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.
  - A 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

Princípios do Maven

- 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 Generador</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

```xml
<project>
    <modelVersion>4.0.0</modelVersion>
    <groupId>br.com.curso</groupId>
    <artifactId>dv-generator</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>DV Generador</name>
    <description>Gerador de digito verificador</description>
    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.4</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</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

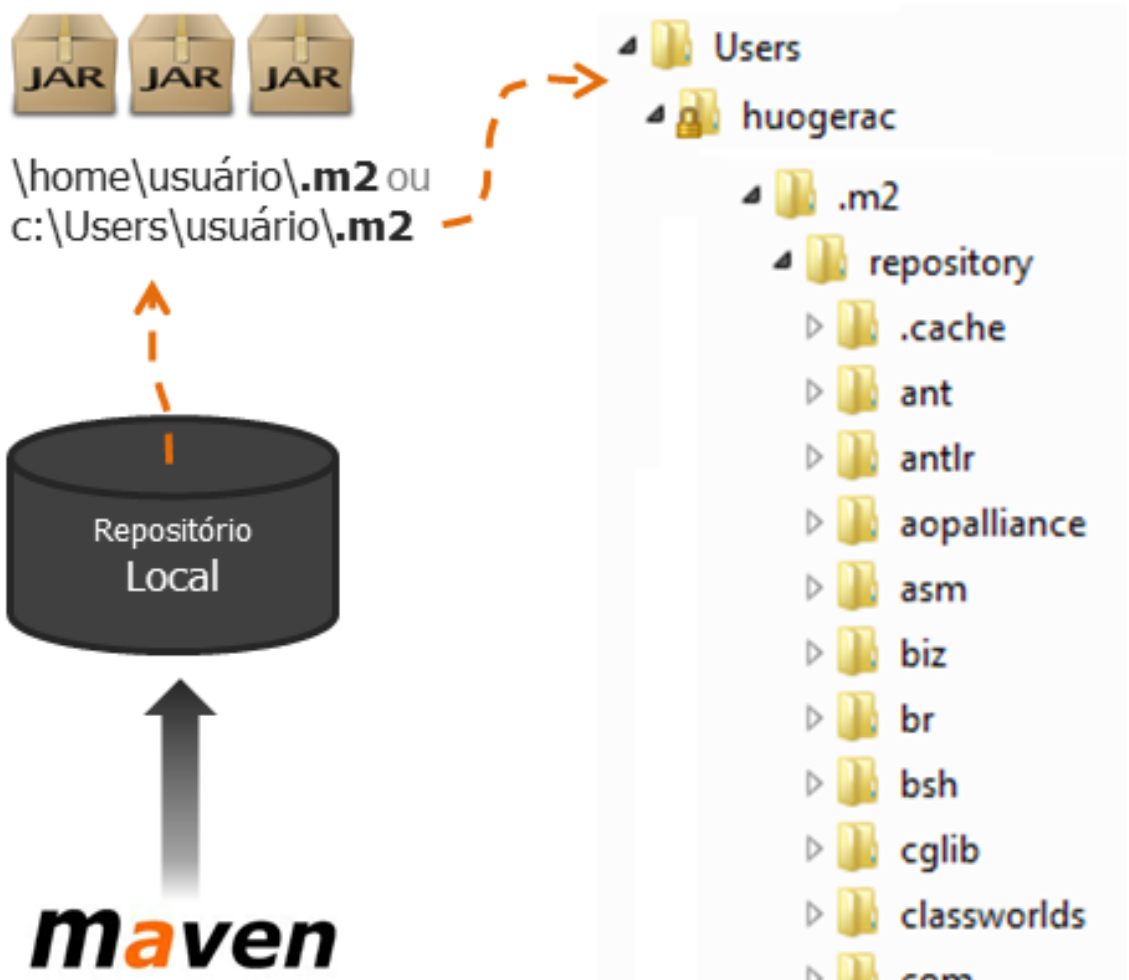


Figure: Fonte: `http://blog.billcode.com.br/2011/09/repositorios-maven-em-5-minutos.html`

Repositórios Internos

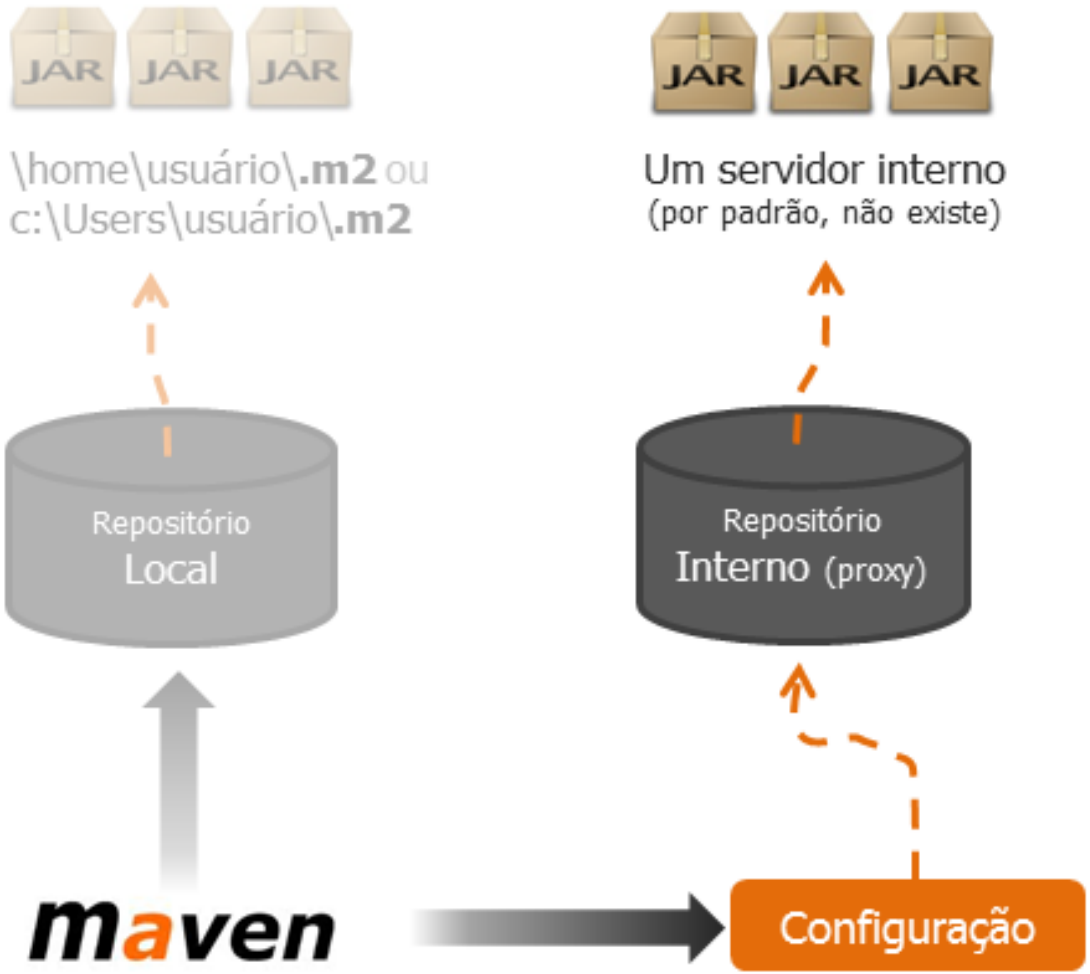


Figure: Fonte: `http://blog.billcode.com.br/2011/09/repositorios-maven-em-5-minutos.html`

Repositórios Remotos

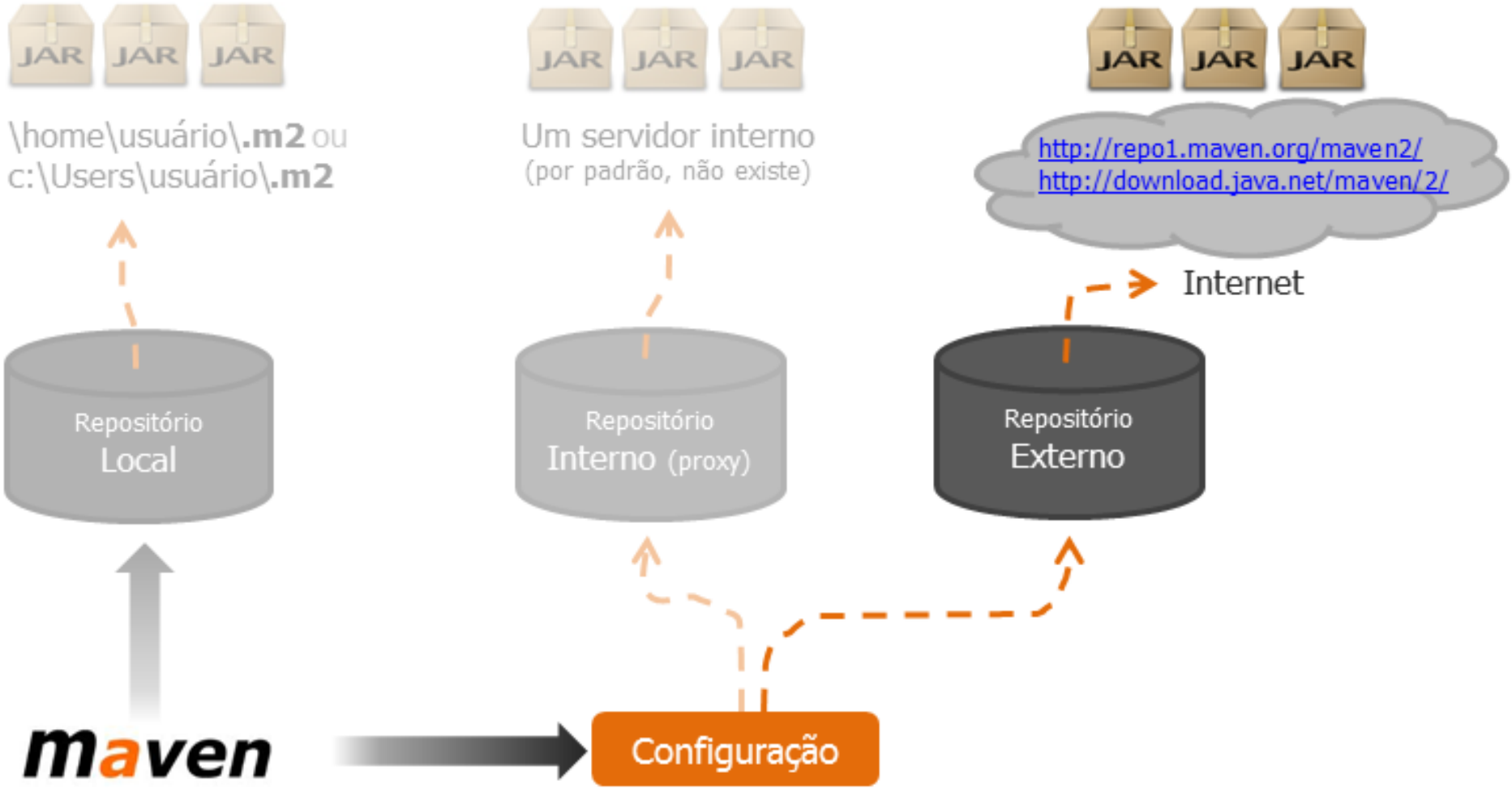


Figure: Fonte: `http://blog.billcode.com.br/2011/09/repositorios-maven-em-5-minutos.html`

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

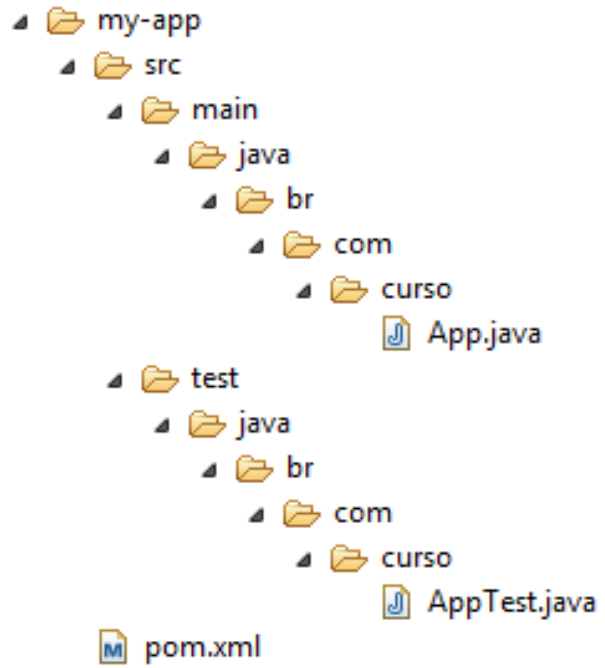
Comando:

```
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

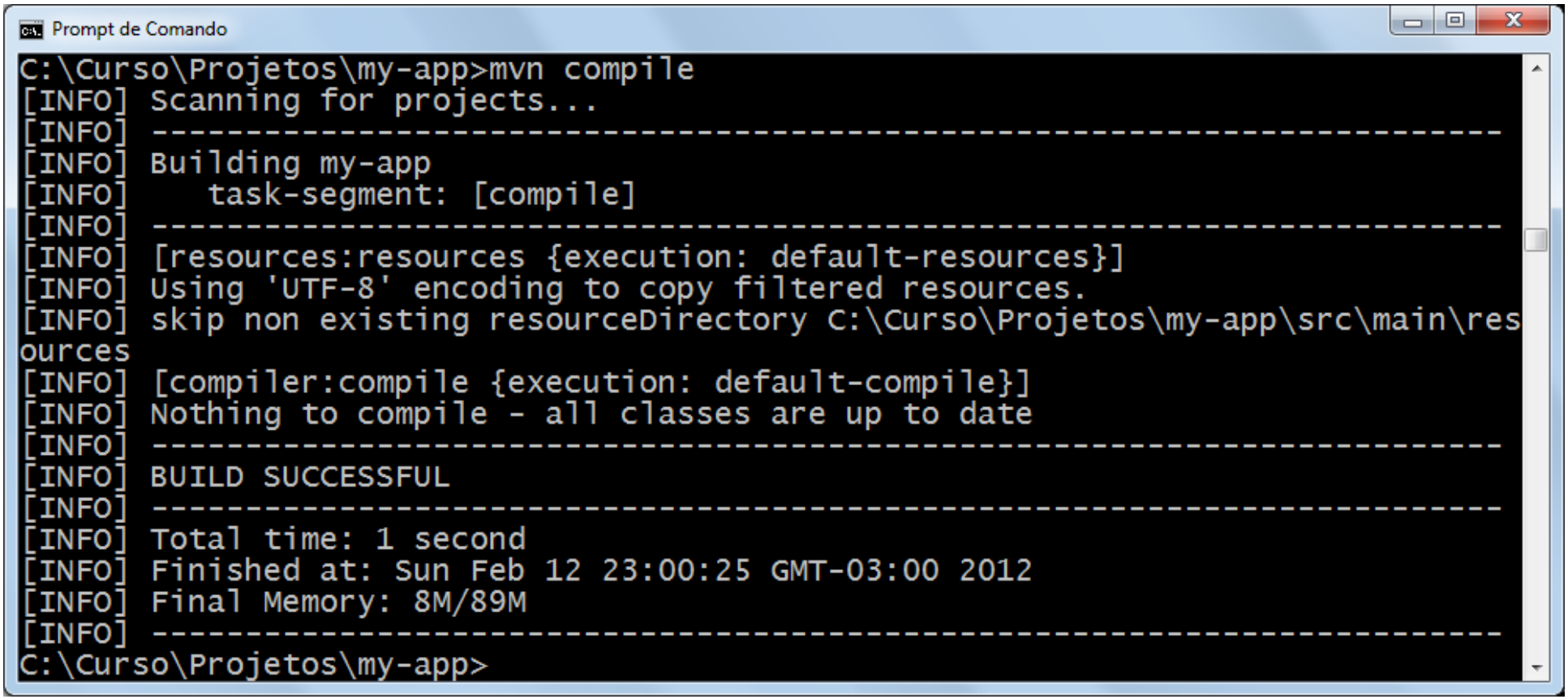


pom.xml

```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.curso</groupId>
  <artifactId>my-app</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <name>my-app</name>
  <url>http://maven.apache.org</url>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

Compilando a aplicação

