Automatizando seus projetos com o Maven 2

Maurício Linhares de Aragão Junior

Automatize toda a gerência do ciclo de vida dos seus projetos Java utilizando o Maven 2

Introdução

O **Maven** é uma ferramenta de gerência e compreensão de projetos. Mas o que seriam a gerência e compreensão de projetos? O **Maven** gerencia projetos desde a sua criação (com a geração do esqueleto inicial do sistema) até a sua implantação em um servidor (remoto ou não).

E a compreensão do projeto? O **Maven** mantém todas as informações do projeto em um único lugar, o Project Object Model (**POM**), que é o arquivo de configuração do projeto onde são definidas todas as suas características. No POM são definidas desde informações básicas do projeto, como nome, desenvolvedores, repositórios de código fonte (sistemas de controle de versão, como CVS e Subversion), como suas dependências em bibliotecas externas e até mesmo plugins do próprio **Maven** que são utilizados para facilitar a vida dos desenvolvedores, como um servidor web embutido que executa diretamente de dentro do projeto.

Além de tudo isso, o **Maven** é uma ferramenta que prega a padronização dos projetos. Se você conhece a estrutura de um projeto básico do **Maven**, não vai ter problemas para entender outro projeto que também siga a mesma estrutura e isso diminui drasticamente o tempo que o desenvolvedor vai levar para "entrar" no novo sistema, pois ele não vai precisar entender uma nova estrutura ou aprender novos conceitos.

O **Maven** também segue a premissa da "convenção sobre configuração", onde se você segue o padrão, não é necessário dizer ao **Maven** que você está fazendo isso. Um exemplo clássico disso é a estrutura de diretórios, se você segue toda a estrutura de diretórios padrão do Maven no seu projeto, não vai precisar dizer ao plugin do compilador Javac onde ficam os seus arquivos de código fonte nem pra onde ele deve copiar os arquivos ".class" resultantes, ele já sabe exatamente onde procurar por tudo.

Ele foi desenvolvido originalmente pela equipe do projeto Jakarta Turbine com o objetivo de simplificar os "build files" do Ant utilizados no projeto. Eles estavam procurando por uma maneira de deixar todos os projetos seguindo uma mesma estrutura padronizada e também queriam não ter mais que enviar os arquivos ".jar" das dependências para os sistemas de controle de versão, foi então que surgiu a primeira versão da ferramenta, que em 2005 foi completamente reescrita, resultando no **Maven 2**, que foi construído sobre toda a experiência dos desenvolvedores e usuários da primeira versão.

Então, mãos a obra!

Antes de continuar a leitura – Instalando o Maven 2 na sua Máquina

Você deve ir até a área de downloads da página do projeto -> http://maven.apache.org/download.html

Procure a versão mais atual disponível (este tutorial utilizou a versão 2.0.4) e baixe o arquivo (os arquivos funcionam para qualquer sistema operacional), descompacte o conteúdo do arquivo em um diretório qualquer do seu computador e coloque a pasta "/bin" do diretório onde o arquivo foi descompactado no PATH do seu sistema operacional.

Você também deve ter a variável de ambiente "JAVA_HOME" definida, com o valor do diretório onde foi instalado o JDK.

No Windows XP, basta ir no "Painel de Controle", selecionar "Desempenho e Manutenção", depois "Sistema", clique na aba "Avançado", "Variáveis de Ambiente", procure a variável de ambiente com o nome PATH e **no fim** do valor atual adicione um ";" e o caminho para a pasta "bin" da instalação do Maven.

Com o serviço feito, abra o console (a linha de comando do seu sistema operacional) e digite "mvn –v", você deve receber uma mensagem parecida com a seguinte:

"Maven version: 2.0.4"

Iniciando o projeto

A primeira parte na criação de um projeto para o **Maven** é a definição do seu arquivo de configuração, o POM, é nesse arquivo que ficam todas as informações sobre o projeto que o **Maven** utiliza para fazer a sua gerência. Seguindo a estrutura de diretórios e arquivos padrão, esse arquivo tem que estar na pasta raiz do projeto e deve se chamar "pom.xml", vejamos o nosso POM inicial:

Listagem 1 – Primeira versão do "pom.xml"

```
<?xml version="1.0" encoding="UTF-8"?>
<project>
	<modelVersion>4.0.0</modelVersion>
	<groupId>br.com.guj</groupId>
	<artifactId>aprendendo-maven</artifactId>
	<packaging>war</packaging>
	<version>0.0.1</version>
	<name>Aprendendo o Maven 2</name>
	<dependencies>
		<dependency>
			<groupId>junit</groupId>
			<artifactId>junit</artifactId>
			<version>4.0</version>
			<scope>test</scope>
		</dependency>
	</dependencies>
</project>
```

O nó raiz da configuração é o <project/>, dentro dele ficam todas as informações do projeto. O primeiro nó interno, <modelVersion/>, contém a versão do arquivo de configuração do **Maven**, que no nosso caso, para o **Maven** 2, é "4.0.0", esse nó é obrigatório para que a ferramenta saiba qual a estrutura do documento.

O nó seguinte, <groupId/>, indica a qual grupo de projetos este projeto faz parte, é uma maneira de se organizar de maneira mais simples os vários projetos dentro de uma mesma empresa ou grupo de desenvolvedores. Se você tem várias aplicações, cada uma com vários módulos diferentes, poderia definir um <groupId/> diferente para cada aplicação. O nó <artifactId/> caracteriza o projeto atual, ele seria o módulo dentro de cada aplicação no exemplo anterior. Você poderia, por exemplo, ter um <groupId/> "com.bi.app" com um <artifactId/> "persistência", outro "web", outro "swing" e etc. A importância na definição dos nomes dos seus projetos vai ser abordada mais a frente na parte de repositórios do **Maven**.

