Cybage®
Delivering Value. Scientifically.

Apache Maven

Authored and Presented by: Cybage Software Pvt. Ltd.

Cybage®
Delivering Value. Scientifically.

Agenda

- What is Maven?
- Benefits Of Maven
- Maven Vs Ant
- Maven conventions
- Maven build lifecycle
- Maven Plugins
- Different Maven Repositories
- Other Features of Maven
- How to create a custom maven plugin

Cybage
Delivering Value. Scientifically.

Build Tool

- Build tools are used for Build Automation.
- **Build automation** is the act of scripting or automating a wide variety of tasks including:
  1. compiling source code into binary code
  2. packaging binary code
  3. running tests
  4. deployment to production systems
  5. creating documentation and/or release notes.

Some Java based Build Tools

Apache Maven

- "A software project management and comprehension tool"
- More than just a "build tool"

Primary Goal Of Maven

- Allow a developer to *comprehend* the complete state of a development effort in the *shortest* period of time possible.

Areas of Concern

- Making the build process easy
- Providing a uniform build system
- Providing quality project information
- Providing guidelines for best practices development

Cybage® Delivering Value. Scientifically.

The Maven Mindset

In Maven..

- We follow Maven's **Convention over configuration** principle
- You don't define custom ***tasks*** or ***script*** the build.
- Instead, you ***describe*** the project and ***configure*** the build.

Cybage® Delivering Value. Scientifically.

Maven Vs Ant

Maven	Ant
Declarative	Procedural
Description of project	Development of a build script per project
Invocation of defined goals (targets)	Invocation of project specific targets
Project knowledge, management and comprehension	"Just" the build process
build lifecycle, standard project layout	too complex scripts
reusable plugins, repositories	scripts are not reusable
moving fast forward	development are slowed down

Cybage
Delivering Value. Scientifically.

Maven Vs Ant

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.sonatype.mavenbook</groupId>
  <artifactId>my-project</artifactId>
  <version>1.0</version>
</project>
```

Maven's Pom.xml

Both are equivalent and perform same task

Ant's Build.xml

```
<project name="my-project" default="dist" basedir=".">
<description> simple example build file </description>
<!-- set global properties for this build -->
<property name="src" location="src/main/java"/>
<property name="build" location="target/classes"/>
<property name="dist" location="target"/>
<target name="init">
<!-- Create the time stamp -->
<tstamp/>
<!-- Create the build directory structure used by compile -->
<mkdir dir="${build}"/>
</target>
<target name="compile" depends="init" description="compile the source " >
<!-- Compile the java code from ${src} into ${build} -->
<javac srcdir="${src}" destdir="${build}"/>
</target>
<target name="dist" depends="compile" description="generate the distribution">
<!-- Create the distribution directory -->
<mkdir dir="${dist}/lib"/>
<!-- Put everything in ${build} into the MyProject-${DSTAMP}.jar file -->
<jar jarfile="${dist}/lib/MyProject-${DSTAMP}.jar" basedir="${build}"/>
</target>
<target name="clean" description="clean up" >
<!-- Delete the ${build} and ${dist} directory trees -->
<delete dir="${build}"/>
<delete dir="${dist}"/>
</target>
</project>
```

Benefits of Maven

- Maven has conventions
- Maven is declarative
- Maven has standard build lifecycle
- Large existing repository
- Dependency management
- Reusability
- Scalability
- SNAPSHOT builds
- Release management

Cybage® Delivering Value. Scientifically.

The Project Object Model (POM)

- Projects are described through the use of a POM file.
  - Project Group(groupId), Name(artifactId) and Version
  - Artifact Type (project packaging)
  - Source Code Management
  - Dependencies
  - Plugins
  - Release Management
  - Profiles (Alternate build configurations)
- Written in XML format

Cybage
Delivering Value. Scientifically.

Basic pom.xml