```
mvn compile
```



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*

Compilando a aplicação

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

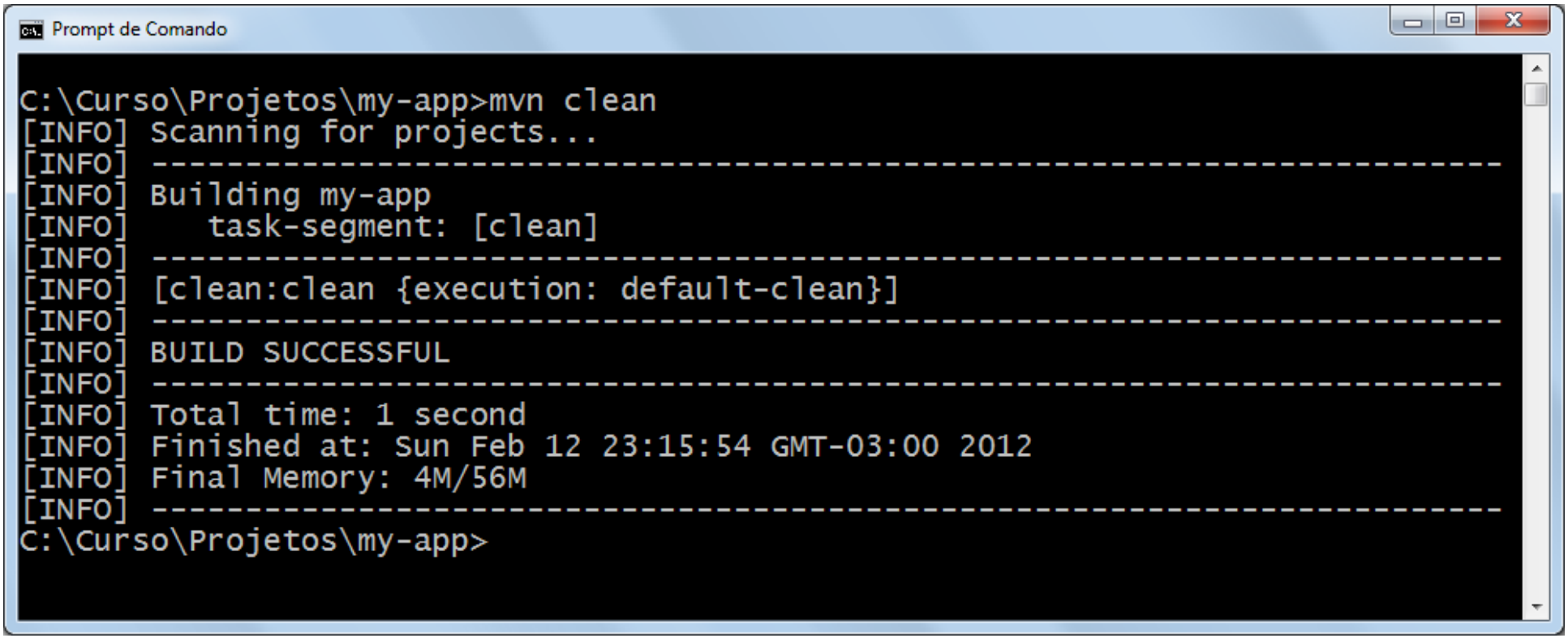
Repositórios:

```
<repositories>
        <repository>
                <id>central</id>
                <name>Maven Repository Switchboard</name>
                <url>http://repo1.maven.org/maven2</url>
        </repository>
</repositories>
<pluginRepositories>
        <pluginRepository>
                <releases>
                        <updatePolicy>never</updatePolicy>
                </releases>
                <id>central</id>
                <name>Maven Plugin Repository</name>
                <url>http://repo1.maven.org/maven2</url>
        </pluginRepository>
</pluginRepositories>
```

Excluindo os arquivos gerados

Para excluir os arquivo gerados na pasta `target` use o comando:

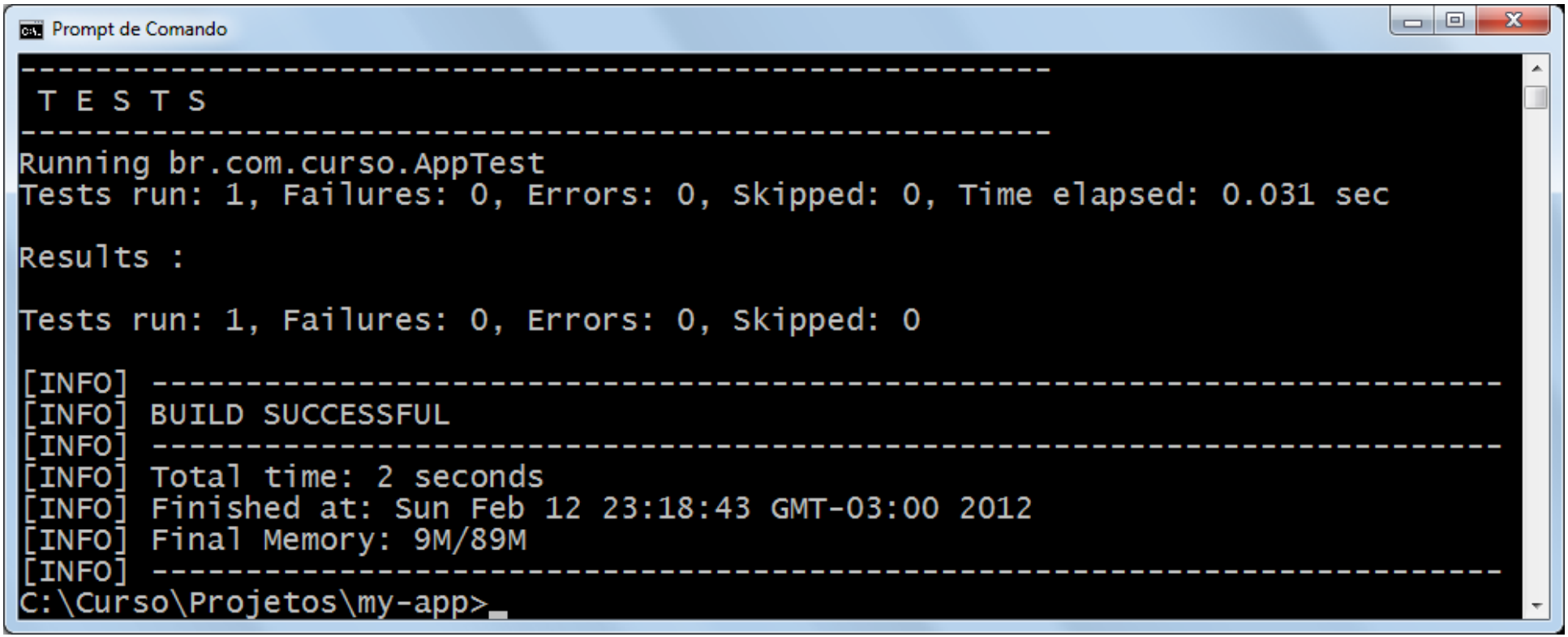
```
mvn clean
```



Compilando os arquivos fontes de testes de unidade e executando

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

