

Gestão de Configuração e Mudanças de Software

Allan Lima

Arquitetura, Design e Implementação de Sistemas para Internet
Pós Graduação
Faculdade 7 de Setembro

arglbr@gmail.com

nov/2015

Introdução ao Maven

Parte 1

O que é Maven?

- Uma tentativa de definir Maven:
 - Ferramenta para simplificar o processo de build.
- Build?
 - Tarefas rotineiras que levam a construção ou montagem de um software a partir do código fonte.
- Tarefas rotineiras?
 - Compilar, executar testes, empacotar aplicação, etc.
- Outra tentativa de definir Maven:
 - Um conjunto de padrões usados para gerenciar e descrever projetos em java.

Quais os benefícios do Maven?

- Modelo que pode ser aplicado aos projetos Java.
- A idéia é que o modelo traga mais transparência, mais reuso, mais facilidade de manutenção e entendimento.
- Fornece uma abstração que a maioria dos desenvolvedores estão familiarizados:
 - Semelhante a abstração do automóvel: se você aprendeu a dirigir em um modelo A de carro, então poderá facilmente dirigir um modelo B.

Quais os benefícios do Maven?

- Abordagem declarativa:
 - POM — Project Object Model.
 - As tarefas são delegadas para o POM e para os plugins.
 - Os desenvolvedores podem usar as tarefas (encapsuladas pelos plugins) sem necessariamente entender como elas funcionam internamente

- CoC - Convenções sobre a configuração
- Reuso de lógicas de *builds*
- Execução declarativa
- Organização coerente de dependências

Princípio 1: CoC - Convenção sobre a configuração

- Estratégia de “propriedades *defaults*” para a maioria das tarefas (podem ser alteradas quando conveniente) economiza tempo.
- Convenções primárias:
 - Estrutura de diretórios padrão para projetos
 - Código fonte, recursos (xml, properties), saída de arquivos gerados, documentação etc
 - Cada projeto gera um único resultado: jar, war, ear...
 - Padrões de nomes
 - Para diretórios: my-app/src/main/java
 - Para arquivos gerados (*outputs*): commons-logging-1.2.jar

Princípio 2: Reuso de lógicas de *builds*

- Toda a lógica de *build* é encapsulada pelos *plugins*
- Um *plugin* para
 - Para compilar o código fonte
 - Para executar os testes de unidade
 - Para empacotar a aplicação (jar, war, ear)
 - Para gerar javadocs
 - etc

Princípio 3: Execução declarativa

- Tudo no *maven* é orientado de forma declarativa no POM e nas configurações específicas dos *plugins*. Exemplo do POM:

```
1 <project>
2   <modelVersion>4.0.0</modelVersion>
3   <groupId>br.com.curso</groupId>
4   <artifactId>dv-generator</artifactId>
5   <version>0.0.1-SNAPSHOT</version>
6   <name>DV Gerador</name>
7   <description>Gerador de digito verificador</description>
8   <dependencies>
9     <dependency>
10       <groupId>junit</groupId>
11       <artifactId>junit</artifactId>
12       <version>4.4</version>
13       <scope>test</scope>
14     </dependency>
15   </dependencies>
16 </project>
```

Princípio 3: Execução declarativa

- Ciclo de Vida de *build*:
 - Consiste de uma série de fases onde cada uma pode executar uma ou mais ações (goals) relacionada a essa fase
 - Uma fase pode incluir outras fases:

A fase `compile` executará:

- `validate`
- `initialize`
- `generate-sources`
- `process-sources`
- `generate-resources`
- `compile`

Princípio 4: Organização de dependências

- Conceitos conectados:

- Dependências
- Artefatos
- Repositórios

```
1 <project>
2   <modelVersion>4.0.0</modelVersion>
3   <groupId>br.com.curso</groupId>
4   <artifactId>dv-generator</artifactId>
5   <version>0.0.1-SNAPSHOT</version>
6   <name>DV Gerador</name>
7   <description>Gerador de digito verificador</description>
8   <dependencies>
9     <dependency>
10       <groupId>junit</groupId>
11       <artifactId>junit</artifactId>
12       <version>4.4</version>
13       <scope>test</scope>
14     </dependency>
15   </dependencies>
16 </project>
```

Princípio 4: Organização de dependências

- Uma dependência é uma **referência** a um artefato específico em um repositório
- Para o maven disponibilizar a dependência ele precisa saber em qual repositório procurar usando as **coordenadas** da dependência: `groupId`, `artifactId` e `version`
- Maven busca a dependência de um **repositório remoto** e copia para um **repositório local**
- Sempre que uma dependência não se encontrar em um repositório local, maven tenta buscar a dependência em repositórios remotos

Repositórios Locais

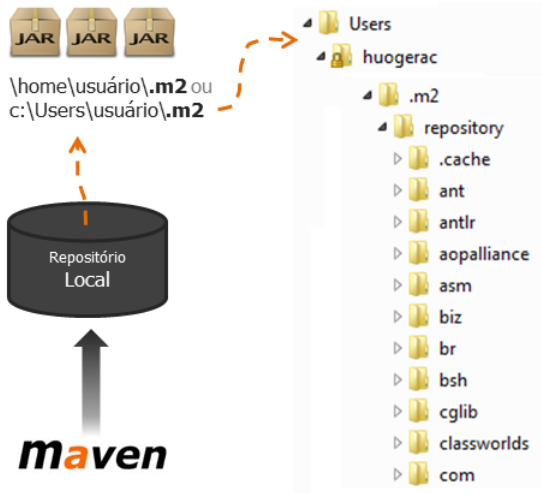


Figure: Fonte: <http://blog.billcode.com.br/2011/09/repositorios-maven-em-5-minutos.html>