```
<project>
        <modelVersion>4.0.0</modelVersion>
        <groupId>org.sonatype.mavenbook</groupId>
        <artifactId>my-project</artifactId>
        <version>1.0</version>
</project>
```

Version of Pom

Project group id

Name of project

Version of Project

- Maven creates a **SuperPom** for every Maven project according to it's packaging type
- SuperPom acts as a parent and Pom acts as a child
- Pom inherits SuperPom and creates an **EffectivePom**

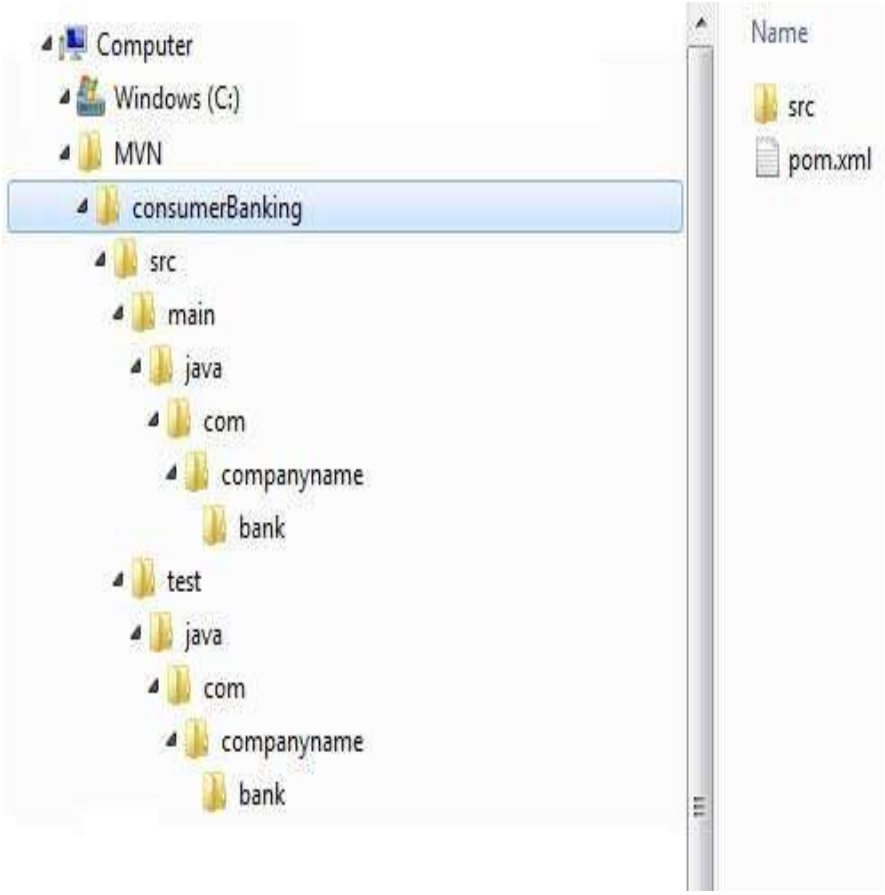
SuperPom + Pom = EffectivePom

Cybage
Delivering Value. Scientifically.

Maven Conventions

- **Standard Project Directory Layout convention**

  - src/main/java
  - src/main/resources
  - src/main/config
  - src/main/scripts
  - src/main/webapp
  - src/test/java
  - src/test/resources
  - src/site
  - …



Maven Conventions

- **Standard Artifact naming convention**

  - All projects are uniquely identified by a set of Maven Coordinates
    - Group ID
    - Artifact ID
    - Version
  - Examples (group-id:artifact-id:version):
    - junit:junit:4.11
    - org.kuali.rice:core-api:2.2.1

Build Lifecycle

- Maven 2.0 and above is based around the central concept of a build lifecycle. What this means is that the process for building and distributing a particular artifact (project) is clearly defined.
- Build lifecycles are broken down into build phases represents a stage in lifecycle
- Build phases are made up of goals

Build Lifecycle

- Only three build lifecycles
  - Clean - handles project cleaning
  - Default - handles building and deploying project
  - site - handles generation of project documentation

Build Phases for Clean Lifecycle

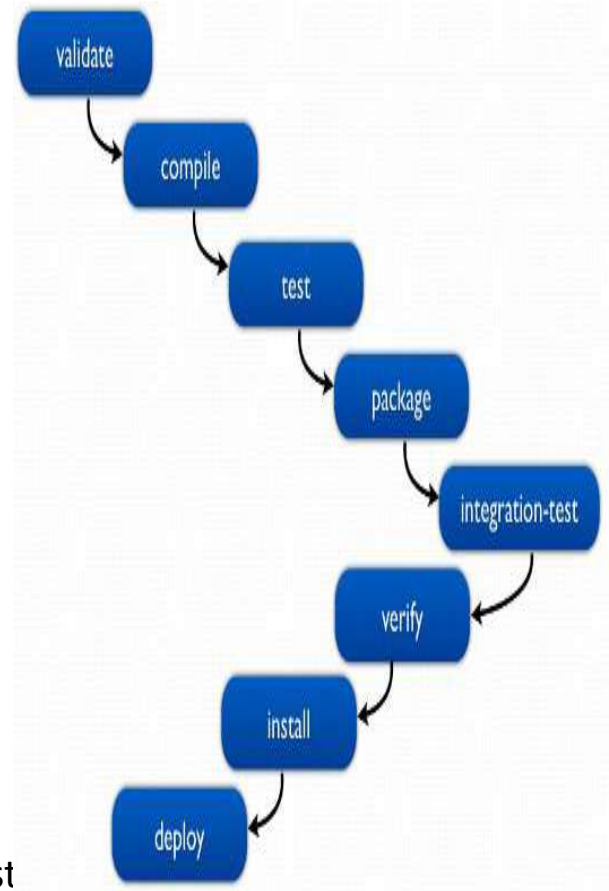
- Pre-clean - process needed prior actual project cleaning
- Clean - remove files generated in previous build
- Post-clean - process needed to finalize project cleaning

Build Phases for Default Lifecycle

- 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
- Package - take the compiled code and package it in its distributable format
- 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

Build Phases for Default Lifecycle

- 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.



This list is not actually comprehensive, but is a list of the most important build phases. It also contains pre and post phases also. This lifecycle have total 23 phases

Build Phases for Site Lifecycle

- Pre-site - process needed prior actual project site generation
- Site - generates the project's site documentation
- Post-site - process needed to finalize site generation and prepare for site deployment
- Site-deploy - deploy generated site documentation

Cybage
Delivering Value. Scientifically.

Package-Specific Lifecycle

jar-Specific Lifecycle

Lifecycle Phase	Goal
process-resources	resources:resources
compile	compiler:compile
