-maven-3.3.9/bin
```

Para salvarmos as modificações, usaremos `:wq` , porque utilizamos o `vi` . Caso você tenha utilizado outro editor, salve o arquivo `bash_profile` no diretório do seu usuário.

Ainda não podemos acionar o `mvn` , pois ao configurarmos o *path* devemos reabrir o terminal. Em uma nova aba digitaremos `mvn --help` , e as informações de ajuda ficarão visíveis no terminal, e isso quer dizer que o Maven foi encontrado no diretório do *path*.

Temos o Maven descompactado e instalado, e estamos prontos para criar um projeto. Com `cd Documents/guilherme/workspace/` solicitaremos que o Maven gere um novo projeto, mas antes precisamos estipular sua estrutura básica ou arquétipo, portanto escreveremos:

```
mvn archetype:generate
```

Precisamos passar o nome ou ID do nosso projeto, que será `produtos` . Digitaremos `-DartifactId=produtos` para instaurar essa informação.

```
mvn archetype:generate -DartifactId=produtos
```

O próximo passo consiste em declararmos o pacote básico da empresa, já que `produtos` se refere ao projeto em si, isto é, apenas uma parte do pacote. Usaremos `-DgroupId=br.com.alura.maven` , pois estamos elaborando um curso sobre Maven na **Alura**.

Desabilitaremos o método interativo usando as informações padrão para configurarmos nosso projeto. Para isso, incluiremos `-DinteractiveMode=false` :

```
mvn archetype:generate -DartifactId=produtos -DgroupId=br.com.alura.maven -DinteractiveMode=fals
```

Até agora temos as seguintes solicitações: a criação do projeto ( `archetype:generate` ), com o nome `produtos` , pacote base `br.com.alura.maven` , e configurações automáticas. Ainda resta uma última solicitação: queremos que o projeto siga um exemplo simples, disponibilizado pelo Maven, e que pode ser utilizado como base.

Para acionarmos o projeto modelo do Maven, usaremos `-DarechetypeArtifactId` . Se buscarmos esse termo no Google, por exemplo, encontraremos diferentes tipos de projeto modelo.

Caso precisemos de um *start* rápido com Spring MVC, basta procurarmos qual `ArchetypeArtifactId` é usado para criar um projeto com estas características usando Maven. Você pode variar as ferramentas de acordo com seu interesse. No nosso caso, queremos um modelo simples e rápido, portanto buscaremos o **ArchetypeArtifactId quickstart** no Google. No terminal, incluiremos o resultado da nossa pesquisa, `-DarchetypeArtifactId=maven-archetype-quickstart` , desse modo teremos um modelo de projeto a ser seguido.

```
mvn archetype:generate -DartifactId=produtos -DgroupId=br.com.alura.maven -DinteractiveMode=fals
```

Após pressionarmos o botão "Enter", será feito o download de várias informações. Existe um ditado de que o "Maven baixa a internet inteira", pela quantidade de conteúdos que ele insere no projeto. Qualquer programa de *build* "baixa a internet inteira", e entenderemos melhor isso analisando o Maven.

O Maven possui um *core* muito pequeno, por isso seu download é tão rápido. **Todas as ferramentas utilizadas por este software são plugins**, portanto na primeira vez em que ele é executado são feitas as captações dos elementos que de fato serão necessários para a construção do projeto.

Neste instante, estamos inserindo uma série de bibliotecas que irão possibilitar a construção do nosso projeto. Todas as vezes em que precisarmos de algo indisponível no Maven, ele irá fazer o download do plugin necessário automaticamente.

Terminado o processo de baixar todos os conteúdos, será criado o modelo de projeto. No diretório `workspace`, passaremos a ter `produtos`. Dentro deste, encontraremos `src`, que por sua vez abriga mais duas pastas, `main` e `test`, o código principal e o de teste, respectivamente. Abriremos o projeto `produtos` no editor Sublime, para que possamos visualizar sua estrutura com mais clareza. Abriremos o seguinte arquivo no editor: "produtos > src > main > java > br > com > alura > maven > App.java".

O Maven cria automaticamente uma classe `App` no diretório `br.com.alura.maven`, dentro da qual há o método `main`, apenas como exemplo.

```
package br.com.alura.maven;

/**
*Hello world!
*
*/
public class App
{
    public static void main( String[] args )
    {
        System.out.println( "Hello World!");
    }
}
```

No diretório `test`, temos "test > java > br > com > alura > maven > AppTest". Na classe `AppTest` há um construtor e alguns métodos simples, e assim temos um pequeno exemplo de teste.