```
mvn test
```



Compilando os arquivos fontes de testes de unidade e executando

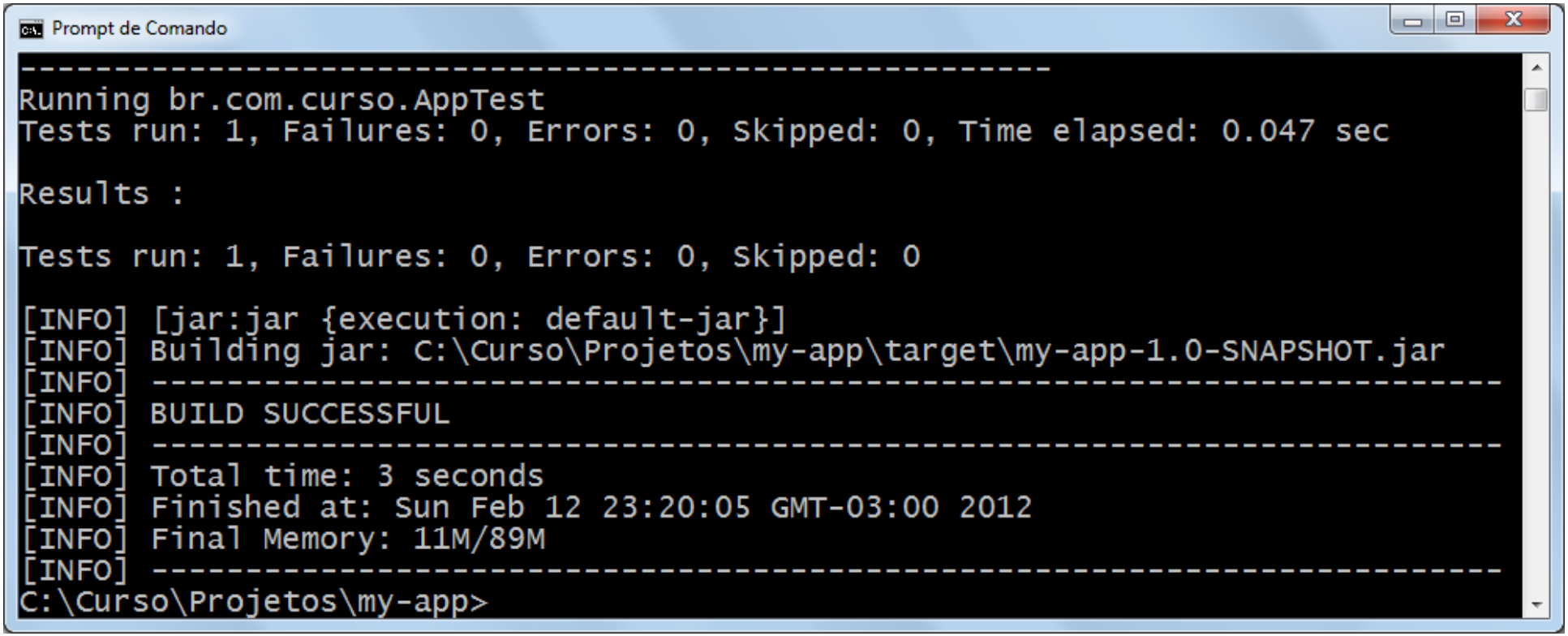
Para compilar e **NÃO** executar os testes de unidade basta executar o comando:

```
mvn test-compile
```

Empacotando e publicando a aplicação no seu repositório local

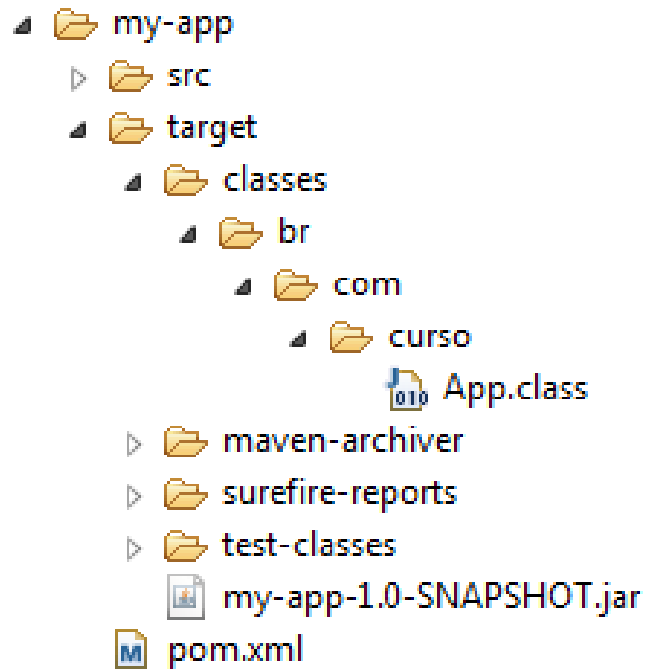
Para empacotar a aplicação basta executar o comando:

```
mvn package
```



Empacotando e publicando a aplicação no seu repositório local

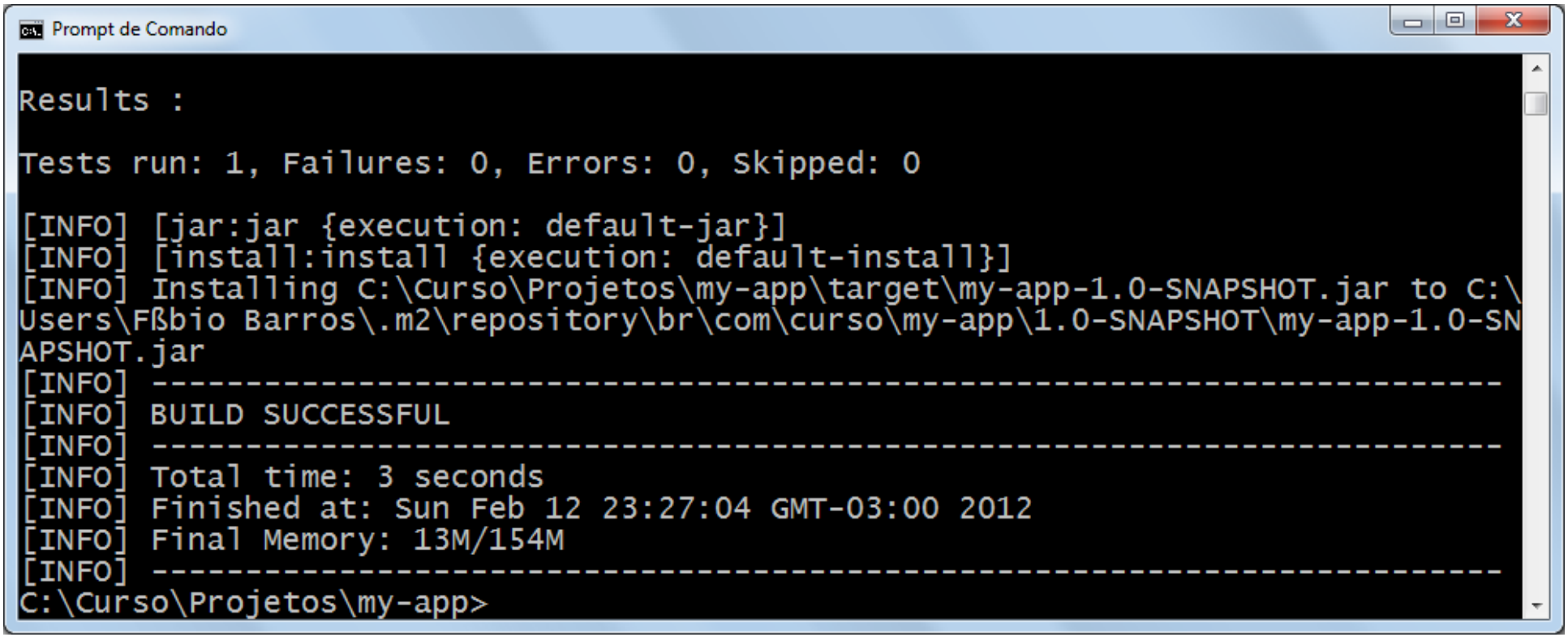
O diretório target conterá o arquivo JAR gerado



Empacotando e publicando a aplicação no seu repositório local

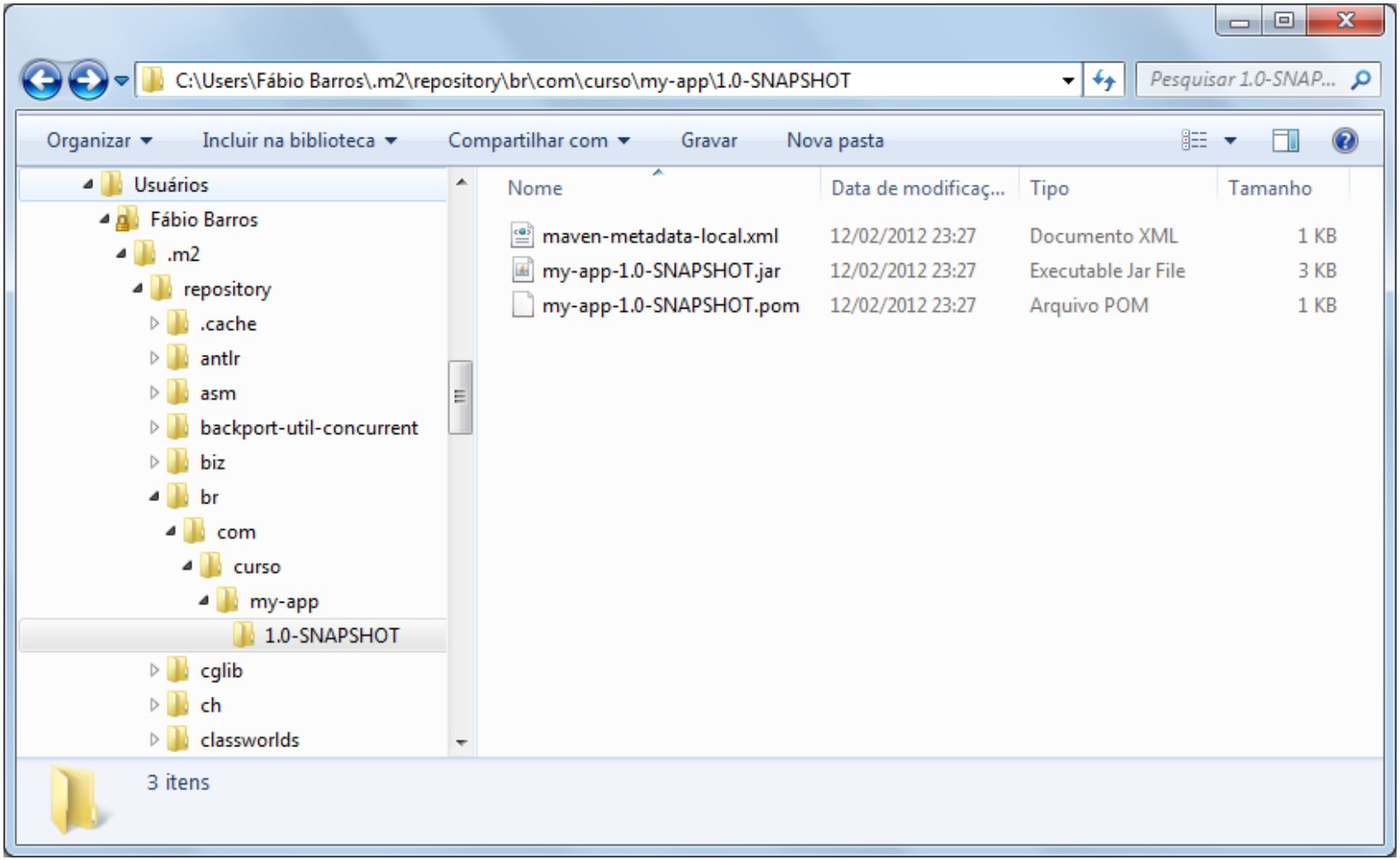
Para instalar a aplicação no repositório local basta executar o comando:

```
mvn install
```



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

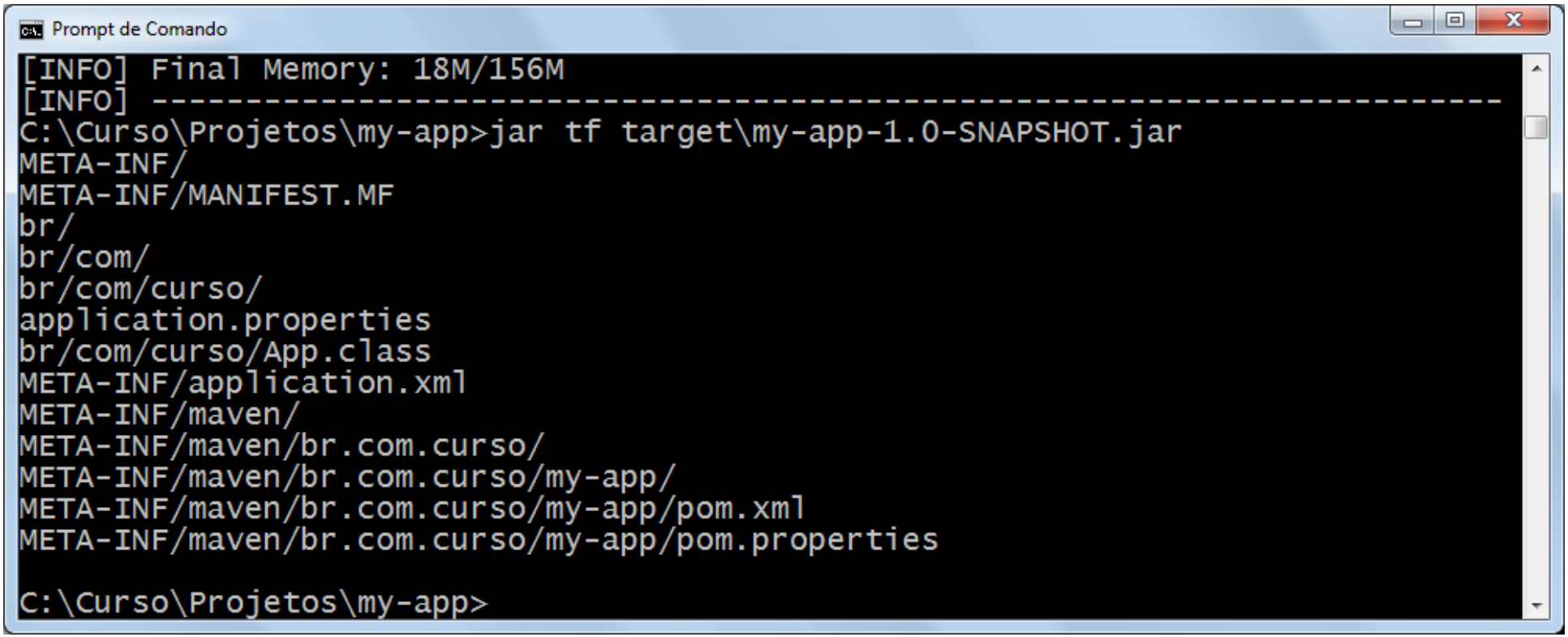
Manipulando recursos de *classpath*

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 arquivo dentro da pasta `resources`, foram empacotados no JAR



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  [...]
```