process-test-resources	resources:testResources
test-compile	compiler:testCompile
test	surefire:test
package	jar:jar
install	install:install
deploy	deploy:deploy

war-Specific Lifecycle

Lifecycle Phase	Goal
process-resources	resources:resources
compile	compiler:compile
process-test-resources	resources:testResources
test-compile	compiler:testCompile
test	surefire:test
package	war:war
install	install:install
deploy	deploy:deploy

Cybage® Delivering Value. Scientifically.

Package-Specific Lifecycle

MavenPlugin-Specific Lifecycle

Lifecycle Phase	Goal
generate-resources	plugin:descriptor
process-resources	resources:resources
compile	compiler:compile
process-test-resources	resources:testResources
test-compile	compiler:testCompile
test	surefire:test
package	jar:jar, plugin:addPluginArtifactMetadata
install	install:install, plugin:updateRegistry
deploy	deploy:deploy

EJB-Specific Lifecycle

Lifecycle Phase	Goal
process-resources	resources:resources
compile	compiler:compile
process-test-resources	resources:testResources
test-compile	compiler:testCompile
test	surefire:test
package	ejb:ejb
install	install:install
deploy	deploy:deploy

Cybage
Delivering Value. Scientifically.

Package-Specific Lifecycle

Pom-Specific Lifecycle

Lifecycle Phase	Goal
package	site:attach-descriptor
install	install:install
deploy	deploy:deploy

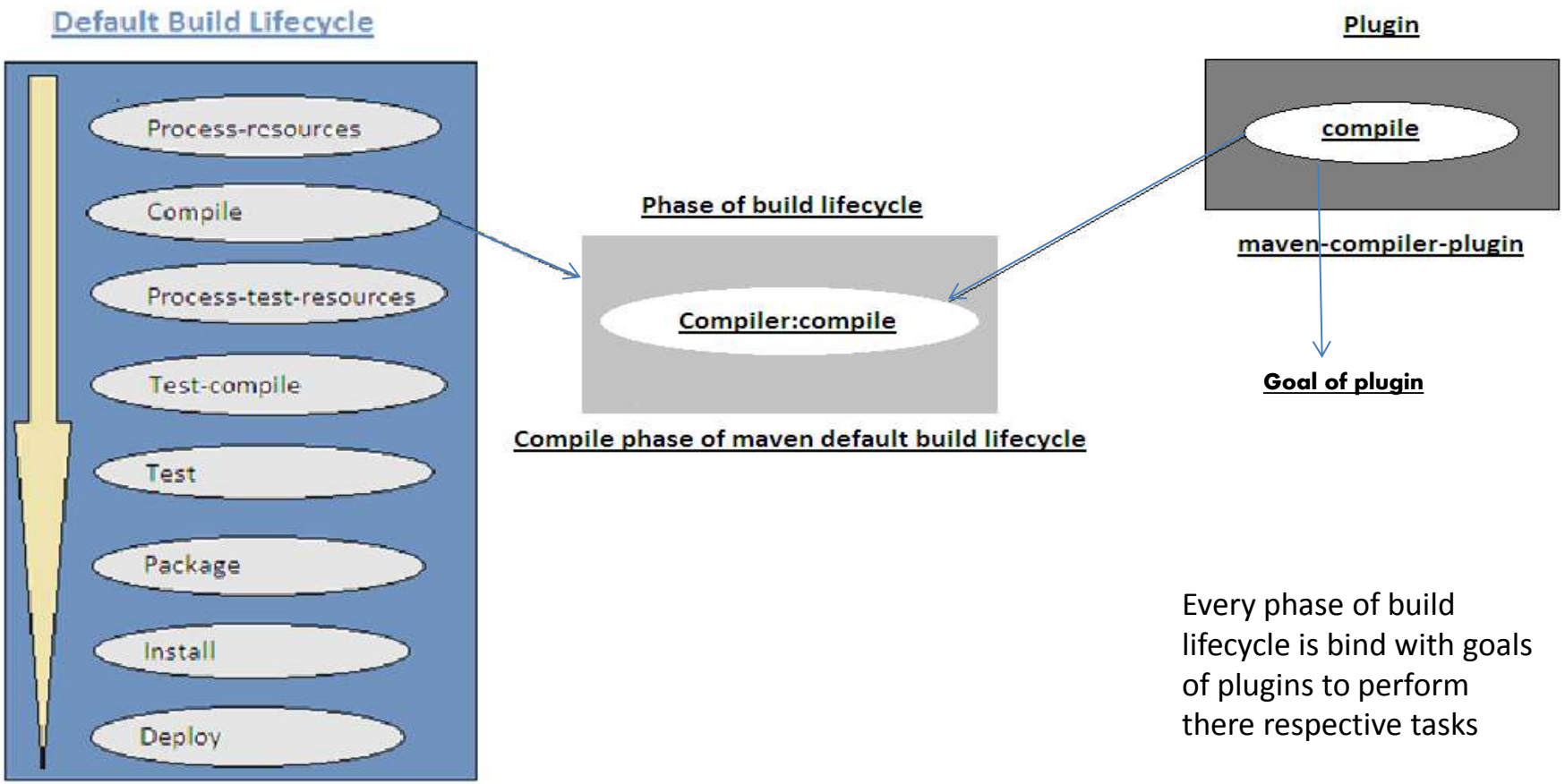
ear-Specific Lifecycle

Lifecycle Phase	Goal
generate-resources	ear:generate-application-xml
process-resources	resources:resources
package	ear:ear
install	install:install
deploy	deploy:deploy

There are a number of packaging formats available through external projects and plugins: the NAR (native archive) packaging type, the SWF and SWC packaging types for projects that produce Adobe Flash and Flex content, and many others. You can also define a custom packaging type and customize the default lifecycle goals to suit your own project packaging requirements.

Cybage® Delivering Value. Scientifically.

Relation in Phases, Plugins and Goals



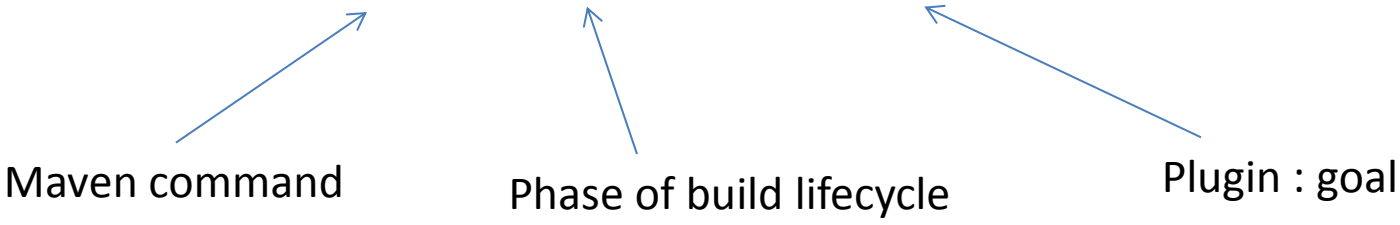
Every phase of build lifecycle is bind with goals of plugins to perform there respective tasks

Example : Build Phase and Goal

- mvn clean
- mvn test-compile
- mvn clean install
- mvn compile dependency:analyze

All phases and plugin goals get executed in the sequence in which they are specified

$ mvn compile dependency:analyze



Cybage® Delivering Value. Scientifically.

Plugins

- Plugins are artifacts that provide goals to Maven
- There is a large library of plugins out there which do all sorts of different things
- You can also write your own plugin!
- Plugins are then bound to different lifecycle phases.
  - Either through configuration in the plugin itself
  - Or through manual configuration in your POM

Cybage® Delivering Value. Scientifically.

Plugins

