Release Management Tooling

Maven Renewed for a new release world

John Engelke
Senior Data Systems Software Development Specialist
March 7, 2020
Partners

Raytheon
Maven Renewed?
(Why renew something that works?)
Why renew something that works?

- Practical solution to make Maven fast!
- Modernize legacy installations
- Relatively quick and easy implementation!
- Bring Maven into the 2020s
- *Runway Transparency* – It’s all clear now!
Why renew something that works?

Runway Transparency

- SREs can say:
  - “This issue came from this exact commit (hash)!”
  - “This is from a developer’s desktop build!”
  - “This is build <x> passed by QA on date <y>!”
- CI/CD system is recognized in release tags
- Every stage of release is demarcated by unique, critical identifiers: 
  version, commit hash and build number
Methods
(Brief Review of CI/CD Concepts)
Brief Review of CI/CD Concepts

• Continuous Integration (CI) / Continuous Delivery (CD)
• Decoupling (at all stages)
• Fail Fast, Fail Often (with Rapid Notification)
• Traceability
Brief Review of CI/CD Concepts
Continuous Integration (CI) / Continuous Delivery (CD)

• Validate Software Works when Combined
  • Developers work in silos, software just builds + works
• Code Freeze
  • Bundle software into reusable packages
• Automate Whenever Possible
  • Validate functional requirements w/ Test Automation
• Every Build is a Potential Release
  • “The Eternal Beta Strategy” (some companies live it!)
**Brief Review of CI/CD Concepts**

Decoupling (at all stages)

- Components Work Independently
  - Software AND Release Management!
- A Given System is Not Required for Process Completion
- Minimize:
  - Interdependency
  - Coordination
  - Information Flow
Brief Review of CI/CD Concepts

Fail Fast, Fail Often (with Rapid Notification)

• Know the Domain
  • Stop failing processes before errors manifest
• Iterations Help Fix Errors
  • Discovering errors early isolates the cause
  • Repairing errors early prevents costly propagation
• Constant Feedback from CI/CD Systems
  • Software engineering is looped into release process
Brief Review of CI/CD Concepts

Traceability

- noun, “The quality of having an origin or course of development that may be found or followed.”
- Semantic Versioning
  - SemVer.org (MAJOR.MINOR.PATCH)
- Stepwise processes contribute coordinates
- Output can be precisely correlated with inputs
- Mechanisms: **Counters, Hashes, Coordinates**
Release Management Stack
(How Release Management Software Helps!)
How Release Management Software Helps!

- Repeatable Process for Build/Release Automation
  - Shared semantics and commands with development
- Predictable Outcomes
  - Automate routine processes, like cleaning or packaging
- Versioning Automation
- Artifact Management and Deployment
- Value Added Benefits
How Release Management Software Helps!

Components of a Release Management Stack
Maven

(Brief Review of Maven Paradigm)
Brief Review of Maven Paradigm

• Project Object Model (POM)
• Standard Directory Layout
• Build Lifecycle
• Dependency Management
  • Project AND Plugins
• Extensible
  • Release Management (Maven-Release-Plugin)
Brief Review of Maven Paradigm

Apache Maven is 15 years old, yet remains one of the most widely used build management systems with 2.3 million dependencies published (and used routinely) in Maven Central\(^1\).

Release Management Tooling: Maven Renewed

Brief Review of Maven Paradigm

Project Object Model (POM)

The Basics

The POM contains all necessary information about a project, as well as configurations of plugins to be used during the build process. It is the declarative manifestation of the "who", "what", and "where", while the build lifecycle is the "when" and "how". That is not to say that the POM cannot affect the flow of the lifecycle - it can. For example, by configuring the `maven-antrun-plugin`, one can embed Apache Ant tasks inside of the POM. It is ultimately a declaration, however. Whereas a `build.xml` tells Ant precisely what to do when it is run (procedural), a POM states its configuration (declarative). If some external force causes the lifecycle to skip the Ant plugin execution, it does not stop the plugins that are executed from doing their magic. This is unlike a `build.xml` file, where tasks are almost always dependant on the lines executed before it.

```
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.  
7.  <groupId>org.codehaus.mojo</groupId>
8.  <artifactId>my-project</artifactId>
9.  <version>1.0</version>
10.  </project>
```
# Brief Review of Maven Paradigm