O nó <packaging/> define o tipo de artefato que este projeto deve gerar, no nosso caso, definimos o artefato como um "war", que é um arquivo de aplicação web, mas poderíamos definir ele como um "jar" comum ou qualquer outro artefato que o **Maven** venha a suportar. O nó <version/> indica a versão na qual este projeto se encontra, por padrão, os desenvolvedores do **Maven** adotaram o versionamento das aplicações utilizando 3 números separados por "." (pontos), mas você pode utilizar isso da forma que achar apropriado. O nó <name/> é apenas um nome "mais bonito" para o projeto e vai ser utilizado na hora de gerar os logs e na criação da documentação do projeto.

Estes são basicamente os nós que vão ser encontrados em qualquer POM, o último nó, <dependencies/>, define uma dependência do projeto e nós vamos entender mais a frente o que isso quer dizer.

Estrutura de diretórios padrão

Como já foi dito, o **Maven** também tem como objetivo padronizar as estruturas dos projetos para que eles possam ser compreendidos mais facilmente, então ele tem uma estrutura básica de diretórios tida como padrão para os seus projetos, que é a seguinte:

- pom.xml -- Arquivo de configuração do projeto
- src/ -- pasta raiz
  - main/ -- tronco principal
    - java/ -- código fonte Java
    - resources/ -- recursos (arquivos de configuração, imagens, etc)
    - webapp/ -- aplicação web Java
  - test/ -- tronco de testes unitários e de integração
    - java/ -- código fonte dos testes do JUnit
    - resources/ -- recursos dos testes
  - site/ -- tronco principal da documentação

Seguindo esta estrutura, não é necessário dizer a ferramenta onde ficam os seus arquivos, ele sabe automaticamente onde procurar por eles. Um ponto importante desta estrutura é que ela separa os arquivos da aplicação dos arquivos dos testes da aplicação, assim, quando você gerar um "JAR" ou "WAR" do sistema, os seus testes não vão junto, porque não há necessidade de se empacotar testes unitários junto com o sistema.

Nas pastas "java" você só deve colocar aquivos ".java" (arquivos de código fonte), **qualquer** outro tipo de arquivo vai ser ignorado pelo **Maven** e seus plugins, se você precisa adicionar arquivos de configuração no classpath da aplicação (dentro do "JAR", como arquivos de configuração do Hibernate ou Log4J) eles devem estar dento da pasta "resources", pois é nela que o **Maven** procura por arquivos de configuração.

A pasta "webapp" que é vista só é necessária se o projeto em questão for de uma aplicação web, senão ela não precisa ser colocada. A pasta "webapp" contém os arquivos de uma aplicação web Java, como os JSPs, imagens, e as pastas "WEB-INF" e "META-INF", como é definido na estrutura de diretórios das aplicações web em Java. Os arquivos de configuração da aplicação web "web.xml" **não são colocados** nas pastas "resources" e sim dentro de "webapp/WEB-INF/web.xml". Lembre-se sempre que só vão para as pastas "resources" os arquivos de configuração que precisam estar no classpath da aplicação.

A última pasta definida é a pasta "site", que contém os arquivos de documentação que vão ser utilizados para gerar um "mini-site" do projeto, com informações extraídas dom POM e de outros plugins utilizados, como geradores de relatórios de análise de código.

Repositórios e dependências

Na introdução nós falamos sobre repositórios e no nosso primeiro arquivo de configuração nós vimos o nó <dependencies/> que declara as dependências do projeto, mas o que são as dependências do projeto?

As dependências de um projeto para o **Maven** são os arquivos ou bibliotecas (arquivos "JAR") que ele precisa em alguma das fases do seu ciclo de vida. No nosso POM de exemplo, declaramos uma dependência no JAR do JUnit, para que pudéssemos utilizar as classes do JUnit no projeto. Todas as dependências necessárias para o projeto devem ser definidas no POM, para que o **Maven** possa utilizá-las nas fases do ciclo de vida do projeto que elas são necessárias.

Toda a execução de ações do **Maven** depende de em qual parte do ciclo de vida o projeto está, em um projeto, existem as fases de preparação, compilação, teste, empacotamento e instalação e as dependências estão intimamente ligadas a este ciclo de vida.

Uma dependência é definida no nó <dependencies/> do POM, cada dependência fica dentro de um nó <dependency/>, que tem os seguintes nós componentes (existem outros além dos aqui definidos):

- <groupId/> - O valor do "groupId" do POM da dependência
- <artifactId/> - O valor do "artifactId" do POM da dependência
- <version/> - O valor da "version" do POM da dependência
- <scope/> - O escopo do ciclo de vida do projeto no qual esta dependência vai estar disponível

Os nós são todos auto-explicativos, pois já os vimos no POM do projeto, entretanto, existe um nó que ainda não tínhamos visto, <scope/>, que define quando uma dependência está ou não disponível para o projeto. Este nó pode assumir os seguintes valores:

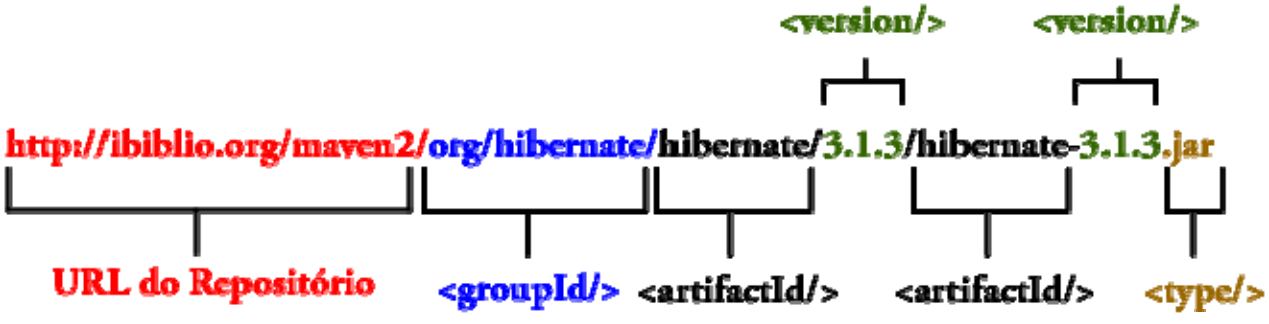
- **compile** – A dependência fica disponível durante todas as fases do projeto, desde a compilação até a instalação do sistema. Deve ser utilizada para dependências que são chamadas diretamente pelo código. Este é o escopo padrão do Maven quando nenhum escopo é definido (o nó <scope/> em uma dependência é opcional).
- **provided** – A dependência está disponível para compilação mas em tempo de execução ela deve ser disponibilizada pelo ambiente no qual a aplicação está executando. Um exemplo deste escopo é a API de Servlets, pois nós precisamos utilizar ela para compilar os nossos sistemas Java para a Web, mas não é necessário empacotar a aplicação com ela para rodar em um servidor web, o próprio servidor se encarrega de fornecer uma implementação.
- **runtime** – É o contrário de **provided**, a dependência não está disponível em tempo de compilação mas é enviada junto com o projeto em tempo de execução, normalmente é utizada para bibliotecas que são carregadas dinamicamente (como drivers JDBC) e que não precisam estar disponíveis para que o sistema seja compilado.
- **test** – A dependência só vai estar disponível para a execução dos testes do sistema e não vai ser enviada junto com a aplicação. Deve ser utilizada para bibliotecas que são utilizadas apenas para testar o sistema, como a biblioteca do JUnit no nosso exemplo, que só está disponível no escopo "test".
- **system** – Indica que a dependência não estará disponível no repositório do Maven e sua localização deve ser fornecida dentro do POM.

Escolha com cuidado o escopo no qual você quer que a sua dependência esteja, pois definir um escopo errado tanto pode fazer com que o seu projeto contenha bibliotecas inúteis (como adicionar bibliotecas de teste ao ambiente de produção) como também podem fazer com que ele simplesmente não funcione (como deixar uma dependência que deveria ser "runtime" como "provided").

A gerência de dependências é uma das partes mais importantes do **Maven** e uma das partes mais importantes desta funcionalidade é a busca automática de dependências em repositórios na sua máquina ou na internet.

Um repositório para o **Maven** é uma estrutura de diretórios e arquivos na qual ele armazena e busca por todas as dependências dos projetos que ele gerencia. Sempre que você declara uma dependência em um projeto, o **Maven** sai em busca dessa dependência no seu repositório local (normalmente fica em "sua pasta de usuário"/.m2/repository, no Windows XP seria algo como "C:\\Documents and Settings\seu_usuário\.m2\repository"), se ele não encontrar nada no repositório local, vai tentar buscar a dependência em um dos seus repositórios remotos (na internet) que vem configurados automaticamente na ferramenta (você também pode definir outros repositórios além dos padrão).

Um repositório segue uma estrutura simples de pastas baseadas nas identificações do próprio projeto, através das informações disponíveis nos nós <groupId/>, <artifactId/> e <version/>. O **Maven** define a estrutura de pastas da seguinte forma:



Para declarar esta dependência no nosso POM, ela seria definida da seguinte forma:

Listagem 2: Declarando a dependência no Hibernate

```
<dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate</artifactId>
      <version>3.1.3</version>
      <scope>test</scope>
      <type>jar</type>
</dependency>
```

A primeira parte da URL é o caminho para o repositório (que neste caso é o repositório remoto do Ibiblio, mas poderia ser o seu repositório local), a segunda parte é o valor do <groupId/> onde a cada "." encontrado, uma nova pasta é criada ("org.hibernate" torna-se "/org/hibernate/"), a terceira parte é o valor do <artifactId/> (se houverem pontos aqui eles vão ser inseridos no nome, não são criadas novas pastas como no <groupId/>).

A última parte é o próprio arquivo em si, que também tem o seu nome gerado com base nas informações do POM. Na declaração da dependência do JUnit no nosso POM anterior, não havia a tag <type/>, porque a definição dela é opcional, quando você não a define, o **Maven** assume automaticamente que a dependência é um arquivo ".JAR". O Maven também adiciona um "-" (hífen) entre o <artifactId/> e a <version/> na hora de gerar o nome do arquivo.

A capacidade de gerenciar as dependências de forma igual para todos os projetos é uma das características que o diferencia de outras ferramentas de "build" como o Ant, pois quando você utiliza o Ant, precisa definir e guardas as dependências dentro do seu projeto ou apontar diretamente para elas no seu sistema de arquivos, já com o **Maven** você define todas as dependências de uma única maneira (no POM) e a ferramenta se encarrega de fazer com que ela esteja disponível na sua aplicação.

Outro fator que diferencia o **Maven** de outras ferramentas é que ele também gerencia as "dependências das dependências" do seu projeto. Frameworks como o Hibernate utilizam várias outras bibliotecas e em outras ferramentas seria necessário definir todas estas dependências direto na sua configuração, com o **Maven**, ele utiliza a própria descrição da dependência (o seu POM) para descobrir quais são as suas dependências e as traz junto para o projeto, você não precisa mais se preocupar em saber todas as bibliotecas que precisam ser adicionadas ao classpath, o **Maven** adiciona todas elas automaticamente.