```
<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>com.companyname.projectgroup</groupId>
<artifactId>project</artifactId>
<version>1.0</version> <build>
<plugins>
        <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-antrun-plugin</artifactId>
        <version>1.1</version>
        <executions>
                <execution>
```

To include a plugin we need to specify these attributes

Plugins

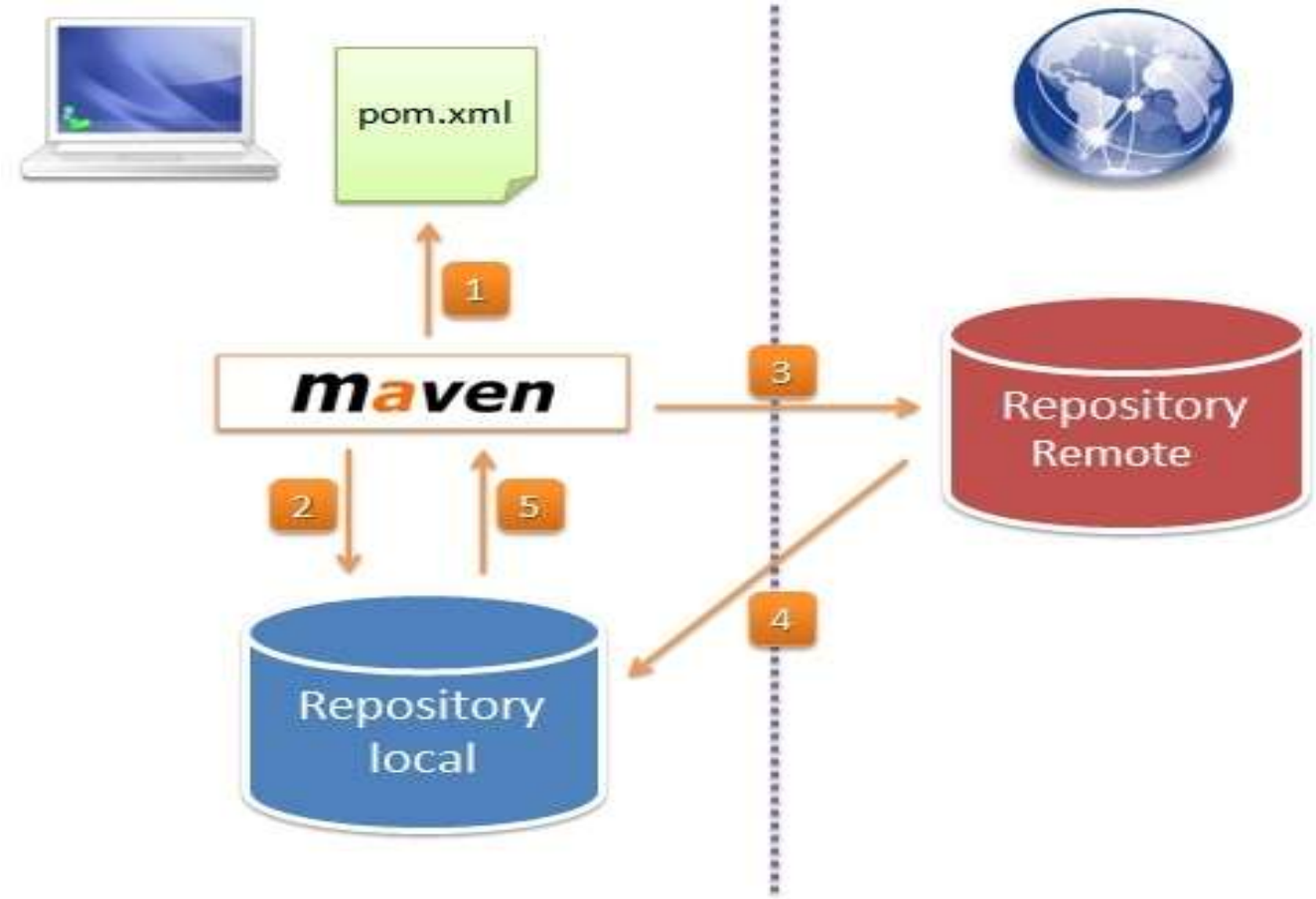
```
                        <id>id.clean</id>
                        <phase>clean</phase>
                        <goals>
                                <goal>run</goal>
                        </goals>
                        <configuration>
                                <tasks>
                                        <echo>
                                        clean phase
                                        </echo>
                                </tasks>
                        </configuration>
                </execution>
        </executions>
        </plugin>
</plugins>
</build>
</project>
```

This section is used for binding any plugins particular goal with a phase of build lifecycle. We can also bind multiple goals of a plugin with a particular phase of build lifecycle

Repositories

- Holds build artifacts of various types
- Can be local or remote
- Local repository acts as a cache
- Remote repositories include Maven Central or any number of other repositories that are available or specified by user

Repositories



Snapshot Builds

- Snapshot builds are used to designate a version of a project which has not yet been released
- The version number will end with “-SNAPSHOT”
- Stored in separate repositories or in local repository
- Are never synchronized to Maven Central
- Can be updated after deployed, unlike release artifacts which are “frozen” once released.

Cybage® Delivering Value. Scientifically.

Other Maven Features

- Profiles
- Archetypes
- Transitive dependency resolution
- Project Inheritance
- Multi-Module Project

Build Profiles

- A *Build* ***profile*** is a set of configuration values which can be used to set or override default values of Maven build.
- Using a build profile, you can customize build for different environments such as *Production* v/s *Development* environments
- It provides Portability
- Modify Pom at build time.

Cybage® Delivering Value. Scientifically.

Build Profiles

