# MAVEN SUCKS – NO(W) REALLY

26.01.2009

Building Projects with Maven vs. Ant

by Karl Banke

In the past few years  Maven has surpassed Ant as the build tool for choice for many projects. Its adoption by most Open Source projects creates the impression that Maven is the tool of choice for any Enterprise Java project. While Maven got some things right - constant project layout and some aspects of dependency management to name a few - it fails when confronted with basic challenges of development of distributed applications. Maven's initial promise of making simpler, cleaner build files quickly gets swamped by dummy projects for workarounds, vast configuration sections inside simple files and unnatural mappings onto Maven's build phases. This paper describes when and why Maven falls short of other build mechanism and what can be done about it - either by using an alternate process, or by fixing the Maven process.

# Maven sucks – No(w) really

**BUILDING PROJECTS WITH MAVEN VS ANT**

## Inhalt

**ABOUT THE AUTHOR** ........ 1
**A SIMPLE SCENARIO** ........ 1
**BUILDING A SIMPLE APPLICATION** ........ 2
Change of target ........ 4
**ENTERPRISE CLASS** ........ 5
Setting up dependencies ........ 5
Building with Ant ........ 5
Building with Maven ........ 7
**END-TO-END BUILDS** ........ 10
Using Ant ........ 10
Using Maven ........ 11
**REPORTING** ........ 15
**DEPENDENCY MANAGMENT AND REPOSITORIES** ........ 16
Dependency management ........ 16
Self supporting projects ........ 17
Multiple dependency trees per project ........ 17
Viral effects ........ 18
**CONCLUSION** ........ 19

## ABOUT THE AUTHOR

Karl is a senior Java architect and developer. He has been working as a consultant in the J2EE space for the last 10 years. He is one of the authors of the groundbreaking book "Enterprise SOA" published in 2004. Lately he has been involved in various organizations that have been or are migrating their build systems from Ant to Maven. He has come across many pitfalls in both migrated and greenfield projects, in particular where remote services are concerned. You can reach him at karl.banke@iternum.com.

## A SIMPLE SCENARIO

We want to find out if building a project with Maven is indeed better and easier with than building the same kind of project with Ant. To set the stage, we consider a very simple scenario. We consider a very simple application that is nevertheless dependent on some other  third party libraries. For the sake of simplicity our application consists of a method that returns back account balances. Its core method is something like the following.

```
public AccountInfo getAccountInfo(long accountNumber) {
      AccountInfo info = new AccountInfo();
      log.debug("Looking up account "+accountNumber);
      return info;
      }
}
```

In the example that we use, the only third party libary that the project depends on is the apache commons-logging library. In the real world, of course such a service will do several things that usually create dependencies on various third party libraries for building and testing the application.

| **Task** | Dependency | Type |
|---|---|---|
| **Query a database** | ORM Mapper | Compile Time |
| | Concrete Database Driver | Runtime |
| **Logging** | Logging Abstraction Layer | Compile Time |
| | Concrete Logging Implementation | Runtime |
| **…** | | |
| **Unit testing** | Unit Testing Framework | Compile Time |

**TABLE 1TYPICAL DEPENDENCIES**

## BUILDING A SIMPLE APPLICATION

Building such an application as a standalone Java application is as simple as it gets. In this simple scenario we only have three classes. Two that comprise the application – AccountInfo and AccountService – and one that is used for testing purposes – TestAccountService. For such a setup the maven pom.xml file that describes the entire build is as simple as it gets. All that needs to be declared are the project dependencies.

```
<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/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.iternum.demo</groupId>

<artifactId>SimpleAccountInfo</artifactId><packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>SimpleAccountInfo</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>commons-logging</groupId>
      <artifactId>commons-logging</artifactId>
      <version>1.1.1</version>
      <scope>compile</scope>
    </dependency>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

The corresponding Ant build.xml file is a lot longer in comparison. In fact it will usually have three times the number of lines.

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="SimpleAccountInfo" default="package" basedir=".">
  <!-- set global properties for this build -->
  <property name="version"  value="1.0-SNAPSHOT"/>
  <property name="target"   location="target"/>
  <property name="src.main" location="src/main/java"/>
  <property name="src.test" location="src/test/java"/>
  <property name="classes.main" location="${target}/classes"/>
  <property name="classes.test" location="${target}/test-classes"/>
  <property name="junit.report.dir" location="${target}/surefire-
reports"/>
  <!-- Define classpaths -->
  <path id="compile.classpath">
    <fileset dir="lib/compile" includes="*.jar"/>
  </path>

  <path id="test.classpath">
          <path refid="compile.classpath"/>
          <path location="${classes.main}"/>
    <fileset dir="lib/test" includes="*.jar"/>
  </path>

  <target name="init">
    <!-- Create the time stamp -->
    <tstamp/>
    <!-- Create the build directory structure used by compile -->
    <mkdir dir="${classes.main}"/>
    <mkdir dir="${classes.test}"/>
  </target>

  <target name="compile" depends="init" description="compile the
source" >
    <javac srcdir="${src.main}" destdir="${classes.main}"
classpathref="compile.classpath"/>
    <javac srcdir="${src.test}" destdir="${classes.test}"
classpathref="test.classpath"/>
  </target>

  <target name="test" depends="compile">
          <mkdir dir="${junit.report.dir}"/>
          <junit printsummary="yes" haltonfailure="yes">
                  <classpath>
                          <path refid="test.classpath"/>
                          <path  location="${classes.main}"/>
                          <path  location="${classes.test}"/>
                  </classpath>
                  <batchtest todir="${junit.report.dir}">
                          <fileset dir="${classes.test}">
                                  <include name="**/*Test*.class"/>
                          </fileset>
                          <formatter type="xml"/>
                          <formatter type="plain"/>
```

```
                   </batchtest>
           </junit>
   </target>

   <target name="package" depends="test" description="generate the
distribution" >
     <jar jarfile="${target}/${ant.project.name}-${version}.jar"
basedir="${classes.main}"/>
   </target>

   <target name="clean" description="clean up" >
     <delete dir="${target}"/>
   </target>
</project>
```