Fazendo o primeiro build com o Maven

Agora que você já conhece o básico da configuração e como o Maven gerencia as dependências e os repositórios, podemos passar ao verdadeiro trabalho de gerenciar um projeto web. Você pode utilizar os arquivos que acompanham o tutorial ou utilizar uma aplicação web sua. Utilizando a aplicação de exemplo do tutorial, abra o console (o aplicativo de linha de comando) e vá até a pasta do projeto (a que contém o arquivo "pom.xml").

Chegando a esta pasta, digite (você tem que estar conectado na internet para executar este comando):

```
mvn compiler:compile
```

"mvn" é o nome do executável que chama o **Maven** no seu ambiente, no Windows é um arquivo ".BAT", no Linux e MacOS é um script shell, sempre que você for executar qualquer comando no **Maven**, deve chamar esse executável. Se é a primeira vez que você executa o compilador do **Maven** ele vai baixar algumas dependências do seu repositório na internet (por isto você precisa estar conectado) e após baixar todas as dependências, ele vai executar o comando "compile" do plugin "compiler".

Todas as funcionalidades do **Maven** são fornecidas por plugins, que são conjuntos de código que executam ações sobre um projeto. Normalmente, cada plugin faz um trabalho específico ou vários trabalhos relacionados a uma característica específica de um projeto. O plugin que nós acabamos de executar é o plugin compilador do Maven, que pega todos os arquivos de código fonte disponibilizados em "src/main/java", compila e os envia pra pasta "target/classes" dentro da pasta do projeto. O **Maven** cria automaticamente uma pasta chamada "target" no diretório do projeto para colocar todos os artefatos que ele gera.

Os plugins que são utilizados em um projeto também precisam ser declarados no POM, vejamos então como declarar um plugin no **Maven**:

Listagem 3 – POM com o plugin compilador definido

```
<?xml version="1.0" encoding="UTF-8"?>
<project>
	<modelVersion>4.0.0</modelVersion>
	<groupId>org.maujr</groupId>
	<artifactId>aprendendo-maven2</artifactId>
	<packaging>war</packaging>
	<version>1.0.0</version>
	<inceptionYear>2006</inceptionYear>
	<name>Aprendendo o Maven 2</name>
	<description>
		Aprendendo a utilizar o Maven 2 em projetos Java
	</description>
	<dependencies>
		<dependency>
			<groupId>junit</groupId>
			<artifactId>junit</artifactId>
			<version>3.8.1</version>
			<scope>test</scope>
		</dependency>
	</dependencies>
	<build>
		<plugins>
			<plugin>
				<artifactId>maven-compiler-plugin</artifactId>
				<configuration>
					<source>1.5</source>
					<target>1.5</target>
				</configuration>
			</plugin>
		</plugins>
	</build>
</project>
```

Os plugins utilizados em um projeto são declarados dentro do nó <plugins/> que fica no nó <build/>. O nó <build/> contém informações gerais sobre o processo de build do projeto, é nele que nós poderíamos definir uma estrutura de diretórios diferente do padrão do **Maven**, mas isso não vai ser abordado neste material.

Para definir um plugin, nós criamos um nó <plugin/> dentro do nó <plugins/> e nele definimos as informações específicas do plugin, <groupId/>, <artifactId/>, <version/> e <configuration/>. O funcionamento destes nós é idêntico que já visto anteriormente, apenas <configuration/> ainda não foi visto, porque ele é específico da declaração do plugin. É neste nó que são definidas as propriedades e configurações para a execução do plugin. Cada plugin tem o eu próprio conjunto de variáveis de configuração (você pode ver a lista de variáveis disponíveis na documentação do próprio plugin).

No nosso exemplo, nós redefinimos duas variáveis de configuração do plugin, <source/> e <target/>, para indicar que nós desejamos que o compilador gere classes que utilizam e que são compatíveis apenas do Java 1.5 em diante. Para saber todas as variáveis do plugin "compiler", basta ir ao seu site: http://maven.apache.org/plugins/maven-compiler-plugin

Existem plugins para a automatização de várias tarefas dentro do **Maven**, você pode encontrar a lista oficial no próprio site da ferramenta: http://maven.apache.org/plugins/index.html

Além deste site, existem várias outras comunidades que desenvolvem plugins para o **Maven**, basta dar uma boa procurada na internet que a sua necessidade provavelmente já foi resolvida por alguma outra pessoa.

Desenvolvendo um projeto web com o Maven 2

Normalmente, para desenvolver um projeto web em Java, você precisa instalar um servidor web, configurar ele junto ao seu ambiente de desenvolvimento e começar a trabalhar com ele no seu projeto. A integração entre as ferramentas não costuma ser simples e a maioria dos servidores web disponíveis são pesados e difíceis de serem configurados. Com vistas sobre estes problemas, os desenvolvedores do levíssimo servidor web Java Jetty desenvolveram um plugin para o **Maven** onde de dentro da própria ferramenta e com um mínimo de configuração você pode ter um servidor web rodando para fazer todos os testes da sua aplicação.