```
<project>
…..
        <profiles>
                <profile>
                        <id>development</id>
                        <properties>
                                <db.driverClass>oracle.jdbc.driver.OracleDriver</db.driverClass>
                                <db.connectionURL>
                                jdbc:oracle:thin:@127.0.0.1:1521:XE
                                </db.connectionURL>
                                <db.username>devuser</db.username>
                                <db.password>devpassword</db.password>
                                <logo.image>mylogo.png</logo.image>
                        </properties>
                </profile>
                <profile>
                        <id>production</id>
                        <properties>
                                <db.driverClass>oracle.jdbc.driver.OracleDriver</db.driverClass>
                                <db.connectionURL>
                                jdbc:oracle:thin:@10.0.1.14:1521:APPS
                                </db.connectionURL>
                                <db.username>productionuser</db.username>
                                <db.password>productionpassword</db.password>
                                <logo.image>production_logo.png</logo.image>
                        </properties>
                </profile>
        </profiles>
……
</project>
```

Profile for development environment

Profile for production environment

Cybage
Delivering Value. Scientifically.

Types of Build Profile

- **Per Project** - Defined in Project's Pom itself. Applicable only at project level.
- **Per User** - Defined in the Maven-settings (%USER_HOME%/.m2/settings.xml). Applicable on all Maven projects of a particular user
- **Global** - Defined in the Global Maven-settings (%M2_HOME%/conf/settings.xml). Applicable on all Maven projects

Cybage
Delivering Value. Scientifically.

Build Profile Activation

We can activate build profiles in following ways

- Explicitly using command prompt
- Through maven settings
- Based on Environment variable
- Based on system properties
- Based on Operating System
- Based on present and missing files
- Activate Default profile

Cybage® Delivering Value. Scientifically.

Build Profile Activation

- **Explicitly using command prompt**

