# An introduction for Dev-OPS professionals

## What is Maven

.

A set of concepts, “The maven way”

- A tool that embodies them:
- Project Organization
- Build

## An enabler



# Agenda for today

-

Understand “The Maven Way”

- Learn about the maven tool

# The Basics

4

4

# History of Maven

.

First version in late 2001

. Goals:

- Define a model for a project so you could look in one place for everything that pertained to the project
- Require a standard directory structure so you didn’t have to go fishing around for libraries, sources and documentation
- Initial Wins:
- Made movement between projects much simpler
- Build works out of the box, everywhere



# Concept 1: Convention over Configuration

-

Every type of project should have a clear convention – Structure, file names, etc.

- A project should strive to follow the convention as much as possible



# Example 1: JAR

# Example 2: WAR

8

8

# Demo

.

See directory structure

. Look at pom

. Perform build

- See tests
- See target
- See execution



## Concept 2: Declarative vs. Imperative

-

Declarative programming defines WHAT, not HOW. e.g. SQL

- It’s benefits:
- Easier to maintain
- More open to optimization and

improvement



# Basic POM Structure

.

XML

. GAV
. Packaging (main convention)
. Properties
. Administrative Details
. Dependencies

. Plugins (~ additional conventions)

11

11

# A Simple POM

GAV

Packaging

“Admin”

Dependencies

12

# Maven Architecture

Maven is actually a framework for plug-ins

- Each plug-in defines how to do a specific task
- Each plug-in comes with conventions, and allows over-riding them using configuration
- Our build is defined by the plug-ins we use



13

# Dependencies

# Dependency Management

Dependencies are declarative

- Maven infers transitive dependencies
- We can:
- List the project dependencies

- See the dependency tree
- Package the dependencies in a jar/war/zip/…
- Etc.



# Artifacts

-

We call the packages we depend upon “Artifacts”

- These packages reside in “artifact repositories” and are accessible through their GAV coordinates

16

# Concept 3: Source vs. Artifact

- Source is written by humans
  - Source resides in Version Control
  - Source is found under <src>
- Artifacts are generated from source by machines

- Artifacts reside in artifact repositories

**There is no grey zone!**



# A look at dependencies

-

Try the following

>mvn dependency:list

>mvn dependency:tree



18

# Artifact coordinates (GAV)

Every maven pom has coordinates and creates an artifact

- This means that every module built by maven is potentially re-usable
- The coordinates have a strict structure and (of course) clear conventions



# Group ID

-

Similar in concept to the java “package” this is the namespace

- We are expected to use one that we own, avoiding clashes with

other artifacts

- Example: com.amdocs.oms.att



# Artifact ID

-

The identifier for the artifact itself, conceptually similar to a “jar name”

- Examples: log4j, rater, …

# Version

.

The specific version we are releasing.

- Versions may be either “Release” or “Snapshot”
- In maven convention a version is **release** unless it ends with “-SNAPSHOT”



# Concept 4: Everything is Versioned

Every artifact is always versioned

- This allows us:
- To depend on other artifacts with confidence
- To communicate clearly among ourselves



# Release Versions

A release is universally identical. **It never changes.**

- Remember: Maven caches artifacts based on this premise
- Examples:
- org.apache.commons:commons-lang3:3.3.2
- org.hibernate:hibernate-entitymanager:4.3.0.CR2
- org.springframework:spring-core:2.5.6.SEC03



# Snapshot Versions

-

A snapshot is inherently unstable. It denotes a “work in progress”.

- In the SCM (git) we will always see snapshot POMs, as we are working
- Maven updates snapshots based on checksum
- Examples:
- develeap.core:builder:9.1.3-SNAPSHOT



25

# Putting it all together

GAV allows us to define our dependencies

- Since they also have POMs – we have access to their dependencies. These are called transitive dependencies.
- Since releases are stable – build will work everywhere in the same way
- When we depend on snapshots – we are updated whenever work in progress changes

# But where does maven find these artifacts?



# Artifact Repository

- An artifact repository is a database that holds can deliver artifacts based on their GAV
  - It can be as simple as a file

system or as complex as a version management system



# Artifact Repository Hierarchy (client)

Maven performs a search:

- ~/.m2/repository
- POM defined repositories
- ~/settings.xml defined repositories
- Maven Central

# Artifact Repository Hierarchy (server)

Organizational repositories will typically have their own hierarchy, which client is un-aware of

- Optimizing access to network
- Blocking unwanted artifacts
- Hosting organizational artifacts

# Scope

When defining a dependency, you can limit it to one of 5 scopes:

- **compile**: full dependency
- **provided**: needed during compilation, not runtime (will be provided by rt env)
- **runtime**: needed during runtime, not compilation (do not allow it’s direct usage)
- **test**: needed only by test classes
- **system**: local. DO NOT USE.

# The Maven Lifecycle

## Concept 5: standard lifecycle

.

**Another manifestation of “convention over configuration”:**

- **A build always goes through the same stages in the same order, and achieves the same conceptual objectives**

- This is called a “Life Cycle”.