Para utilizar o plugin do Jetty no seu projeto, basta apenas adicioná-lo ao seu projeto, da mesma forma que o plugin compilador foi adicionado anteriormente, vejamos como o POM deve ficar:

Listagem 4 – POM com o Jetty 6 configurado

```
<?xml version="1.0" encoding="UTF-8"?>
<project>
      … mesmo conteúdo do anterior
      <build>
            <plugins>
                  … mesmo conteúdo do anterior, só muda daqui pra frente
                  <plugin>
                        <groupId>org.mortbay.jetty</groupId>
                        <artifactId>maven-jetty-plugin</artifactId>
                        <version>6.0.1</version>
                        <configuration>
                              <contextPath>/</contextPath>
                              <connectors>
                                    <connector
                  implementation="org.mortbay.jetty.nio.SelectChannelConnector">
                                          <port>80</port>
                                          <maxIdleTime>60000</maxIdleTime>
                                    </connector>
                              </connectors>
                        </configuration>
                  </plugin>
            </plugins>
      </build>
</project>
```

Como você pode ver, a declaração do plugin é idêntica a anterior, mudando apenas as configurações específicas do plugin, vejamos o que cada nó significa:

- <contextPath/> - contexto web pelo qual a aplicação vai ser acessada, declarando o valor como "/" significa que a aplicação fica na raiz do servidor.
- <connectors/> - são os responsáveis por gerenciar as conexões do navegador, cada <connector/> pode ser de um tipo e "escutar" por requisições em uma porta diferente, a implementação mais comum é o SelectChannelConnector, pois ele é o mais rápido de todos.
    - <port/> - é a porta na qual o servidor deve esperar por conexões, a porta "80" é a porta padrão do http

Com essa configuração, já podemos executar a nossa aplicação web no servidor, vamos primeiro executar o comando para chamar o servidor (se é a primeira vez que você faz isso, deve estar conectado na internet):

```
mvn jetty:run
```

Com esse comando, nós mandamos o Maven inicializar o servidor web dentro do nosso projeto. Quando aparecer na linha de comando a mensagem "Started Jetty Server", o servidor já pode começar a receber requisições. Com base nas configurações que nós fizemos, a aplicação de exemplo vai estar disponível no endereço http://localhost/.

Seguindo a idéia de padronização das estruturas de diretório, as aplicações web gerenciadas pelo Maven 2 devem colocar os seus arquivos específicos da web dentro da pasta "src/main/webapp", é nesta

pasta que você vai montar a sua estrutura de diretórios de aplicação web em Java (com o diretório raiz, as pastas "WEB-INF" e "META-INF"). Mais uma vez, seguindo o padrão, não é necessário fazer nenhuma configuração para que os plugins entendam a sua estrutura como a estrutura de uma aplicação web.

Para parar o servidor, basta executar o comando que para a execução das aplicações na linha de comando, no Windows o comando é CTRL+C. Sempre que você atualizar as classes ou os arquivos de configuração da aplicação, o servidor precisa ser parado e inicializado outra vez, se você quer que ele descubra automaticamente que houveram mudanças, adicione o nó **<scanIntervalSeconds/>** a configuração do plugin com o valor em segundos que é a freqüência com a qual ele vai procurar atualizações na aplicação e se reiniciar sozinho. Por exemplo, se você define esta propriedade como "10" ele vai varrer toda a aplicação a cada 10 segundos procurando por atualizações, se elas forem encontradas, ele vai se reiniciar automaticamente.

Agora, além de uma ferramenta de gerência de projetos, você tem um servidor web leve e rápido para desenvolver as suas aplicações sem ter que ficar se preocupando em instalação, variáveis de ambiente e todas essas coisas, é só mandar o **Maven** executar o plugin e mãos a obra!

Fazendo a implantação (deployment) do sistema

Vimos como fazer a nossa aplicação web ser executada no servidor web do próprio **Maven**, mas nós normalmente enviamos a aplicação para ser implantada em um servidor de produção, tanto em formato WAR quanto na estrutura de diretórios padrão de aplicações web em Java.

Para gerar um WAR da aplicação, você só precisa digitar o seguinte comando no console na pasta raiz do projeto (a que contém o pom.xml):

```
mvn package
```

Se você verificar a pasta "target" do seu projeto, dentro dela vai haver uma pasta com o nome "**aprendendo-maven2-1.0.0**", que contém a estrutura completa da aplicação web, com os arquivos compilados, as dependências em bibliotecas na pasta "WEB-INF/lib" (o Maven copia automaticamente para cá qualquer dependência de escopo "runtime", "compile" ou "system"). Além desta pasta, você também vai encontrar o arquivo "**aprendendo-maven2-1.0.0.war**", que é o WAR gerado por este projeto, nele também são encontrados todos os arquivos que estavam dentro da pasta "webapp" do projeto do **Maven**, as classes e arquivos de configuração e todas as bibliotecas. O arquivo gerado está pronto para ser implantado em qualquer servidor web Java que suporte a especificação de servlets.

Como o nó <packaging/> do projeto foi definido com o valor "war", o **Maven** sabe que na fase "package" do ciclo de vida do projeto ele deve ser empacotado como um arquivo "war". O ciclo de um projeto no Maven é definido normalmente através das seguintes fases (na verdade, a seqüência é um pouco maior, veja as referências do tutorial para encontrar a lista completa):

- **validate** – valida se o projeto é válido (se a sua configuração está correta) e se o ambiente está corretamente configurado (com as devidas variáveis de ambiente configuradas)
- **compile** – compila o código fonte do projeto
- **test** – executa os testes unitários do projeto
- **package** – empacota o projeto conforme foi definido na sua descrição
- **integration-test** – executa os testes de integração (os que são feitos com a aplicação empacotada e implantada em um container)
- **verify** – verifica a qualidade da aplicação e do código (normalmente com relatórios de análize de código, resultados de testes de integração e carga)
- **install** – instala o artefato gerado no repositório local do **Maven**
- **deploy** – envia o artefato gerado para um servidor remoto para que ela seja implantada

Quando você manda o **Maven** executar qualquer uma destas fases, ele automaticamente executa todas as fases anteriores, no nosso caso, como mandamos o **Maven** executar a fase "package" ele automaticamente executou as fases "validate", "compile" e "test" antes de finalmente executar "package". Alguns plugins se associam automaticamente a fases de um projeto, para que eles sejam executados sem que o usuário precise indicar o seu uso, um caso comum é o plugin "compiler" (o compilador **Javac**) que é registrado para executar sempre na fase "compile" de um projeto, para compilar os seus arquivos de código fonte.