```
$ mvn groupId:artifactId:goal -P profile-1,profile-2
```

- **Through Maven Settings**

```
<settings>
        ...
        <activeProfiles>
        <activeProfile>profile-1</activeProfile>
        </activeProfiles>
        ...
</settings>
```

Cybage
Delivering Value. Scientifically.

Build Profile Activation

- **Based on Environment variable**

```
<profiles>
        <profile>
                <activation>
                        <jdk>1.4</jdk>
                </activation>
                ...
        </profile>
</profiles>
```

```
<profiles>
        <profile>
                <activation>
                        <jdk> [1.3,1.6) </jdk>
                </activation>
                ...
        </profile>
</profiles>
```

You can give a range of values also

Cybage® Delivering Value. Scientifically.

Build Profile Activation

- **Based on System Properties**

```
<profiles>
        <profile>
                <activation>
                        <property>
                                <name>environment</name>
                                <value>test</value>
                        </property>
                </activation>
                ...
        </profile>
</profiles>
```

```
$ mvn groupId:artifactId:goal -Denvironment=test
```

on command prompt to activate profile on the basis of system properties

Cybage
Delivering Value. Scientifically.

Build Profile Activation

- **Based on Operating System**

```
<profiles>
        <profile>
                <activation>
                        <os>
                                <name>Windows XP</name>
                                <family>Windows</family>
                                <arch>x86</arch>
                                <version>5.1.2600</version>
                        </os>
                </activation>
                ...
        </profile>
</profiles>
```

Cybage
Delivering Value. Scientifically.

Build Profile Activation

- **Based on present and missing files**

```
<profiles>
        <profile>
                <activation>
                        <file>
                                <missing>
                                target/generated-sources/axistools/wsdl2java/org/apache/maven
                                </missing>
                        </file>
                </activation>
                ...
        </profile>
</profiles>
```

Cybage
Delivering Value. Scientifically.

Build Profile Activation

- **Activate Default Profile**

```
<profiles>
        <profile>
                <id>profile-1</id>
                <activation>
                        <activeByDefault>true</activeByDefault>
                </activation>
                ...
        </profile>
</profiles>
```

If you do not activate any profile then
this profile get activated by default

Archetypes

- Archetype is a Maven plugin whose task is to create a project structure as per its template.

```
$ mvn archetype:generate
-DgroupId=com.companyname.bank
-DartifactId=consumerBanking
-DarchetypeArtifactId=maven-archetype-quickstart
-DinteractiveMode=false
```

Archetype ArtifactIds	Description
maven-archetype-archetype	An archetype which contains a sample archetype.
maven-archetype-j2ee-simple	An archetype which contains a simplifed sample J2EE application.
maven-archetype-mojo	An archetype which contains a sample a sample Maven plugin.
maven-archetype-plugin	An archetype which contains a sample Maven plugin.
maven-archetype-plugin-site	An archetype which contains a sample Maven plugin site.
maven-archetype-portlet	An archetype which contains a sample JSR-268 Portlet.
maven-archetype-quickstart	An archetype which contains a sample Maven project.
maven-archetype-simple	An archetype which contains a simple Maven project.
maven-archetype-site	An archetype which contains a sample Maven site which demonstrates some of the supported document types like APT, XDoc, and FML and demonstrates how to i18n your site.
maven-archetype-site-simple	An archetype which contains a sample Maven site.
maven-archetype-webapp	An archetype which contains a sample Maven Webapp project.

Transitive Dependency Resolution

- Transitive Dependency Definition:
  - A dependency that should be included when declaring project itself is a dependency
- Project A depends on Project B
- If Project C depends on Project A then Project B is automatically included
- Only compile and runtime scopes are transitive
- Transitive dependencies are controlled using:
  - Exclusions
  - Optional declarations
  - Dependency management
  - Dependency mediation

Cybage
Delivering Value. Scientifically.

Dependency Exclusion

- *Exclusions exclude transitive dependencies*

```
<project>
  ...
  <dependencies>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-core</artifactId>
      <version>3.0.5.RELEASE</version>
      <exclusions>
        <exclusion>
          <groupId>commons-logging</groupId>
          <artifactId>commons-logging</artifactId>
        </exclusion>
      </exclusions>
    </dependency>
  </dependencies>
</project>
```

Optional Dependency

- Don’t propagate dependency transitively
- Optional is under used