Repositórios Internos

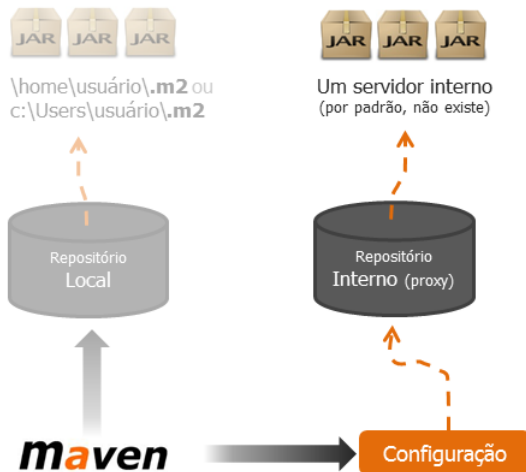


Figure: Fonte: <http://blog.billcode.com.br/2011/09/repositorios-maven-em-5-minutos.html>

Repositórios Remotos

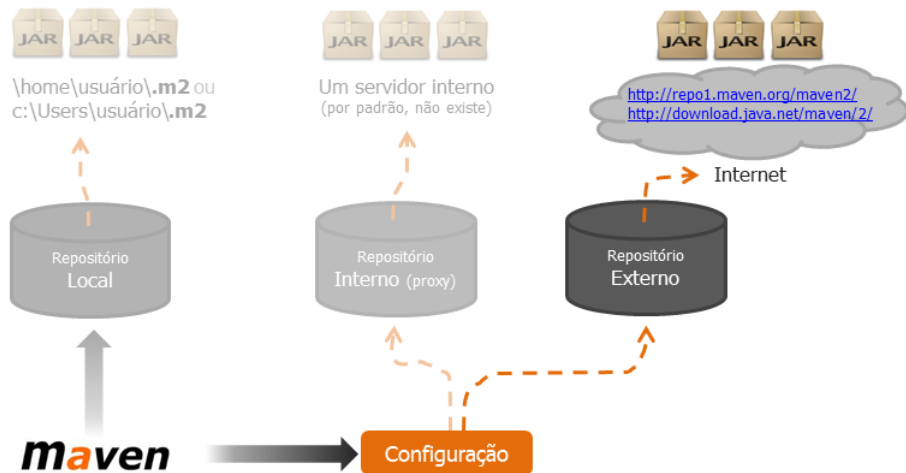


Figure: Fonte: <http://blog.billcode.com.br/2011/09/repositorios-maven-em-5-minutos.html>

Iniciando um projeto com Maven

O que nos aguarda:

- Preparando-se para usar o Maven
- Criando o seu primeiro projeto
- Compilando os arquivos fontes da aplicação
- Compilando os arquivos fontes de teste e executando os testes de unidade
- Empacotando e publicando a aplicação no seu repositório local

Preparando-se para usar o Maven

Configurar o ambiente de desenvolvimento:

- Instalar o Maven
- Iniciar o repositório local

Criar um projeto:

- Configuração inicial
- Estrutura de pastas

Preparando-se para usar o Maven

Executar principais tarefas:

- `compile`
- `clean`
- `test`
- `test-compile`
- `package`
- `install`

Configurando o ambiente de desenvolvimento

Instalar o JDK:

- Preferência pela última versão
- `http://www.oracle.com/technetwork/java/javase/downloads/`

Instalar o Maven:

- Fazer o download do .zip na página do Maven
- `http://maven.apache.org/download.html`

Configurando o ambiente de desenvolvimento

WINDOWS - Descompactar em uma pasta:

- `<HOME>\Aplicativos\apache-maven-X.Y.Z\}`

WINDOWS - Configurar variáveis de ambiente

- `JAVA_HOME=C:\Program Files\Java\jdk1.X.Y_ZW`
`PATH=%PATH%;`
`C:\Program Files\Java\jdk1.X.Y_ZW\bin;`
`<HOME>\Aplicativos\svn-win32-1.7.2\bin;`
`<HOME>\Aplicativos\apache-maven-2.2.1\bin`

Criando o seu primeiro projeto

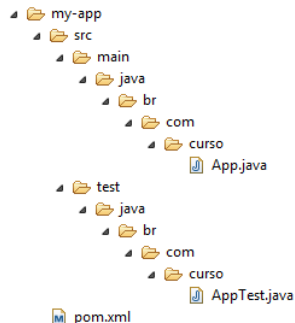
Usaremos o mecanismo de **Archetype**: *template* de projeto

Comando:

```
mvn archetype:create  
    -DgroupId=br.com.curso  
    -DartifactId=my-app  
  
mvn archetype:generate
```

Criando o seu primeiro projeto

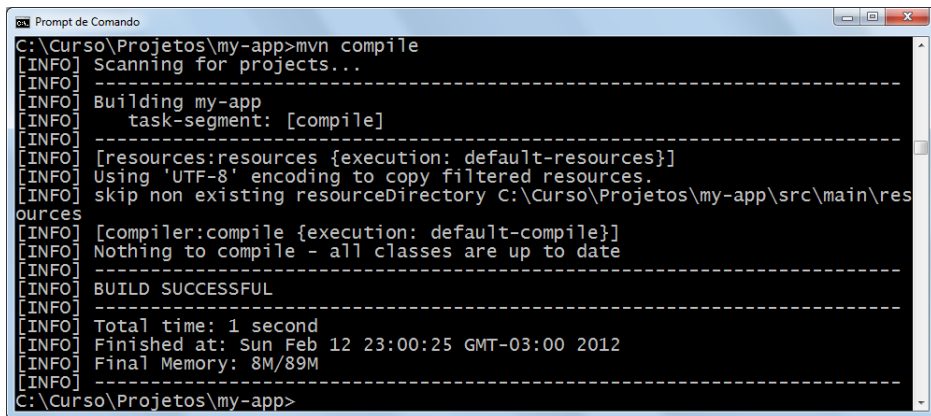
- Um diretório chamado “my-app” é gerado contendo um arquivo “pom.xml”
- O diretório “src” contém todas as entradas necessárias para compilar, testar, documentar e publicar o projeto



```
1 <project xmlns="http://maven.apache.org/POM/4.0.0"
2   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
4     http://maven.apache.org/xsd/maven-4.0.0.xsd">
5   <modelVersion>4.0.0</modelVersion>
6   <groupId>br.com.curso</groupId>
7   <artifactId>my-app</artifactId>
8   <version>1.0-SNAPSHOT</version>
9   <packaging>jar</packaging>
10  <name>my-app</name>
11  <url>http://maven.apache.org</url>
12  <properties>
13    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
14  </properties>
15  <dependencies>
16    <dependency>
17      <groupId>junit</groupId>
18      <artifactId>junit</artifactId>
19      <version>3.8.1</version>
20      <scope>test</scope>
21    </dependency>
22  </dependencies>
23 </project>
```