Filtrando Recursos de *classpath*

- 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>}`

Filtrando Recursos de *classpath*

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}
```

Referenciando propriedades

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.

Mecanismo de Dependência

Exercício

Adicione a seguinte dependência ao projeto ‘‘my-app"

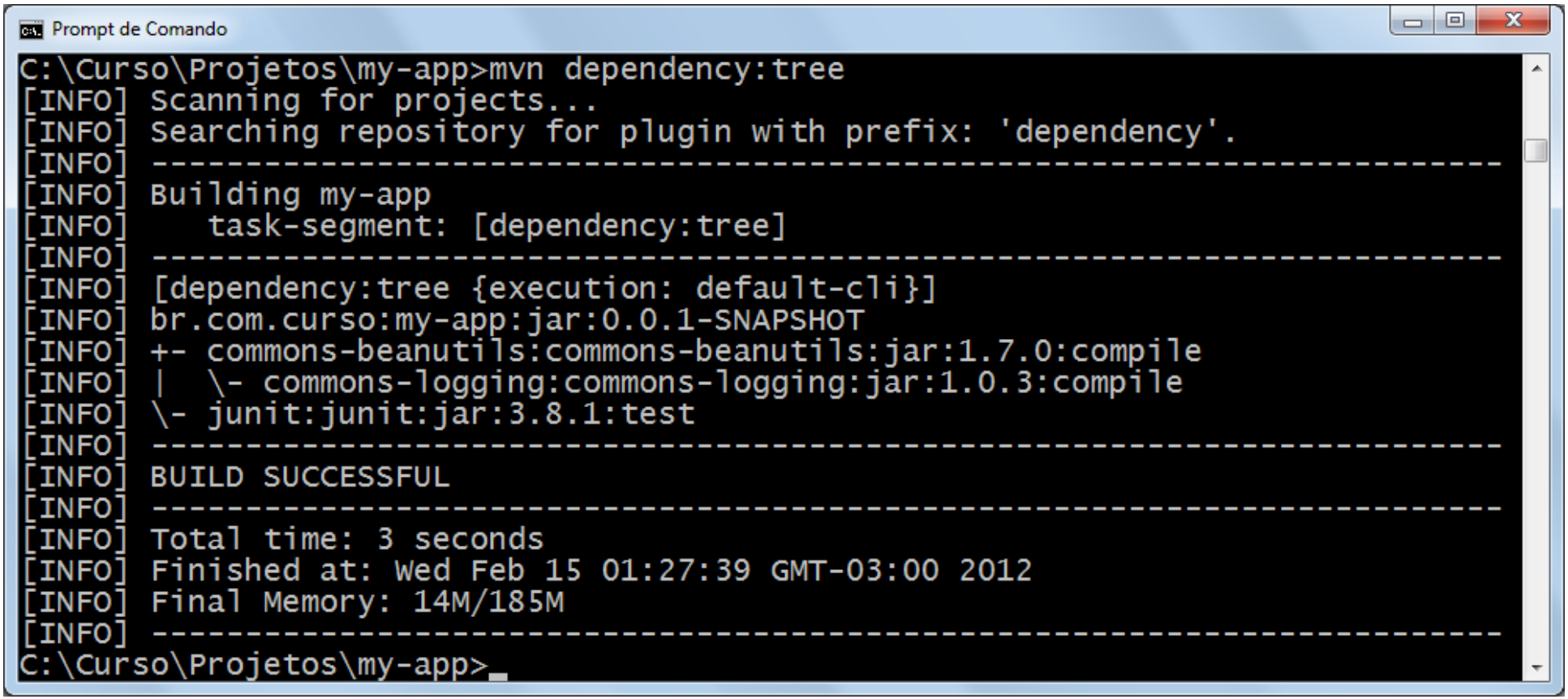
```
1 <dependency>
2         <groupId>commons-beanutils</groupId>
3         <artifactId>commons-beanutils</artifactId>
4         <version>1.7.0</version>
5 </dependency>
```

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

`mvn dependency:tree`

Mecanismo de Dependência

```
mvn dependency:tree
```



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 *classpaths* de um projeto
- Essas dependências são propagadas para os projetos dependentes.

Escopo de Dependência

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.

Escopo de Dependência

É possível instalar uma dependência no repositório local

```xml
<dependency>
        <groupId>br.teste</groupId>
        <artifactId>dependencia-teste</artifactId>
        <version>1.1.2</version>
</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

Perfils (*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

Perfils (*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

Perfils

- 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 portáveis, 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

Geração de Relatórios de Build

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

Geração de Relatórios de `Build`

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`

Fases do Maven

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

Fases do Maven

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

Fases do Maven

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

Fases do Maven

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.

Fases do Maven

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.

Fases do Maven

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>???</groupId>
8                  <artifactId>???</artifactId>
9                  <version>???</version>
10                 <configuration>
11                     ...
12                 </configuration>
13             </plugin>
14         </plugins>
15     </reporting>
16 </project>
```

Configurando os Relatórios

```
1  <!-- Relatorios -->
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>
```

Configurando os Relatórios

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**