```
<project>
  ...
  <dependencies>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-core</artifactId>
      <version>3.0.5.RELEASE</version>
      <optional>true</optional>
    </dependency>
  </dependencies>
</project>
```

Cybage
Delivering Value. Scientifically.

Dependency Management

- Directly specifies the version of artifact
- Used when no version is specified of dependency

```
<project>
        ...
    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>test</groupId>
                <artifactId>a</artifactId>
                <version>1.2</version>
            </dependency>
            ...
        </dependencies>
    </dependencyManagement>
</project>
```

Cybage
Delivering Value. Scientifically.

Dependency Mediation

- Determines version of artifact when multiple versions encountered
- Maven2.0 supports “nearest definition”
- If project A depends on project B, Project B depends on Project C and project C depends on dependency D 2.0 version Simultaneously project A depends on project E and project E depends on dependency D 1.0 version then D 1.0 is nearest definition and it get included in project A

Dependency Scope

- To limit the transitivity of a dependency
- To affect the classpath used for various build tasks.

➢ **compile**
This is the default scope. Compile dependencies are available in all classpaths of a project.

➢ **provided**
like compile, but indicates you expect the JDK or a container to provide the dependency at runtime. This scope is only available on the compilation and test classpath, and is not transitive.

➢ **runtime**
Indicates that the dependency is not required for compilation, but is for execution. It is in the runtime and test classpaths, but not the compile classpath.

Dependency Scope

- **test**
  This scope indicates that the dependency is not required for normal use of the application, and is only available for the test compilation and execution phases.

- **system**
  Similar to provided except that you have to provide the JAR which contains it explicitly. The artifact is always available and is not looked up in a repository.

- **import** *(only available in Maven 2.0.9 or later)*
  Used on a dependency of type pom in the <dependencyManagement> section. It indicates that the specified POM should be replaced with the dependencies in that POM's <dependencyManagement> section.

Cybage
Delivering Value. Scientifically.

Project Inheritance

- Pom files can inherit configuration
  - GroupId, version
  - Project config
  - Dependencies
  - Plugin configuration
  - Etc.

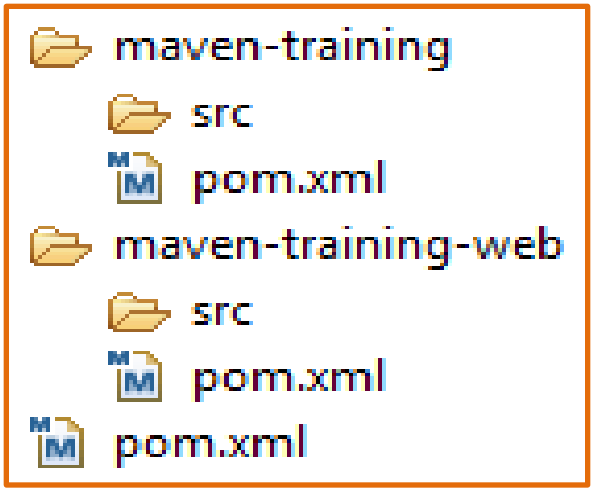
```
<?xml version="1.0" encoding="UTF-8"?>
<project>
   <parent>
      <artifactId>maven-training-parent</artifactId>
      <groupId>org.lds.training</groupId>
      <version>1.0</version>
   </parent>
   <modelVersion>4.0.0</modelVersion>
   <artifactId>maven-training</artifactId>
   <packaging>jar</packaging>
</project>
```

Cybage
Delivering Value. Scientifically.

Multi-Module Projects

- Maven has 1st class multi-module support
- Each maven project creates 1 primary artifact
- A parent pom is used to group multiple modules and build then simultaneously at the time of build of parent

```
<project>
  ...
  <packaging>pom</packaging>
  <modules>
    <module>maven-training</module>
    <module>maven-training-web</module>
  </modules>
</project>
```



Cybage® Delivering Value. Scientifically.

How to Create a Maven Plugin

- Naming convention - **<your plugin>-maven-plugin**
- Every plugin has its goals to perform its different tasks
- Goals means **MOJO [M**aven plain **O**ld **J**ava **O**bject**]**

```
$ mvn archetype:create \
  -DgroupId=org.sonatype.mavenbook.plugins \
  -DartifactId=first-maven-plugin \
  -DarchetypeGroupId=org.apache.maven.archetypes \
  -DarchetypeArtifactId=maven-archetype-mojo
```

For creating a plugin we need to create a maven-archetype-mojo project

Cybage® Delivering Value. Scientifically.

Plugin Project Definition – pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
        <project>
        <modelVersion>4.0.0</modelVersion>
        <groupId>org.sonatype.mavenbook.plugins</groupId>
        <artifactId>first-maven-plugin</artifactId>
        <version>1.0-SNAPSHOT</version>
        <packaging>maven-plugin</packaging>
        <name>first-maven-plugin Maven Mojo</name>
        <url>http://maven.apache.org</url>
                <dependencies>
                        <dependency>
                                <groupId>org.apache.maven</groupId>
                                <artifactId>maven-plugin-api</artifactId>
                                 <version>2.0</version>
                        </dependency>
                         <dependency>
                                <groupId>junit</groupId>
                                <artifactId>junit</artifactId>
                                <version>3.8.1</version>
                                <scope>test</scope>
                        </dependency>
                </dependencies>
        </project>
```

Cybage® Delivering Value. Scientifically.