Compilando a aplicação

```
mvn compile
```



```
Prompt de Comando
C:\Curso\Projetos\my-app>mvn compile
[INFO] Scanning for projects...
[INFO] -----
[INFO] Building my-app
[INFO]    task-segment: [compile]
[INFO] -----
[INFO] [resources:resources {execution: default-resources}]
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory C:\Curso\Projetos\my-app\src\main\resources
[INFO] [compiler:compile {execution: default-compile}]
[INFO] Nothing to compile - all classes are up to date
[INFO] -----
[INFO] BUILD SUCCESSFUL
[INFO] -----
[INFO] Total time: 1 second
[INFO] Finished at: Sun Feb 12 23:00:25 GMT-03:00 2012
[INFO] Final Memory: 8M/89M
[INFO] -----
C:\Curso\Projetos\my-app>
```


Compilando a aplicação

O simplório comando de compilação usa os quatro princípios que fundamentam o Maven:

Convenções sobre a configuração

- Como o maven sabe onde estão os arquivos para serem compilados?
[src/main/java]
- Como o maven sabe onde por as classes compiladas?
[target/classes]
- Os valores de propriedades *defaults* são herdados do Super POM
`mvn help:effective-pom`

Reuso de lógicas de builds

- O que realmente compilou o código?
- O *plugin* padrão *compiler* junto com suas configurações *defaults*

Execução declarativa

- Como o maven sabe que a fase `compile` deve chamar o *plugin* `compiler` e seu comando `compiler:compile`?
- Através do ciclo de vida de *build* padrão

Compilando a aplicação

Organização coerente de dependências

- Como maven é capaz de obter o *plugin* compiler se na instalação padrão do *plugin* não é disponibilizado?
- Maven faz o download a partir de um repositório remoto do *plugin*

```
1 [INFO] Scanning for projects...
2 [INFO] -----
3 [INFO] Building my-app
4 [INFO]    task-segment: [compile]
5 [INFO] -----
6 Downloading:
  http://repo1.maven.org/maven2/org/apache/maven/plugins/maven-resources-plugin/2.2/maven-re
  sources-plugin-2.2.pom
7 1K downloaded
8 Downloading:
  http://repo1.maven.org/maven2/org/apache/maven/plugins/maven-resources-plugin/2.2/maven-re
  sources-plugin-2.2.jar
9 13K downloaded
10 Downloading:
   http://repo1.maven.org/maven2/org/apache/maven/plugins/maven-compiler-plugin/2.0.2/maven-c
   ompiler-plugin-2.0.2.pom
11 2K downloaded
12 Downloading:
   http://repo1.maven.org/maven2/org/apache/maven/plugins/maven-compiler-plugin/2.0.2/maven-c
   ompiler-plugin-2.0.2.jar
13 17K downloaded
```

Configurando um repositório

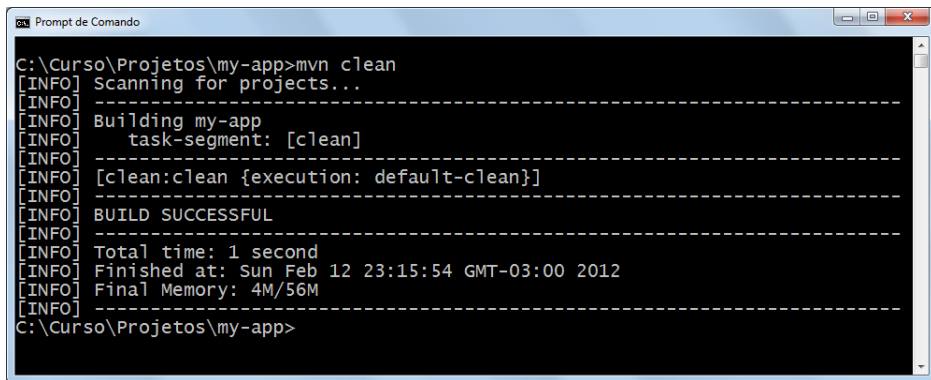
Repositórios:

```
1 <repositories>
2     <repository>
3         <id>central</id>
4         <name>Maven Repository Switchboard</name>
5         <url>http://repo1.maven.org/maven2</url>
6     </repository>
7 </repositories>
8 <pluginRepositories>
9     <pluginRepository>
10         <releases>
11             <updatePolicy>never</updatePolicy>
12         </releases>
13         <id>central</id>
14         <name>Maven Plugin Repository</name>
15         <url>http://repo1.maven.org/maven2</url>
16     </pluginRepository>
17 </pluginRepositories>
```

Excluindo os arquivos gerados

Para excluir os arquivos gerados na pasta target use o comando:

```
mvn clean
```