## Standard Directory Layout

<table>
<thead>
<tr>
<th>Directory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>src/main/java</td>
<td>Application/Library sources</td>
</tr>
<tr>
<td>src/main/resources</td>
<td>Application/Library resources</td>
</tr>
<tr>
<td>src/main/filters</td>
<td>Resource filter files</td>
</tr>
<tr>
<td>src/main/webapp</td>
<td>Web application sources</td>
</tr>
<tr>
<td>src/test/java</td>
<td>Test sources</td>
</tr>
<tr>
<td>src/test/resources</td>
<td>Test resources</td>
</tr>
<tr>
<td>src/test/filters</td>
<td>Test resource filter files</td>
</tr>
<tr>
<td>src/it</td>
<td>Integration Tests (primarily for plugins)</td>
</tr>
<tr>
<td>src/assembly</td>
<td>Assembly descriptors</td>
</tr>
<tr>
<td>src/site</td>
<td>Site</td>
</tr>
<tr>
<td>LICENSE.txt</td>
<td>Project's license</td>
</tr>
<tr>
<td>NOTICE.txt</td>
<td>Notices and attributions required by libraries that the project depends on</td>
</tr>
<tr>
<td>README.txt</td>
<td>Project's readme</td>
</tr>
</tbody>
</table>
Brief Review of Maven Paradigm

Build Lifecycle

Maven Phases

Although hardly a comprehensive list, these are the most common default lifecycle phases executed.

- `validate`: validate the project is correct and all necessary information is available
- `compile`: compile the source code of the project
- `test`: test the compiled source code using a suitable unit testing framework. These tests should not require the code be packaged or deployed
- `package`: take the compiled code and package it in its distributable format, such as a JAR.
- `integration-test`: process and deploy the package if necessary into an environment where integration tests can be run
- `verify`: run any checks to verify the package is valid and meets quality criteria
- `install`: install the package into the local repository, for use as a dependency in other projects locally
- `deploy`: done in an integration or release environment, copies the final package to the remote repository for sharing with other developers and projects.

There are two other Maven lifecycles of note beyond the default list above. They are

- `clean`: cleans up artefacts created by prior builds
- `site`: generates site documentation for this project

Phases are actually mapped to underlying goals.
Brief Review of Maven Paradigm

Dependency Management
Brief Review of Maven Paradigm

Extensible

- Central Repository: Maven.org
- Dependency mechanism applies to both projects and Maven itself
- Core functionality extends via build plugins
  - packaging (maven-assembly-plugin)
  - scripting (groovy-maven-plugin)
  - software tag/push (maven-release-plugin)
Maven Renewed
(An Old Friend Reworked for a New Release World)
Reworked for a New Release World

Release Plugin

• Maven’s de-facto standard mechanism for tagging and artifact publishing
• Map Group-Artifact-Version coordinates
• Of four core plugins the Release Plugin evolved over a 15 year history, but methodology remains
• Series of synchronous steps execute to test compilation, test software, package code, modify source code and tag plus push new packages
Reworked for a New Release World
Release Plugin: The Challenge

• Synchronous steps mean repeat processing
• Source code changes (POM version insertion) require expensive pushes
• Minor changes (PATCH) versions result in complete rebuilds

• Is this valuable for modern release management systems? … NO!
Reworked for a New Release World

Release Plugin: The Challenge

- Costly recompiles violate FAIL FAST
- Multiple Recompiles over source code changes tie systems together, violating DECOUPLING
- Strict Semantic Versioning requires multiple lookups to reconnect the dots for TRACEABILITY
- CI/CD is beholden to a rigid multi-tiered process that extends release times over multiple compiles!
  - Consider the case of a 15-minute compile. …
Reworked for a New Release World

Release Plugin: Legacy Approach – OUCH!

<table>
<thead>
<tr>
<th>Task</th>
<th>New Approach</th>
<th>Release Plugin</th>
</tr>
</thead>
<tbody>
<tr>
<td>POM Changes</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Clean/Compile/Test</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Commits</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>SCM Revisions</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Maven Executions</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Reworked for a New Release World
Release Plugin: Legacy Approach – OUCH!

• `mvn --batch-mode release:prepare release:perform`
• For a 15-minute compile, THREE total clean/compile/test cycles
• A simple 15-minute software release is extended to 50-minute plus!
• Correlation to source-build coordinates requires research
Reworked for a New Release World
Introducing CI-friendly Maven

- **Goodbye** Release Plugin
  - Remove maven-release-plugin from all POMs
- Introducing Flexible Versioning Properties
  - Bug free and stable since Apache-Maven 3.6.3
  - *revision*, *sha1* and *changelist*\(^2\)
- Say Hello to the Maven Flatten Plugin
- Set tag syntax in the Maven SCM Plugin

---

Reworked for a New Release World
Introducing CI-friendly Maven: Properties

<!-- ... 
... -->
<name>JPL - Group - Subgroup - Project</name>
<description>Project POM describing shared directory structure, artifact repository coordinates, development
<organization>
  <name>NASA Jet Propulsion Laboratory</name>
  <url>https://www.jpl.nasa.gov/</url>
</organization>
<groupId>gov.nasa.jpl.subdomain.subdomain.group</groupId>
<artifactId>artifact-id</artifactId>
<version>1.0.0b${revision}${sha1}${changelist}</version>
<packaging>pom</packaging>