# The Standard Life Cycles

Maven has 3 standard lifecycles

- Clean – removes all traces of build
- Default – incrementally builds the product

- Site – incrementally builds related project documentation



# Phases

Each lifecycle is comprised of a set of linear steps called “phases”

- When we execute maven, we pass a phase name, and maven will go over ALL phases up to that moment
- You cannot skip phases
- Example:

Will go through “validate”, “compile” and “test”

```
>mvn test
```



# Goals

.

Life cycle phases are just “milestones”. They are NOT actions.

- Actions are performed by plugins. Each such action is called a “Goal”.
- Maven can also be called for a specific action

- Example:

Will compile the code in src. It will not go through the verify phase.

**>mvn compiler:compile**



36

# Binding

-

Goals are bound to phases, executing in order through the lifecycle

- As usual – plug-ins come with "default binding" (i.e.

"convention"), but you can change this (i.e. "configuration")

- Example: **compiler:compile** goal is bound by default to the **compile** phase.

  (but you guessed that already)



# Clean lifecycle

-

The simplest lifecycle:

- Pre-clean
- Clean
- Post-clean

# Concept 6: Clean Build

A build should always be possible on a clean machine

- Corollary:
- Projects are self contained
- Build is reproducible on any environment

This is perhaps the most important concept, and

maintaining it is the basis for many best practices!



# The Default Lifecycle

.

Responsible for:

- Build
- Test
- Distribution
- Is the most complex: 23 phases!
- We usually refer to the 10 “major” phases

# Default Lifecycle “major” phases

- **validate** - project correct, info available
- **intialize** - build POM, calc properties
- **generate-sources** – source generators
- **compile** - compile source code
- **test** - unit testing
- **package** - package as distributable(JAR,…)
- **integration-test** - E2E tests on

package

- **verify** - final verification
- **install** - install the package into the .m2
- **deploy** - release to artifact repository



41

# Install Phase

-

Installs into local repository (~/.m2/repository)

- Allows us to (locally) build

dependant artifacts

- Does not affect other team members

- Often the most useful phase

```
>mvn install
```



# Deploy Phase

-

Deploys into the distribution repository (the account Nexus)

- Is performed by the account CI Server (Jenkins)
- SNAPSHOT versions – Every build. Will affect everyone in the team
- Release versions – Release build. Will be ready for SWP/Next iteration

**>mvn deploy**



# Maven Plugins

-

# Maven plug-in architecture

Internally maven is simply a container:

- It loads the project object model
- It loads **plug-ins**, and “hosts” them
- It runs through a **lifecycle**, triggering **goals** based on **phases**

- In order to use maven well, we need to know and use the correct plug-ins

# Using Plug-Ins

Plug-ins are used by adding a <plugin> node to the build/plugins node in the POM

- If we want to bind them to a specific phase – we add an <executions> node

  If we want to pass them parameters (i.e. to “configure” them) – we add a <configuration> node

# Default Bindings

Most plugins will come with a default binding, for example:

- assembly:single is bound to **package**
- dockerfile-maven:push to **deploy**

- This is yet another manifestation of convention over configuration

## Pre-packaged plug-ins

-

maven-clean-plugin

- maven-resources-plugin
- maven-compiler-plugin

- maven-jar-plugin
- maven-surefire-plugin
- maven-install-plugin
- maven-deploy-plugin
- maven-site-plugin

## Common “tool” plugins

-

Dependency – list, tree, analyze, copy,...

- Helper – eval, effective
- Version - set

-

# Common lifecycle plugins

Assembly

- Shade
- Docker-file
- Surefire
- Failsafe
- Enforcer

- There are literally thousands of plugins

# Restructuring

It often makes sense to re-structure the project into smaller artifacts to obtain faster builds and better focus

- It often makes sense to create a convention hierarchy via parent poms
- Multi-artifact lifecycles may require the use of BOMs

# The Maven Way

Convention over Configuration

- Declarative over Imperative
- Differentiate Sources from Artifacts
- Uniform, Reproducible, Clean build
- Fully self contained modules
- Everything is versioned
- Releases are frozen, Snapshots are not

# The Maven Tool

.

mvn <phase>:

. clean
. package/test/verify/install/deploy
. mvn <plugin:goal>
. help:effective-pom
. dependency:list/tree/analyze
. version:set



# Why Maven

.

## Conventions lead to easy comprehension

- Transitive dependencies improve both stability and agility
- Multi-module allows efficient build for complex projects
- A rich set of pre-built plug-ins
- Standard ways to extend without complicating build



54

# Some things we did not cover...

Properties

- Inheritance & Super Pom
- Multi module, Reactor
- Bill Of Materials
- IDEs, Archtypes and dev support
- Project Release Process

- Practice, Practice, Practice...

# Today

Thumbnailer is A Java based application:

- Not a server
- Uses imageio extensions dynamically
- Imageio-extensions – a propritary library that adds tiff support to imageio

- Our mission: Support thumbnailer development!

# Workplan Highlights

.

Create Git-Jenkins-Artifactory topology

. Implement CI Jobs for both projects
. Dockerize the app, so it is easier to distrubute
. Create release jobs

- Allow app to handle tiffs, by packaging extension with it



# Questions…?



58