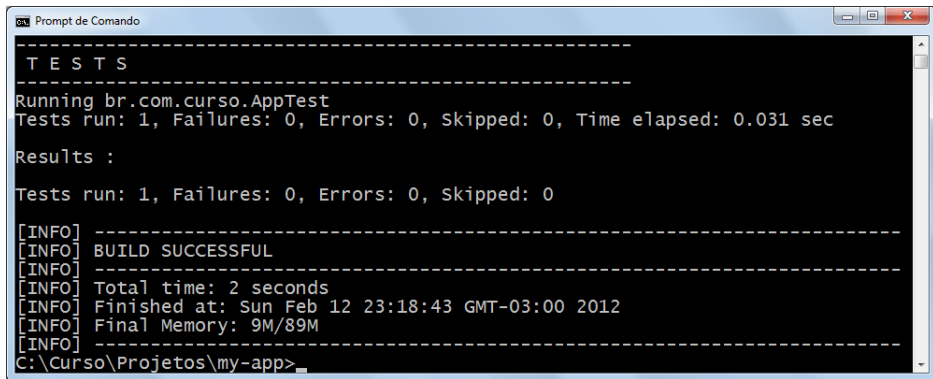
```

C:\Curso\Projetos\my-app>mvn clean
[INFO] Scanning for projects...
[INFO] -----
[INFO] Building my-app
[INFO]    task-segment: [clean]
[INFO] -----
[INFO] [clean:clean {execution: default-clean}]
[INFO] -----
[INFO] BUILD SUCCESSFUL
[INFO] -----
[INFO] Total time: 1 second
[INFO] Finished at: Sun Feb 12 23:15:54 GMT-03:00 2012
[INFO] Final Memory: 4M/56M
[INFO] -----
C:\Curso\Projetos\my-app>
```

Compilando os arquivos fontes de testes de unidade e executando

Para compilar e executar os testes de unidade basta executar o comando:

```
mvn test
```



```
Prompt de Comando

-----
T E S T S
-----
Running br.com.curso.AppTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.031 sec

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

[INFO] -----
[INFO] BUILD SUCCESSFUL
[INFO] -----
[INFO] Total time: 2 seconds
[INFO] Finished at: Sun Feb 12 23:18:43 GMT-03:00 2012
[INFO] Final Memory: 9M/89M
[INFO] -----
C:\Curso\Projetos\my-app>
```

Compilando os arquivos fontes de testes de unidade e executando

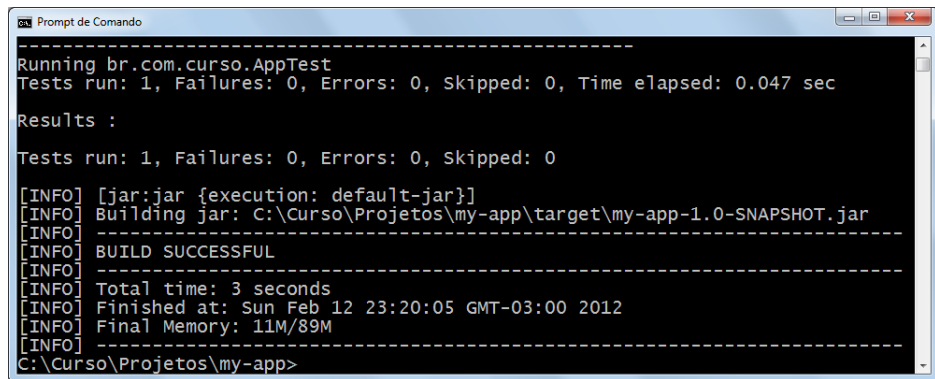
Para compilar e **NÃO** executar os testes de unidade basta executar o comando:

```
mvn test-compile
```

Empacotando e publicando a aplicação no seu repositório local

Para empacotar a aplicação basta executar o comando:

```
mvn package
```



```
Prompt de Comando

-----
Running br.com.curso.AppTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.047 sec

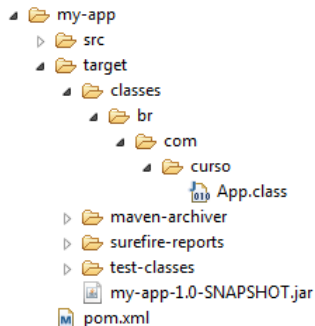
Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

[INFO] [jar:jar {execution: default-jar}]
[INFO] Building jar: C:\Curso\Projetos\my-app\target\my-app-1.0-SNAPSHOT.jar
[INFO] -----
[INFO] BUILD SUCCESSFUL
[INFO] -----
[INFO] Total time: 3 seconds
[INFO] Finished at: Sun Feb 12 23:20:05 GMT-03:00 2012
[INFO] Final Memory: 11M/89M
[INFO] -----
C:\Curso\Projetos\my-app>
```


Empacotando e publicando a aplicação no seu repositório local

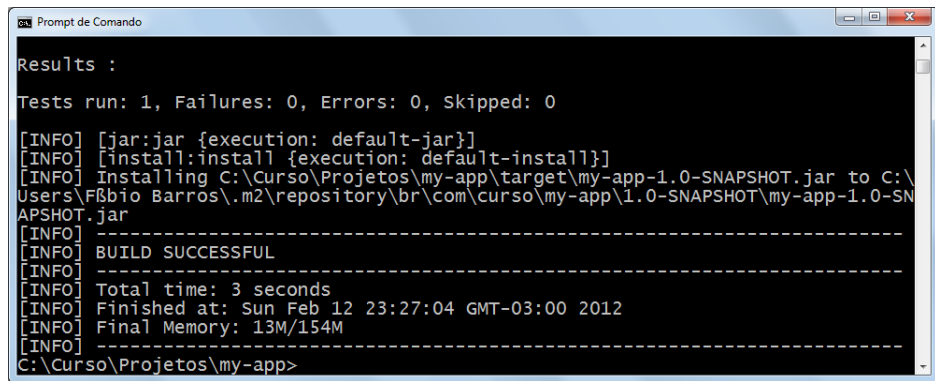
O diretório target conterá o arquivo JAR gerado