Olhando a lista de plugins do **Maven**, você encontra um plugin chamado de "war", este é o plugin responsável pelo empacotamento de suas classes na estrutura de uma aplicação web. Você poderia chamar o plugin diretamente, em vez de indicar a fase "package", só que o plugin "war" não executa todos os passos até a sua chamada (que seriam compilar os arquivos de código fonte e copiar os arquivos de configuração que estão em "src/main/resources"), você tem que chamar todos os plugins explicitamente, vejamos como fazer isso:

```
mvn resources:resources compiler:compile war:war
```

Neste comando, nós enviamos ao **Maven** uma instrução para que ele execute, sequencialmente, três plugins. Primeiro ele executa o plugin "resources" com a ação "resources", que copia todos os arquivos que estão em "src/main/resources" para a pasta "target/classes", depois ele executa o plugin "compiler" com a ação "compile", que faz com que ele compile todos os arquivos ".java" disponíveis em "src/main/java" e copie os arquivos resultantes para "target/classes". Por último, ele executa o plugin "war" com a ação "war" que vai criar um arquivo "war" com o conteúdo da pasta "src/main/webapp", adicionando as dependências na pasta "WEB-INF/lib" do arquivo e copiando os arquivos que estão em "target/classes" para dentro da pasta "WEB-INF/classes".

É claro que é muito mais simples e cômodo simplesmente digitar "mvn package" e deixar o **Maven** cuidar de todo o resto, mas sempre existem casos específicos que necessitam de um controle mais fino do que está acontecendo no build do projeto.

Relatórios e informações sobre o projeto

Agora que você já entende das dependências, sabe compilar os projetos, sabe executar um servidor web embutido e já entende como funcionam os plugins do **Maven**, parece que não precisa de mais nada. Mas agora nós vamos ver mais uma característica do Maven que o diferencia de outras ferramentas mais simples, a capacidade de gerar informações a partir do seu código fonte e do POM.

As informações podem ser geradas de várias formas diferentes, elas podem, inclusive, ser conteúdo estático que você gostaria de incluir na documentação do projeto (como um guia para o usuário), mas em sua maior parte a documentação gerada pelo **Maven** está intimamente relacionada a obtenção de informações sobre o "estado de saúde" do projeto, com análises de qualidade de código, conformidade com padrões de codificação e outros.

Para lidar com isso, o Maven tem um conjunto de "plugins de relatório" que analisam o projeto (normalmente o código fonte do projeto) e geram relatórios com base nestas informações. Os relatórios podem ser a simples geração de um "javadoc" e os fontes identados em arquivos HTML referenciáveis, como análises complexas de busca de padrões de bugs comuns em código ou cobertura dos testes unitários.

Vejamos o exemplo de um POM com relatórios configurados:

Listagem 5 – POM do projeto com relatórios

```
<?xml version="1.0" encoding="UTF-8"?>
<project>
            .....informações gerais do projeto
      <build>
            .....configuração dos plugins do build
      </build>
      <reporting>
            <plugins>
                  <plugin>
                        <groupId>org.apache.maven.plugins</groupId>
                        <artifactId>maven-jxr-plugin</artifactId>
                  </plugin>
                  <plugin>
                        <groupId>org.apache.maven.plugins</groupId>
                        <artifactId>maven-javadoc-plugin</artifactId>
                  </plugin>
                  <plugin>
                        <groupId>org.apache.maven.plugins</groupId>
```

```
                        <artifactId>maven-checkstyle-plugin</artifactId>
                  </plugin>
                  <plugin>
                        <groupId>org.apache.maven.plugins</groupId>
                        <artifactId>maven-pmd-plugin</artifactId>
                  </plugin>
            </plugins>
      </reporting>
</project>
```

Os relatórios são definidos no nó em um nó <plugins/> dentro do nó <reporting/>. Cada relatório funciona como um plugin comum do **Maven** e podem ser configurados da mesma forma que os outros plugins são, a única diferença é que eles são executados quando é gerada a documentação do projeto, a através do plugin "site". Para gerar a documentação do projeto, você só precisa enviar o seguinte comando:

mvn site

O **Maven** vai automaticamente gerar toda a documentação possível sobre o projeto (com base no POM) e executar todos os plugins de relatórios definidos. O site do projeto vai estar disponível dentro da pasta "target/site" do seu projeto, basta abrir o arquivo "index.html" ver o resultado gerado pelo **Maven**.

No nosso exemplo, ele vai executar o plugin "javadoc", que gera como relatório o Javadoc de todos os arquivos fontes do sistema, o plugin "jxr", que gera como relatório todo o código fonte do sistema em formatado em HTML, o plugin "checkstyle" que gera uma análise do código quanto a tua aderência a um padrão de desenvolvimento específico (quando nenhum padrão é definido, ele toma como base as regras do Java) e o "PMD", que analiza o código em busca de bugs e outros problemas comuns.

De posse destas informações, um gerente de projetos vai ter muito mais facilidade ao montar uma análise completa do estado atual e até mesmo reforçar algum critério ou padrão de desenvolvimento, com base nas ferramentas de relatório definidas. Além destes, existem vários outros plugins de relatório que geram informações sobre o projeto automaticamente também podem ser grande valia tanto para os gerentes quanto para os próprios desenvolvedores, pois eles estarão garantindo a qualidade do código e do sistema que está sendo entregue.