This should settle the score – or should it? Maven is a lot simpler to use and maintain. It creates more compact build files and on top of that enforces a clean and concise architecture on projects by its "convention over configuration" approach. Lets take a closer look at the files. The Ant File contains more than twice the number of lines than the Maven File. However, a fairly large number of the lines is due to the fact, that we need a significant amount of lines to define properties for our project. Also configuration of unit testing takes a number of lines.

Maven's "convention of configuration" approach saves us a lot of lines here. But consider what happens as we raise the number of dependencies a bit. Each dependency adds 7 more lines to the pom.xml. The build.xml however will not change at all. This is of course, because ant is not a dependency management tool at all. Our libraries are taken from two particular directories by convention - our own of course. Five more dependencies and the pom.xml will grow to the same size as the ant build file.

## Change of target

It is a very common situation that the version of the source code you are compiling and the version of the jsdk you are using to run the build tool do not match. For example if the build tool requires a certain jsdk feature from a newer version. How do we change the build files to reflect the change? It turns out, in Ant we add two properties to the javac tasks. There is also a magic number to change the compiler across the board if you need to. At the end we do something like this.

```
<target name="compile" depends="init" description="compile the
source" >
    <javac source="1.5" target="1.5" …
    <javac source="1.5" target="1.5" …
```

Maven is not build across interdependent targets like ant but around defined lifecycles and plugins that are executed in these lifecycles. It is required to reconfigure the relevant plugin, which adds another 12 lines to our script.

```
<build>
 <plugins>
   <plugin>
     <groupId>org.apache.maven.plugins</groupId>
     <artifactId>maven-compiler-plugin</artifactId>
     <configuration>
        <source>1.5</source>
        <target>1.5</target>
     </configuration>
```

```
    </plugin>
  </plugins>
</build>
```

Even with just one dependency, the difference between the maven and the ant script in length is not that big anymore. (2 : 3). Still, Maven will probably be more attractive for many, since it gives you the dependency management. But keep in mind that this is still a very simple project. Making its configuration only deviate a tiny bit from the convention takes the configuration size well within the range we know from ant.

## ENTERPRISE CLASS

Now consider to make our top-notch service an Enterprise Java Bean. Stateless Session EJB are a common technology that is used for various reasons.

• To make an application available remotely

• To decouple parts of an application effectively

• To make use of automated transaction management

Being modern people we do use EJB3. To turn our simple service into an EJB, all that is required is to extract an interface out of our application, create a home interface and to put some small annotation in front of the code. Using Ant this is pretty much all we need to do to compile our EJB classes.

```
@Remote
public interface AccountService {..
}

@Stateless
public class AccountServiceBean implements AccountService{..
}
```

### Setting up dependencies

Easy enough. Now, all we need to is to add the proper dependency for Maven and/or the library for ant. As it turns  out, there is no proper place in one of the typical default repositories where the Java EE API can be found. Also, most application server vendors won't even supply the proper standalone API but bundle it somewhere inside their custom library. So for our ANT project, we grab one of the proper APIs and throw it into the lib/compile/exclude directory.  We need to choose this slightly different directory in order to be able to easily prevent files from being packaged with the ejb.