Empacotando e publicando a aplicação no seu repositório local

Para instalar a aplicação no repositório local basta executar o comando:

```
mvn install
```



```
Prompt de Comando

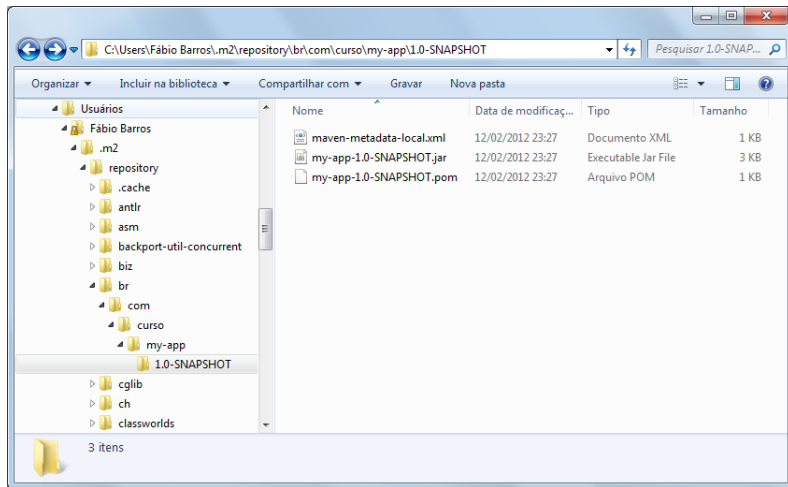
Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

[INFO] [jar:jar {execution: default-jar}]
[INFO] [install:install {execution: default-install}]
[INFO] Installing C:\Curso\Projetos\my-app\target\my-app-1.0-SNAPSHOT.jar to C:\
Users\Fabio Barros\.m2\repository\br\com\curso\my-app\1.0-SNAPSHOT\my-app-1.0-SN
APSHOT.jar
[INFO] -----
[INFO] BUILD SUCCESSFUL
[INFO] -----
[INFO] Total time: 3 seconds
[INFO] Finished at: Sun Feb 12 23:27:04 GMT-03:00 2012
[INFO] Final Memory: 13M/154M
[INFO] -----
C:\Curso\Projetos\my-app>
```

Empacotando e publicando a aplicação no seu repositório local

Repositório Local após o install



Manipulando recursos de *classpath*

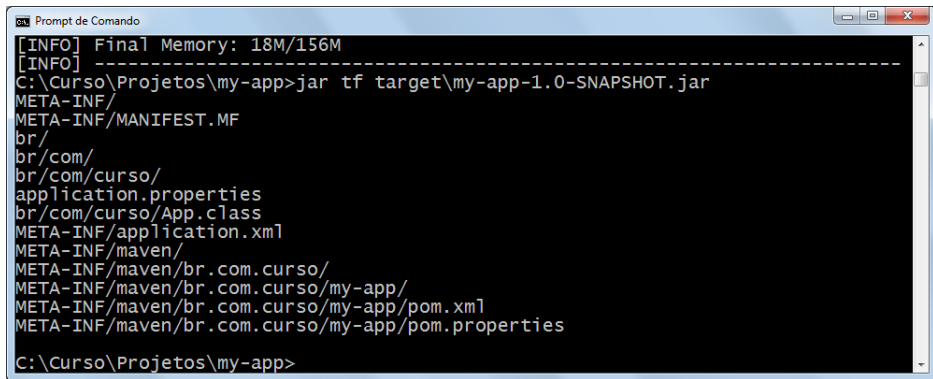
- Às vezes é necessário empacotar arquivos de recursos dentro de um arquivo JAR
- Maven fornece um diretório padrão para os recursos:
`[src/main/resources]`
- Os arquivos colocados nesse diretório serão empacotados dentro do JAR com a mesma estrutura

Exercício

- Crie o diretório: `[src/main/resources/META-INF]`
- Dentro de `[src/main/resources]`, crie o arquivo vazio `application.properties`
- Dentro de `[src/main/resources/META-INF]`, crie o arquivo vazio `application.xml`
- Execute `mvn package`
- Execute `jar tvf target/my-app-1.0-SNAPSHOT.jar`

Manipulando recursos de *classpath*

Veja que os arquivos dentro da pasta `resources`, foram empacotados no JAR



```
cmd Prompt de Comando
[INFO] Final Memory: 18M/156M
[INFO] -----
C:\Curso\Projetos\my-app>jar tf target\my-app-1.0-SNAPSHOT.jar
META-INF/
META-INF/MANIFEST.MF
br/
br/com/
br/com/curso/
application.properties
br/com/curso/App.class
META-INF/application.xml
META-INF/maven/
META-INF/maven/br.com.curso/
META-INF/maven/br.com.curso/my-app/
META-INF/maven/br.com.curso/my-app/pom.xml
META-INF/maven/br.com.curso/my-app/pom.properties
C:\Curso\Projetos\my-app>
```

Manipulando recursos de *classpath* para testes de unidade

Exercício

- Para adicionar recursos ao *classpath* dos testes de unidade, siga o mesmo padrão para um JAR
- Maven fornece um diretório padrão para os recursos:
[src/**test**/resources]
- Os arquivos colocados nesse diretório estarão disponíveis durante a execução dos testes de unidade

Manipulando recursos de *classpath* para testes de unidade

Suponha que exista o artefato:

`my-app/src/test/resources/test.properties`

Uma classe de testes poderia usá-lo da seguinte forma:

```
1 [...]
2 // Recuperar resource
3 InputStream is = getClass().getResourceAsStream( "/test.properties" );
4 // Faz alguma coisa com o resource
5 [...]
```


- Às vezes, um arquivo de recurso precisará conter um valor que somente poderá ser fornecido durante a construção do software (execução do maven)
- Para **filtrar** arquivos de recursos dinamicamente é necessário usar uma propriedade no “formato maven”

`${<nome da propriedade>}`

Essa propriedade pode estar definida em:

- `pom.xml`
- `settings.xml`
- Um arquivo de propriedades externo
- Propriedade de sistema

Filtrando Recursos de *Classpath*

Para filtrar propriedades é necessário ativar o processo de filtragem:

```
1 <project>
2   ...
3   <properties>
4     <minha.propriedade>valor</minha.propriedade>
5   </properties>
6   <build>
7     <resources>
8       <resource>
9         <directory>src/main/resources</directory>
10        <filtering>true</filtering>
11      </resource>
12    </resources>
13  </build>
14  <dependencies>
15    ...
16  </dependencies>
17 </project>
```