Dicas para problemas comuns do Maven

Não encontro a dependência em nenhum dos repositórios remoto!

As vezes o seu projeto depende de uma biblioteca ou framework que não está disponível em nenhm dos repositórios do **Maven**, as vezes porque ela não é open-source, outras vezes porque ninguém colocou ela em nenhum repositório. Um caso comum é a biblioteca de interfaces gráficas para Java Thinlet, que realmente ainda não está disponível em nenhum dos repositórios remotos.

Para adicionar ela em nosso projeto, temos que fazer com que ela esteja disponível no repositório local da máquina que está sendo utilizada. Para fazer isso, precisamos do "JAR" da biblioteca e de criar um POM com a sua declaração. Vejamos como ficaria o nosso POM:

Listagem 6 – POM da biblioteca do Thinlet

```
<?xml version="1.0" encoding="UTF-8"?>
<project>
      <modelVersion>4.0.0</modelVersion>
      <groupId>com.thinlet</groupId>
      <artifactId>thinlet</artifactId>
      <version>1.0.0</version>
</project>
```

Como você pode ver, esse POM é mínimo, contando apenas com as informações obrigatórias do projeto. A partir desta definição, nós já sabemos que o caminho para a biblioteca no repositório deve ser:

"caminho para o repositório do Maven"/com/thinlet/thinlet/1.0.0/thinlet-1.0.0.jar

Então, nós só precisamos renomear o JAR do Thinlet para "thinlet-1.0.0.jar", copiá-lo para a pasta "com/thinlet/thinlet/1.0.0/" e colocar nesta mesma pasta o POM da **Listagem 6** com o mesmo nome do arquivo JAR, mudando apenas a extensão para ".pom", ele ficaria como "thinlet-1.0.0.pom". O POM deve ter o mesmo nome do artefato porque é a partir do nome do artefato que o **Maven** sabe o nome do seu arquivo de configuração e baixa as suas dependências.

A maioria dos arquivos JAR que fazem parte do Java (como o JavaMail, o Activation, o Connector) não estão disponíveis nos repositórios do **Maven**, então você provavelmente vai ter que adicioná-los manualmente desta forma ao seu repositório local. Preste sempre atenção aos erros indicados pelo **Maven**, quando ele indicar uma dependência que não pode ser encontrada, provavelmente é o caso de adição manual no seu repositório.

Eu quero configurar o lugar do repositório local na minha máquina!

O **Maven** não exige que você faça esta configuração, mas você pode querer escolher aonde ele vai copiar os arquivos, tanto por causa de espaço ou porque você pode deixar o repositório "local" como sendo um disco compartilhado em uma rede (Windows ou Linux, contanto que ele esteja no sistema de arquivos). O maven pode ser configurado em três níveis:

- Projeto – Você pode manter a configuração específica para o projeto, diretamente no POM
- Usuário – Você pode criar uma configuração específica para o usuário atual, isso é feito definindo um arquivo "settings.xml" na pasta ".m2" que é criada na sua pasta de usuário do sistema operacional (no Windows XP ela fica em "C:\Documents And Settings\seu-login-no-windows")
- Aplicação – É a configuração definida diretamente na instalação do **Maven**, o arquivo "settings.xml" fica dentro da pasta "conf" da sua instalação do **Maven**.

No nosso exemplo, nós vamos definir uma configuração para a aplicação como um todo, escrevendo no arquivo "settings.xml" que fica dentro da pasta "conf" da instalação do Maven. Vejamos como poderia ser feita esta configuração:

Listagem 7 – Exemplo de arquivo "settings.xml"

```
<settings>
	<localRepository>\\Maven\maven-repository\</localRepository>
	<mirrors>
		<mirror>
			<id>ggi-project.org</id>
			<url>http://ftp.ggi-project.org/pub/packages/maven2</url>
			<mirrorOf>central</mirrorOf>
		</mirror>
	</mirrors>
	<profiles>
		<profile>
			<id>base-profile</id>
			<repositories>
				<repository>
					<id>local-repo</id>
					<name>Repositorio da minha empresa</name>
					<url>http://www.minhaempresa.org/maven2/</url>
					<layout>default</layout>
					<snapshotPolicy>always</snapshotPolicy>
				</repository>
			</repositories>
		</profile>
	</profiles>
	<activeProfiles>
		<activeProfile>baseProfile</activeProfile>
	</activeProfiles>
</settings>
```

O nó que define a localização do repositório é o <localRepository/>, nele nós indicamos o local no sistema de arquivos onde o repositório é encontrado. No nosso caso, indicamos uma pasta de rede

compartilhada no Windows, assim todos os desenvolvedores que tem acesso a esta pasta podem usar o mesmo repositório local, sem ter que ficar replicando os mesmos arquivos em todas as máquinas.

Após a definição do repositório local, definimos um "mirror" para o repositório "central" do **Maven**, você pode definir quantos mirrors desejar neste arquivo e eles serão selecionados conforme a seqüência na qual foram declarados. Isto é interessante se você tiver um mirror do repositório que seja geograficamente mais próximo de você, pois isso pode aumentar a velocidade da transmissão dos dados.

Após a definição dos "mirrors" nós temos a definição de "profiles", que são conjuntos de configurações organizados em grupos, para facilitar o seu gerenciamento. Em um profile você pode definir configurações como a versão do JDK que deve ser utilizado, repositórios remotos e outras coisas. Depois de definir os profiles, você tem que definir qual é o "profile" que está ativo no momento, isto é feito no nó <activeProfiles/>, que pode conter vários <activeProfile/>, cada um com o id de um profile diferente.