For Maven, this leaves us with two choices: Install or deploy the required library to our local or central repository, or reference it directly from our lib directory. Directly referencing it is an option but not one very much in the Maven spirit, so we choose to install it into the local repository.

```
> mvn install:install-file -DgroupId=javaee -DartifactId=javaee -
Dversion=5 -Dpackaging=jar -Dfile=javaee.jar
```

### Building with Ant

Once this has been done, we can happily build away. Unfortunately this is only half of what we need. In order to create a proper, deployable EJB, we would need to package all dependencies together with the EJB into an EAR file. Also we need an EJB Client file that contains the proper classes only - obviously we do not want to ship the data. For packaging both the EJB as well as the EAR we require only a few additional lines in ant and end up with a deployable - and inherently platform independent - EAR file. The Ant file is about 70 lines.

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="SimpleAccountInfo" default="package" basedir=".">
  <!-- set global properties for this build -->
  <property name="version"  value="1.0-SNAPSHOT"/>
  <property name="target"   location="target"/>
  <property name="src.main" location="src/main/java"/>
<property name="resources.main" location="src/main/resources"/>
  <property name="src.test" location="src/test/java"/>
  <property name="classes.main" location="${target}/classes"/>
  <property name="classes.test" location="${target}/test-classes"/>
  <property name="junit.report.dir" location="${target}/surefire-
reports"/>

    <path id="compile.classpath">
    <fileset dir="lib/compile" includes="**/*.jar"/>
  </path>

  <path id="test.classpath">
          <path refid="compile.classpath"/>
          <path location="${classes.main}"/>
    <fileset dir="lib/test" includes="*.jar"/>
  </path>

  <target name="init">
    <!-- Create the time stamp -->
    <tstamp/>
    <!-- Create the build directory structure used by compile -->
    <mkdir dir="${classes.main}"/>
    <mkdir dir="${classes.test}"/>
  </target>

  <target name="compile" depends="init" description="compile the
source" >
    <javac source="1.5" target="1.5" srcdir="${src.main}"
destdir="${classes.main}" classpathref="compile.classpath"/>
    <javac source="1.5" target="1.5" srcdir="${src.test}"
destdir="${classes.test}" classpathref="test.classpath"/>
  </target>

  <target name="unittest" depends="compile">
          <mkdir dir="${junit.report.dir}"/>
          <junit printsummary="yes" haltonfailure="yes">
                  <classpath>
                          <path refid="test.classpath"/>
                          <path  location="${classes.main}"/>
                          <path  location="${classes.test}"/>
                  </classpath>
                  <batchtest todir="${junit.report.dir}">
                          <fileset dir="${classes.test}">
                                  <include name="**/*Test*.class"/>
                                  <exclude name="**/remote/**"/>
                          </fileset>
```

```
                              <formatter type="xml"/>
                              <formatter type="plain"/>
                      </batchtest>
              </junit>
  </target>

  <target name="package" depends="unittest" description="generate
the distribution" >
    <jar jarfile="${target}/${ant.project.name}-${version}.jar"
basedir="${classes.main}"/>
          <jar jarfile="${target}/${ant.project.name}-client-
${version}.jar" basedir="${classes.main}"
excludes="**/impl,**/impl/*"/>
          <copy file="${target}/${ant.project.name}-${version}.jar"
tofile="${target}/ear/${ant.project.name}.jar"/>
          <copy todir="${target}/ear/lib">
            <fileset dir="lib/compile" includes="*.jar"/>
          </copy>
          <ear destfile="${target}/${ant.project.name}-
${version}.ear" basedir="${target}/ear"
appxml="${resources.main}/META-INF/application.xml"/>
  </target>

  <target name="clean" description="clean up" >
    <delete dir="${target}"/>
  </target>
</project>
```

## Building with Maven

With maven, the only suitable way of building the ear is to create two maven projects - one that creates the actual EJB and one that packages the EAR file.

First we create the project that builds the EJB. The EJB Plugin is briefly described in http://maven.apache.org/plugins/maven-ejb-plugin/usage.html. Because we have an EJB 3.0 Bean, and because we want to create the EJB client, we need just one more configuration in our build file. This creates a server and a client file with the client missing some files that match a predefined exclude path. In most real world cases, the predefined exclude path will not suffice but in our case everything works out because Maven is so kind to exclude all classes that match the *Bean Pattern.