Referenciando propriedades definidas no pom.xml

```
${minha.propriedade}  
${project.name}  
${project.version}  
${project.build.finalName}
```

Exercício

- Edite o arquivo:
`src/main/resources/application.properties`
- Insira o seguinte conteúdo no arquivo:

```
# application.properties
application.name=${project.name}
application.version=${project.version}
minha.propriedade=${minha.propriedade}
```

- Execute o seguinte comando:
`mvn process-resources`

Referenciando propriedades definidas em um arquivo externo

Exercício

- Crie o arquivo
src/main/**filters**/filter.properties
- Insira o seguinte conteúdo no arquivo:

```
# filter.properties  
my.filter.value=ola!
```

Referenciando propriedades definidas em um arquivo externo

Adicione a referência no pom.xml

```
1 <project>
2   <!-- ... -->
3   <build>
4     <filters>
5       <filter>src/main/filters/filter.properties</filter>
6     </filters>
7     <resources>
8       <resource>
9         <directory>src/main/resources</directory>
10        <filtering>true</filtering>
11      </resource>
12    </resources>
13  </build>
14  <dependencies>
15    <!-- ... -->
16  </dependencies>
17 </project>
```

Referenciando propriedades definidas em um arquivo externo

Exercício

- Adicionar a referência a propriedade em:
`src/main/resources/application.properties`
- # `application.properties`
`application.name=${project.name}`
`application.version=${project.version}`
`minha.propriedade=${minha.propriedade}`
`mensagem=${my.filter.value}`
- Execute o seguinte comando:
`mvn process-resources`
- Verifique o resultado em:
`my-app/target/classes`

Referenciando propriedades de sistema e em linha de comando

Exercício

- Propriedades de sistema:
`java.home`
`user.home`
- Propriedades definidas em linha de comando usando o parâmetro padrão do java `-D`
`mvn process-resources "-Dpropriedade=valor"`

Referenciando propriedades de sistema e em linha de comando

Exercício

- Acrescente no arquivo:
`src/main/resources/application.properties`
`java.version=${java.version}`
`user.home=${user.home}`
`command.line.prop=${command.line.prop}`
- Execute o seguinte comando:
`mvn process-resources -Dcommand.line.prop="Ola novamente!"`
- Verifique o resultado em:
`my-app/target/classes`

Introdução ao Maven

Parte 2

Mecanismo de Dependência

- **Dependências transitivas** são uma nova característica no Maven 2.0
- Permite evitar a necessidade de descobrir e especificar as bibliotecas que as suas próprias dependências necessitam, pois são incluídas automaticamente.
- Viabilizada pela leitura dos arquivos de projeto de suas dependências a partir dos repositórios remotos especificados.

Exercício

Adicione a seguinte dependência ao projeto ‘my-app’

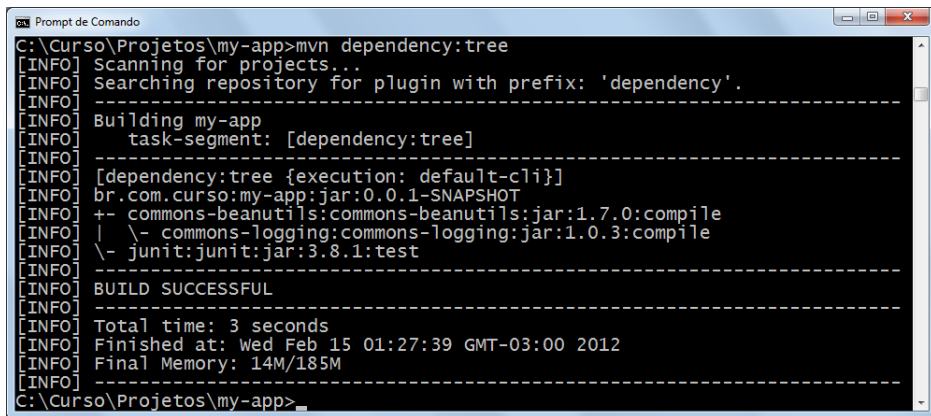
```
1 <dependency>
2     <groupId>commons-beanutils</groupId>
3     <artifactId>commons-beanutils</artifactId>
4     <version>1.7.0</version>
5 </dependency>
```

Execute o comando abaixo para ver como Maven gerencia as novas dependências:

```
mvn dependency:tree
```

Mecanismo de Dependência

```
mvn dependency:tree
```



```

C:\Curso\Projetos\my-app>mvn dependency:tree
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'dependency'.
[INFO] -----
[INFO] Building my-app
[INFO]    task-segment: [dependency:tree]
[INFO] -----
[INFO] [dependency:tree {execution: default-cli}]
[INFO] br.com.curso:my-app:jar:0.0.1-SNAPSHOT
[INFO] +- commons-beanutils:commons-beanutils:jar:1.7.0:compile
[INFO] |   \- commons-logging:commons-logging:jar:1.0.3:compile
[INFO] \- junit:junit:jar:3.8.1:test
[INFO] -----
[INFO] BUILD SUCCESSFUL
[INFO] -----
[INFO] Total time: 3 seconds
[INFO] Finished at: Wed Feb 15 01:27:39 GMT-03:00 2012
[INFO] Final Memory: 14M/185M
[INFO] -----
C:\Curso\Projetos\my-app>
```

Escopo de Dependência

- Todas as dependências são usadas no projeto, sejam herdadas de projetos pai ou a partir das dependências.
- O escopo da dependência é usado para limitar a transitividade de uma dependência. Há cinco escopos disponíveis:

Compile

Provided

Runtime

Test

System

Escopo de Dependência

compile: esse é o escopo padrão, usado quando nenhum é especificado.

- Dependências de compilação estão disponíveis em todos os *classpath*s de um projeto
- Essas dependências são propagadas para os projetos dependentes.

provided: muito parecido com o compile, mas indica que o JDK ou um contêiner fornecerá a dependência em tempo de execução.