Se você deseja montar um repositório remoto dentro de sua empresa, pode usar um profile e definir este repositório remoto dentro do profile (como no POM de exemplo), assim vai poder deixar todas as bibliotecas que desejar no repositório para que outras pessoas possam acessar (dentro ou fora da empresa, dependendo da sua necessidade). Quando você declara um novo repositório remoto ele é **adicionado** a lista de repositórios do **Maven**, então, todos os outros repositórios declarados ainda serão chamados caso aja necessidade.

Integrando o Maven 2 com o Eclipse

O **Maven 2** tem plugins para integração com o NetBeans 5 e o Eclipse 3.1, aqui nós vamos ver como funciona o plugin para integração com o Eclipse. Antes de continuar, você deve ter o Maven 2 e o Eclipse 3.1 devidamente instalado na sua máquina.

Para instalar o plugin no Eclipse, siga os seguintes passos:

1. Selecione o menu "Help"
2. Selecione a opção "Software Updates"
3. Selecione a opção "Find and Install"
4. Na tela que aparece, escolha a opção "Search for New Features do Install" e clique no botão "Next"
5. Na tela que aperece, clique no botão "New Remote Site", digite o nome "Maven 2 Plugin" no campo "Name" e a URL http://m2eclipse.codehaus.org/ no campo "URL". Clique no botão "OK".
6. Veja se o quadro ao lado do nome "Maven 2 Eclipse" na lista está marcado, se não estiver, marque-o.
7. Clique no botão "Finish".

Deve aparecer uma tela pedindo a você confirmação sobre a instalação do plugin, confirme a instalação e o Eclipse vai começar a baixar os arquivos necessários automaticamente. Quando ele terminar, vai pedir pra reiniciar, deixe ele reiniciar e agora você já tem o plugin do **Maven** 2 instalado no seu Eclipse.

O plugin permite executar o **Maven** 2 de dentro do Eclipse e adiciona automaticamente qualquer dependência do POM no classpath do projeto do Eclipse. Para transformar um projeto comum em um projeto gerenciado pelo Maven, basta clicar com o botão direito do mouse na pasta do projeto, procurar a opção "Maven 2" e clicar na opção "Enable". Se o projeto já tiver um "pom.xml" o **Maven** simplesmente usa este POM como configuração para o projeto, se ainda não existe um POM, ele vai mostrar um formulário para que você possa indicar as informações iniciais do projeto.

O plugin também permite que você adicione dependências sem ter que alterar diretamente o arquivo de configuração. Basta clicar com o botão direito do mouse na pasta do projeto, ir na opção "Maven" e selecionar a opção "Add Dependency", um formulário onde você pode digitar o nome da dependência vai aparecer e você vai poder fazer uma busca por ela nos repositórios configurados no seu **Maven**. A parte de busca do plugin ainda tem alguns problemas, então mesmo que você não encontre a dependência, ela pode estar disponível no plugin, mas provavelmente não pode ser indexada por algum motivo, então em alguns momentos ainda pode ser necessário alterar diretamente o POM para adicionar uma dependência ao seu projeto.

Adicionar o plugin do Maven ao seu Eclipse vai facilitar ainda mais a gerência do seu classpath, pois não vai mais ser preciso ficar colocando todos os JAR que ele depende dentro do próprio projeto, pois o classpath vai ser gerenciado pelo Maven e a configuração do POM do projeto.

Referências

Sites

Página oficial do Maven: http://maven.apache.org/

Ciclo de vida dos projetos do Maven: http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html

Página oficial do Jetty: http://jetty.mortbay.org/

Lista de plugins do Maven 2: http://maven.apache.org/plugins/index.html

Página oficial do PMD: http://pmd.sourceforge.net/

Página oficial do Checkstyle: http://checkstyle.sourceforge.net/

Plugin do Maven 2 para o Eclipse: http://m2eclipse.codehaus.org/

Página do Eclipse: http://www.eclipse.org/

Livros

Maven: A Developer's Notebook. Massol, Vincent. Editora O'Reilly, 2005.

Better Builds with Maven. Casey, John; Massol, Vincent; Porter, Brett; Sanches, Carlos; Van Zyl, Jason. Mergere Library Press, 2006.

Conclusão

Como você já percebeu, o **Maven** é uma poderosa ferramenta tanto na automatização dos projetos como também na gerência de qualidade e visibilidade dos projetos. A partir da definição de um padrão de desenvolvimento, com estruturas comuns para todos os projetos, o Maven diminui os problemas relacionados a o acesso de novos desenvolvedores a projetos que já estejam caminhando.

Os seus vários plugins e integração com ferramentas como o Eclipse e o NetBeans fazem do Maven uma ferramenta ainda mais incrível e indispensável pra os desenvolvedores Java atualmente, pois não é necessário deixar de lado o que já foi aprendido nem os investimentos feitos nestas ferramentas. O Maven tem como objetivo principal padronizar e integrar o ambiente de desenvolvimento para que os desenvolvedores se preocupem com o desenvolvimento dos sistemas, não na criação de uma infra-estrutura que os permita automatizar os processos de desenvolvimento.

O Maven já provê toda a infra estrutura necessária para o desenvolvimento, livrando os desenvolvedores e os gerentes das dores de cabeça comuns na criação e padronização de processos de build em sistemas.

Sobre o autor

Maurício Linhares de Aragão Junior (*mauricio.linhares@gmail.com* – *http://maujr.org/*) é graduando em Desenvolvimento de Software para Internet no CEFET-PB e Comunicação Social (habilitação Jornalismo) na UFPB. É membro da administração do Grupo de Usuários Java da Paraíba, moderador dos fóruns do Grupo de Usuários Java, instrutor e consultor independente e desenvolvedor da Phoebus Tecnologia (http://www.phoebus.com.br/).