```
package br.com.alura.maven;

import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestCase;
import junit.framework.TestSuite;

/**
*Unit test for simple Apps
*/
public class AppTest
    extends TestCase
{
```

```
    /**
    *Create the test case
    *
    * @param testName name of the test case
    * /
    public AppTest( String testName )
    {
        super( testName );
    }

    /**
    * @return the suit of tests being tested
    */
    public static Test suite()
    {
        return new TestSuite(AppTest.class);
    }

    /**
    * Rugourous Test :-)
    */
    public void testApp()
    {
        assertTrue( true );
    }
}
```

O arquétipo que utilizamos cria uma estrutura de código fonte de Java principal e de teste. Temos, ainda, um arquivo em XML, o `pom.xml` que estudaremos mais adiante. Por ora, devemos lembrar que criar um projeto não costuma ser a parte mais difícil, porém o Maven nos auxilia também nessa parte.

Além disso, ele **nos ajuda a compilar todo o processo de *build***, e não precisamos fazer toda a compilação manualmente, basta solicitarmos que o software realize todo esse processo para nós por meio do comando `mvn compile` .

Na primeira vez em que o Maven compila um projeto, ele irá baixar o plugin de compilação e outras bibliotecas necessárias. No caso do nosso projeto, aparentemente não precisamos de nenhuma biblioteca por enquanto. Ao final da instalação dos conteúdos, teremos a seguinte mensagem:

```
[INFO] Compiling 1 source file to /Users/alura/Documents/guilherme/workspace/produtos/target/cla
```

O Maven compila um arquivo para o diretório `target/classes` que contém o arquivo `App.class` . Se precisarmos refazer a compilação, o Maven não irá realizar todas as instalações novamente. De forma geral, as ferramentas de *build* precisam baixar as dependências em algum momento, e conforme o projeto vai se tornando cada vez mais complexo, novas necessidades são criadas.

Para compilarmos a classe de teste, basta usarmos `mvn test` no terminal. Como estamos compilando uma classe de teste pela primeira vez, os plugins relacionados e possíveis bibliotecas necessárias serão baixadas. Por exemplo, é comum usarmos o **JUnity**, portanto o download desse *framework* é realizado. Temos também o download do *Surefire*, que gera o relatório de teste, por exemplo.

Caso executemos o teste novamente, nada novo será baixado, e o processo será bem mais rápido. No diretório `target` , haverá uma nova pasta chamada `surefire-reports` , que abriga arquivos TXT em que o resultado do teste em cada

classe é exibido. Como só temos a classe `AppTest`, teremos um único relatório.

À medida em que formos criando novas classes, novos relatórios serão gerados. Na prática, criaremos um arquivo no diretório `maven`, que conterá uma classe que chamaremos de `Produto`. Essa nova classe está inserida do pacote `br.com.alura.maven`.

```
package br.com.alura.maven;

public class Produto {

}
```

Uma vez que a classe `Produto` tiver sido criada, voltaremos ao terminal e executaremos a compilação por meio de `mvn compile`, que já conhecemos bem. Ao final, teremos dois arquivos fonte no diretório `classes` — `App.class` e `Produto.class`.

Existe algo que não analisamos com profundidade: a classe `AppTest` possui algumas importações do JUnit, uma biblioteca, isto é, um arquivo `.jar`.

```
package br.com.alura.maven;

import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestCase;
import junit.framework.TestSuite;
```

Para que esse arquivo seja executado, o `.jar` deve estar no *classpath*. Como o Maven decidiu que precisávamos do `.jar`? Todas as informações estão no arquivo `pom.xml`, o modelo do nosso projeto.

Ao acessá-lo, perceberemos que ele configura sua própria versão ( `4.0.0` ), e essa informação não será alterada. Veremos que nosso projeto está no grupo `br.com.alura.maven`, seu nome é `produtos`, e ele gera um `.jar`. Temos ainda a versão atual `1.0-SNAPSHOT` em `<url>`, e podemos colocar o site do nosso projeto, como `http://www.alura.com.br`, com o qual temos as dependências.

```
<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/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>br.com.alura.maven</groupId>
    <artifactId>produtos</artifactId>
    <packaging>jar</packaging>
    <version>1.0-SNAPSHOT</version>
    <name>produtos</name>
    <url>http://www.alura.com.br</url>
    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>jnuit</artifactId>
            <version>3.8.1</version>
            <scope>test</scope>
```

```
        </dependency>
    <dependencies>
</project>
```

Reparem que há uma dependência para o JUnit, cuja versão inserida por padrão é bem antiga. Na prática, utilizamos versões bem mais novas, e podemos modificar essa informação. Mais abaixo, temos o `<scope>`, e caso o arquivo seja apenas para teste, é inserida a palavra `test`, caso seja também para execução, não precisamos inserir nada, afinal a execução é o padrão.

Esta é apenas uma amostra do que é trabalhar com o Maven na linha de comando. Ao invés de realizarmos tudo manualmente, podemos utilizar comandos como `mvn test` para testar o seu projeto, ou `mvn clean` para limpá-lo. Além disso, existem plugins para uma infinidade de ações, e nós estudaremos mais as possibilidades do Maven ao longo do curso.