- Por exemplo, quando construindo um aplicação web para J2EE, você configurará a dependência da API do Servlet e as APIs Java EE relacionadas ao escopo de provided porque o contêiner web fornece essas classes.
- Esse escopo está somente disponível para o classpath de compilação e não é transitivo.

Escopo de Dependência

runtime: indica que a dependência não é necessária para compilação, mas é para execução. Está no *classpath* de *runtime* e *test*, mas não no *classpath* de compile.

test: esse escopo indica que a dependência não é necessária para uso normal da aplicação e é somente necessária para a compilação e execução da fase de teste.

Escopo de Dependência

system: esse escopo é similar ao **provided** exceto que você tem que fornecer o JAR que o contém explicitamente. O artefato está sempre disponível e nunca é procurado no repositório.

```
1 <dependency>
2   <groupId>com.mycorp</groupId>
3   <artifactId>my-lib</artifactId>
4   <version>1.0.0</version>
5   <scope>system</scope>
6   <systemPath>${basedir}/src/main/lib/my-lib-1.0.0.jar</systemPath>
7 </dependency>
```

Escopo de Dependência

Um dependência com escopo de `system` para ser empacotada junto com a aplicação tem que estar em:

```
${basedir}/src/main/lib/
```

Isso significa que:

- A dependência deve ser adicionada ao controle de versão.
- Caso contrário, alguém pode não conseguir executar a aplicação em outra máquina/ambiente.
- O servidor de Integração Contínua pode falhar ao tentar realizar uma build pois não encontrará a dependência.

É possível instalar uma dependência no repositório local

```
1 <dependency>
2     <groupId>br.teste</groupId>
3     <artifactId>dependencia-teste</artifactId>
4     <version>1.1.2</version>
5 </dependency>
```

```
mvn install:install-file
  -DgroupId=br.teste
  -DartifactId=dependencia-teste
  -Dversion=1.1.2
  -Dpackaging=jar
  -Dfile=dependencia-teste-1.1.2.jar
  -DcreateChecksum=true
  -DgeneratePom=true
```

Escopo de Dependência

- Para a abordagem anterior funcionar em outras máquinas/ambientes é necessário que cada máquina/ambiente instale a dependência no repositório local
- A IC-Integração Contínua pode falhar caso a dependência não seja instalada no repositório local que a IC usa
- Melhor caso:
Instalar a dependência no repositório de rede
Assim todos (IC e outros usuários) terão disponível a dependência

Perfis (*Profiles*)

Perfis (*Profiles*) são os meios pelos quais o Maven permite ao desenvolvedor criar **variações de ambiente** no ciclo de vida para acomodar tarefas como:

- Configurar o projeto em plataformas diferentes
- Com JVMs diferentes
- Testar com bancos de dados diferentes
- Referenciar os arquivos locais de sistema
- Gerar a aplicação com parâmetros de compilação diferentes

Perfis (*Profiles*)

Perfis (*Profiles*) são os meios pelos quais o Maven permite ao desenvolvedor criar **variações de ambiente** no ciclo de vida para acomodar tarefas como:

- Tipicamente, o desenvolvedor deverá encapsular o máximo possível de perfis no POM para garantir que o processo de **build** seja **portável**
- Às vezes, o deve-se levar em consideração as variações dos sistemas (SO, BD, JVM, endereço de servidores), esse é o motivo pelo qual os perfis foram incluídos no Maven
- Perfis são especificados usando um subconjunto dos elementos disponíveis no POM e **podem ser ativados** de diversas formas

Perfis (*Profiles*)

Perfis (*Profiles*) são os meios pelos quais o Maven permite ao desenvolvedor criar **variações de ambiente** no ciclo de vida para acomodar tarefas como:

- Perfis modificam o POM em tempo de build. São usados como um conjunto complementar de parâmetros para o conjunto de ambientes alvo
O caminho para a raiz do servidor de aplicação: desenvolvimento, testes, homologação, produção, etc.
Parâmetros de acesso a banco de dados: desenvolvimento, testes, homologação, produção, etc.
- **Atenção:** Os perfis podem levar a resultados diferentes de builds partindo de membros diferentes de uma equipe
- **Perfis preservam a portabilidade de build**

Onde definir um perfil?

- No arquivo Maven de configuração para usuário:

```
<user_home>/m2/settings.xml
```

- Em um arquivo no mesmo diretório do POM, chamado de `profiles.xml`
- No próprio POM

- Perfis em `settings.xml` tem o potencial de afetar todos as builds, logo eles são um tipo perfil **global**
- `profiles.xml` permite que estendam um projeto sem alterar o POM
- Perfis baseados em POM são recomendados, desde que esses sejam portáteis, ou seja, eles serão publicados no repositório de distribuição e ficarão disponibilizados para builds subsequentes

Perfil usando profiles.xml

Crie o arquivo profiles.xml no mesmo nível que o pom.xml

```
1 <?xml version="1.0" encoding="ISO-8859-1"?>
2 <profilesXml xmlns="http://maven.apache.org/POM/4.0.0"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
5   http://maven.apache.org/xsd/profiles-1.0.0.xsd">
6   <profiles>
7     <profile>
8       <id>desenvolvimento</id>
9       <properties>
10        <command.line.prop>Valor do profile.xml</command.line.prop>
11      </properties>
12      <activation>
13        <property>
14          <name>ambiente</name>
15          <value>desenvolvimento</value>
16        </property>
17      </activation>
18    </profile>
19  </profiles>
20 </profilesXml>
```

Ativação de um Perfil

- Explicitamente usando a opção: **-P**
- Essa opção usa um argumento que contém uma lista de identificadores de perfis delimitados por vírgula
- Quando essa opção é especificada, nenhum outro perfil será ativado

```
mvn clean install -Pprofile1,profile2
```

```
mvn clean install -Pdesenvolvimento
```

```
mvn clean install -Dambiente=desenvolvimento
```