<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <!-- Tagging for CI -->
  <revision>0</revision>
  <sha1/>
  <changelist>-SNAPSHOT</changelist>
<!-- ...
Reworked for a New Release World
Introducing CI-friendly Maven: Flatten Plugin

```xml
<!-- Required for CI-friendly release deployment -->
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>flatten-maven-plugin</artifactId>
  <version>${version.plugin flatten}</version>
  <configuration>
    <updatePomFile>true</updatePomFile>
    <flattenMode>resolveCIFriendlyOnly</flattenMode>
  </configuration>
  <executions>
    <execution>
      <id>flatten</id>
      <phase>process-resources</phase>
      <goals>
        <goal>flatten</goal>
      </goals>
    </execution>
    <execution>
      <id>flatten.clean</id>
      <phase>clean</phase>
      <goals>
        <goal>clean</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```
Reworked for a New Release World
Introducing CI-friendly Maven: SCM Plugin

```xml
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-scm-plugin</artifactId>
  <version>${version.plugin.scm}</version>
  <configuration>
    <tag>${project.artifactId}-${project.version}</tag>
  </configuration>
</plugin>
```
Putting It All Together
(Develop, Build, Release and Push)
Develop, Build, Release and Push
Release Implementation

- Build versioning is controlled with Maven property injection using the `-Dproperty=""` CLI flag
- Keys to a development build
  - Developers build without any flags, yielding defaults
  - Revision is ‘b0’; Changelist is ‘-SNAPSHOT’; SHA1 is unset
- CI/CD builds are setup to inject CI/CD properties, such as the actual build number and SCM hash
Develop, Build, Release and Push

Release Implementation

- The release management pipeline optimizes
  - Single Maven execution
  - Single Clean/Compile/Test phase
  - Single SCM publish and artifact repo publish
- Complete traceability!
  - Retrieve the exact build from the CI/CD server
  - Tie back to the exact source code commit without additional research correlating tags/timing
- Developers control the entire Semantic Version and completely own .PATCH updates
Develop, Build, Release and Push

Release Implementation

• Developers install default builds locally
  • group:artifact:<semantic_version>b0-SNAPSHOT
  • **Example:** org.openjax:jetty:9.4.18b0-SNAPSHOT
    yields org/openjax/jetty-9.4.18b0-SNAPSHOT.jar in the local repo

• CI/CD server tags and publishes successful builds to the artifact repository (Nexus/Jfrog/etc.)
  • group:artifact:<semantic_version>b<x>-<SHA1>
  • **Example:** org.openjax:jetty:9.4.18b1170-ade7923
    yields org/openjax/jetty-9.4.18b1170-ade7923.jar in the artifact repository
Develop, Build, Release and Push

Build and Tag Commands

- Developers: Make a local '-SNAPSHOTS' build (at the specified version):
  - `mvn -U clean package`
  - # 1.0.0b0-SNAPSHOT given Semantic Version set at 1.0.0
Develop, Build, Release and Push
Build and Tag Commands

- Developers also own SNAPSHOT deployment:
  - `mvn -U clean install deploy -Drevision=35 -Dsha1=-$( git rev-parse --verify --short HEAD )`
Release Management Tooling: Maven Renewed

Develop, Build, Release and Push

Build and Tag Commands

- Release Engineers own releases and configure CI/CD servers to tie builds to a specific commit hash:
  - `mvn -U clean scm:tag install deploy -Dchangelist="" -Drevision=""${BUILD_NUMBER}"" -Dsha1=\$( git rev-parse --verify --short HEAD )``
  - `# 1.0.0b377-<hash>`
Develop, Build, Release and Push

Build and Tag Commands

• Developers: Make a local '-SNAPSHOT' build (at the specified version):
  • mvn -U clean package
  • # 1.0.0b0-SNAPSHOT given Semantic Version set at 1.0.0

Develop, Build, Release and Push

CI/CD Pipeline Implementation

- On successful build, the CI/CD server tags and pushes the built artifact to a repository
- Webhooks or build variables are used to notify subsequent pipeline processes
- On release, software is completely traceable by SREs charged with troubleshooting and remediating production issues
  - It’s all in the filename!
Develop, Build, Release and Push

Summary

• Build time is reduced by at least 67 percent
• Maven versus Gradle?
  • “The Eternal Beta Strategy” implementors find it attractive
  • Initially faster but requires a steep learning curve
  • Much less automated out of the box
• The improved build timing and workflow optimizations of CI-friendly Maven make it an attractive option to maintain into the future
References and Further Reading


Additional Information

- Questions or Comments?