MOJO

- MOJO – **M**aven plain **O**ld **J**ava **O**bject
- Java MOJO simply consist of a single class
- **AbstractMojo** - abstract superclass for the mojos to consolidate code common to all mojos
- When processing the source tree to find mojos, plugin-tools looks for classes with either **@Mojo** Java 5 annotation or "**goal**" javadoc annotation.

Cybage® Delivering Value. Scientifically.

Simple MOJO

```
package org.sonatype.mavenbook.plugins;
import org.apache.maven.plugin.AbstractMojo;
import org.apache.maven.plugin.MojoExecutionException;
import org.apache.maven.plugin.MojoFailureException;
/**
* Echos an object string to the output screen.
* @goal echo
* @requiresProject false
 */
public class EchoMojo extends AbstractMojo
{
        /**
        * Any Object to print out.
        * @parameter expression="${echo.message}" default-value="Hello
        World..."
        */
        private Object message;
        public void execute() throws MojoExecutionException,
        MojoFailureException
        {
                getLog().info( message.toString() );
        }
}
```

Cybage® Delivering Value. Scientifically.

Simple MOJO

- **org.apache.maven.plugin.AbstractMojo** provides infrastructure required to implement a mojo except for the execute() method.
- Annotation **@Mojo or @goal** is required and control how and when the mojo is executed.
- Annotation **@parameter** is required to pass attributes to the goal.
- **execute** method can throw two exceptions:

1. **org.apache.maven.plugin.MojoExecutionException** – unexpected problem. Causes BUILD ERROR
2. **org.apache.maven.plugin.MojoFailureException** – expected problem(compilation error). Causes BUILD FAILURE.

Cybage® Delivering Value. Scientifically.

Build Goals in maven-plugin

There are few goals which are defined with the **maven-plugin** packaging as part of a standard build lifecycle:

compile	Compiles the Java code for the plugin and builds the plugin descriptor
test	Runs the plugin's unit tests
package	Builds the plugin jar
install	Installs the plugin jar in the local repository
deploy	Deploys the plugin jar to the remote repository

Cybage® Delivering Value. Scientifically.

Executing MOJO(Goals) of Plugins

```
<build>
        ....
         <plugins>
                  ....
                  <plugin>
                           <groupId>org.sonatype.mavenbook.plugins</groupId>
                           <artifactId>first-maven-plugin</artifactId>
                           <version>1.0-SNAPSHOT</version>
                  </plugin>
                  ....
         </plugins>
         ....
</build>
```

Include in your project pom.xml

To execute your plugins goal

```
$ mvn org.sonatype.mavenbook.plugins:first-maven-plugin:1.0-SNAPSHOT:echo
```

Executing plugin goals without parameter

```
$ mvn org.sonatype.mavenbook.plugins:first-maven-plugin:1.0-SNAPSHOT:echo \
  -Decho.message="The Eagle has Landed"
```

Executing plugin goals with parameter

For parameterized Goals

```
/**
* Any Object to print out.
* @parameter expression="${echo.message}" default-value="Hello World..."
*/
private Object message;
```

```
<build>
        ....
        <plugins>
                ....
                <plugin>
                        <groupId>org.sonatype.mavenbook.plugins</groupId>
                        <artifactId>first-maven-plugin</artifactId>
                        <version>1.0-SNAPSHOT</version>
                        <configuration>
                                <echo.message>Welcome</echo.message>
                        </configuration>
                </plugin>
                ....
        </plugins>
        ....
</build>
```

Passing values to parameter **echo.message**

Cybage® Delivering Value. Scientifically.

Maven Best Practices

- **Make the build reproducible**
  - Always specify a version for Maven2 plugins
  - Minimize number of SNASPHOT dependencies
  - Use dependency management section
  - Beware of relocation in maven repo
  - After a dependency modification, double check the produced artifacts
- **Use and abuse of modules**
  - more “technical/layered”
  - more business oriented

Maven Best Practices

- **Make the build maintainable**
  - Prefer default directory layout
  - Avoid duplication by moving common tags to parent pom
  - Always specify a version of dependencies in a parent pom
  - Use Properties Liberally
  - Minimize the number of Profiles
- **Make the build portable**
  - Don’t commit eclipse and maven artifacts
  - Don't modify pom/artifacts in your "enterprise" repository

Cybage®
Delivering Value. Scientifically.

Any Questions?

Thank you!