- Veja como o arquivo `application.properties` ficou

Ativação de um Perfil

- Usando o `settings.xml` do Maven, através da seção `activeProfiles`
- Essa seção usa uma lista de elementos de `activeProfile`, cada um contendo um identificador de perfil.

```
1 <settings>
2   [...]
3   <profiles>
4     <profile>
5       <id>profile1</id>
6       [...]
7     </profile>
8   </profiles>
9   <activeProfiles>
10    <activeProfile>profile1</activeProfile>
11  </activeProfiles>
12  [...]
13 </settings>
```

Ativação de um Perfil

Crie o arquivo

`<user_home>/ .m2/settings.xml`

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <settings xmlns="http://maven.apache.org/POM/4.0.0"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
5     http://maven.apache.org/xsd/settings-1.0.0.xsd">
6     <profiles>
7         <profile>
8             <id>curso</id>
9             <properties>
10                 <command.line.prop>Valor do SETTINGS.XML</command.l
11             </properties>
12         </profile>
13     </profiles>
14 </settings>
```

Ativação de um Perfil

- Por exemplo

```
mvn clean install -Pdesenvolvimento,curso
```

```
mvn clean install -Dambiente=desenvolvimento -Pcurso
```

- Veja como o arquivo `application.properties` ficou

Introdução ao Maven

Parte 3

Maven tem a capacidade de gerar relatórios de qualidade da build

- Através de plugins
- Deve ser configurado através da tag `reporting` no `pom.xml`
- Comando abaixo gera todos os relatórios configurados

```
mvn clean site
```

- Descompacte o arquivo `dv-generator.zip` e execute o comando acima

Gerados por plugins

- Checkstyle
- Cobertura Test Coverage
- CPD Report
- DashBoard Report
- FindBugs Report
- JavaDocs
- JavaNCSS Report
- Maven Surefire Report
- PMD Report
- Source Xref
- Test JavaDocs
- Test Source Xref

Executado durante uma fase site da build

Clean Lifecycle

- pre-clean:** executa processos necessários antes da limpeza projeto
- clean:** remove todos os arquivos gerados pela build anterior
- post-clean:** executa os processos necessários para finalizar a limpeza do projeto

Default Lifecycle

validate: valida se o projeto está correto e todas as informações necessárias estão disponíveis

initialize: inicializa o estado da build (definir propriedades ou criar diretórios)

generate-sources: gera código fonte para compilação

process-sources: processar o código fonte (filtragem de valores no código)

generate-resources: gera os recursos a serem incluídos no pacote final

process-resources: copia e processa os recursos para o diretório de destino, deixando pronto para empacotamento

compile: compila o código fonte do projeto

process-classes: pós-processo dos arquivos gerados a partir de compilação

Default Lifecycle

generate-test-sources: gera código fonte de teste para compilação

process-test-sources: processa o código fonte de teste (filtragem de valores no código)

generate-test-resources: cria os recursos para teste

process-test-resources: copia e processa os recursos para o diretório de destino de teste

test-compile: compilar o código fonte de teste no diretório de destino teste

process-test-classes: pós-processa os arquivos gerados a partir de compilação de teste

test: executar testes usando um framework de testes unidade

Default Lifecycle

prepare-package: realiza todas as operações necessárias para preparar um pacote antes do empacotamento. Resulta em uma versão descompactada do pacote.

package: pega o código compilado e empacota em seu formato distribuível (JAR, WAR, EAR, SAR, etc).

pre-integration-test: executa ações necessárias antes dos testes de integração. Pode envolver tarefas como a criação do ambiente.

integration-test: processa e implanta o pacote (se necessário) em um ambiente onde os testes de integração podem ser executados.

post-integration-test: executa ações necessárias após os testes de integração. Pode limpar o meio ambiente.

verify: executa todos os controles para verificar se o pacote é válido e atende aos critérios de qualidade.

Default Lifecycle

- install:** instala o pacote no repositório local, para uso como uma dependência em outros projetos localmente.
- deploy:** feito em um ambiente de integração ou liberação, copia o pacote final para o repositório remoto para compartilhar com outros desenvolvedores e projetos.

Site Lifecycle

pre-site: executa processos necessários antes da geração

site: gera o site do projeto de documentação

post-site: executa os processos necessários para finalizar a geração site

site-deploy: implanta a documentação do site gerado para o servidor da Web especificado

Configurando os Relatórios

```
1 <project>
2   ...
3   <!-- Relatorios -->
4   <reporting>
5     <plugins>
6       <plugin>
7         <groupId>???
```

Configurando os Relatórios

```
1 <!-- Relatórios -->
2 <reporting>
3     <plugins>
4         <plugin>
5             <groupId>org.apache.maven.plugins</groupId>
6             <artifactId>maven-jxr-plugin</artifactId>
7         </plugin>
8         <plugin>
9             <groupId>org.apache.maven.plugins</groupId>
10            <artifactId>maven-surefire-report-plugin</artifactId>
11            <version>2.0</version>
12        </plugin>
13        <plugin>
14            <groupId>org.codehaus.mojo</groupId>
15            <artifactId>cobertura-maven-plugin</artifactId>
16        </plugin>
17        <plugin>
18            <groupId>org.apache.maven.plugins</groupId>
19            <artifactId>maven-checkstyle-plugin</artifactId>
20            <configuration>
21                <configLocation>config/sun_checks.xml</configLocation>
22            </configuration>
23        </plugin>
24    </plugins>
25 </reporting>
```

Plugins mais utilizados

- org.apache.maven.plugins:**maven-jxr-plugin**
- org.apache.maven.plugins:**maven-surefire-report-plugin**
- org.codehaus.mojo:**cobertura-maven-plugin**
- org.codehaus.mojo:**javancss-maven-plugin**
- org.apache.maven.plugins:**maven-checkstyle-plugin**
- org.apache.maven.plugins:**maven-pmd-plugin**
- org.apache.maven.plugins:**maven-javadoc-plugin**
- org.codehaus.mojo:**findbugs-maven-plugin**
- org.codehaus.mojo:**dashboard-maven-plugin**