```
<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/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.iternum.demo</groupId>
  <artifactId>SimpleAccountInfoEJB</artifactId>
  <packaging>ejb</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>SimpleAccountInfoEJB</name>
  <url>http://maven.apache.org</url>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
```

```
          <source>1.5</source>
          <target>1.5</target>
        </configuration>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-ejb-plugin</artifactId>
        <configuration>
          <ejbVersion>3.0</ejbVersion>
          <generateClient>true</generateClient>
        </configuration>
      </plugin>
    </plugins>
  </build>
  <reporting>
      <plugins>

      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-report-plugin</artifactId>
        <version>2.4.3</version>
      </plugin>

      </plugins>
    </reporting>

  <dependencies>
    <dependency>
      <groupId>commons-logging</groupId>
      <artifactId>commons-logging</artifactId>
      <version>1.1.1</version>
      <scope>compile</scope>
    </dependency>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>javaee</groupId>
      <artifactId>javaee</artifactId>
      <version>5</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>
</project>
```

In order to create the EAR file, we will need to first build the EJB JAR file as described above. The resulting artifact then needs to be installed and deployed and can be picked up by another build file that creates an EAR from the previously created ejb JAR.

```
<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/maven-v4_0_0.xsd">
```

```
   <modelVersion>4.0.0</modelVersion>
   <groupId>com.iternum.demo</groupId>
   <artifactId>SimpleAccountInfoEAR</artifactId>
   <packaging>ear</packaging>
   <version>1.0-SNAPSHOT</version>
   <name>SimpleAccountInfoEAR</name>
   <url>http://maven.apache.org</url>
   <build>
  <plugins>
       <plugin>
         <groupId>org.apache.maven.plugins</groupId>
         <artifactId>maven-ear-plugin</artifactId>
         <configuration>
       <version>5</version>
       <defaultLibBundleDir>lib</defaultLibBundleDir>
       <modules>
         <ejbModule>
           <groupId>com.iternum.demo</groupId>
           <artifactId>SimpleAccountInfoEJB</artifactId>
         </ejbModule>
       </modules>
         </configuration>
       </plugin>
  </plugins>
   </build>
   <dependencies>
  <dependency>
    <groupId>com.iternum.demo</groupId>
       <artifactId>SimpleAccountInfoEJB</artifactId>
       <version>1.0-SNAPSHOT</version>
    <type>ejb</type>
  </dependency>
   </dependencies>
</project>
```

Now we can first build the first project, than build the second project and end up with an EAR like exactly like the one we created with ANT. The two files required for this add up to 90 lines of code. The corresponding ANT file is just a mere 68 lines long, which is significantly smaller. Because you might want to build the whole thing in one go these two projects can be tied together using a so called multi module project. All this does is to provide a common platform to run various Maven builds one after the other. This requires the definition of a project that aggregates our existing projects.  This adds another 16 lines to our total maven configuration. With now 106 lines of Maven configuration we have almost 40 lines more than in the appropriate ant configuration.

```
<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/maven-v4_0_0.xsd">
   <modelVersion>4.0.0</modelVersion>
   <groupId>com.iternum.demo</groupId>
   <artifactId>SimpleAccountInfoParent</artifactId>
   <packaging>pom</packaging>
   <version>1.0-SNAPSHOT</version>
   <name>SimpleAccountInfoParent</name>
   <url>http://maven.apache.org</url>
   <modules>
     <module>SimpleRemoteAccountServiceMavenEJB</module>
     <module>SimpleRemoteAccountServiceMavenIntegration</module>
   </modules>
</project>
```

## END-TO-END BUILDS

To create a real application build it is often necessary to test the application running within its actual container. The reason is that certain application characteristics like transaction like transaction semantics, serialization or security management can not be reliably tested without the actual execution environment. Thus we now need to deploy out application into the EJB container and run integration tests against the deployed EJB.

### Using Ant

Fortunately, almost all application servers come with some ant support for deployment of applications. They usually requires copying some libraries into ant/lib and defining the respective targets.

```

<taskdef name="sun-appserv-deploy" classpathref="as.classpath"

classname="org.apache.tools.ant.taskdefs.optional.sun.appserv.Deploy
Task"/>

<target name="deploy" depends="package">
  <sun-appserv-deploy asinstalldir="C:\Program Files\Sun\glassfish\"
    name="${ant.project.name}" file="${target}/${ant.project.name}-
${version}.ear"
    upload="true" force="true" passwordfile="password.txt"/>
</target>
```

Finally, all we need for a full lifecycle build, is to test certain aspects of the application remotely. This requires a repetition of the rather verbose test definition. With about 100 lines of XML we have arrived at a complete lifecycle build. Some more lines would probably be appropriate to generate a more human readable test report.

```
<target name="integrationtest" depends="deploy">
    <mkdir dir="${junit.report.dir}"/>
    <junit printsummary="yes" haltonfailure="yes">
      <classpath>
        <path refid="test.classpath"/>
        <path  location="${classes.test}"/>
        <path  location="${target}/${ant.project.name}-client-
${version}.jar"/>
      </classpath>
      <batchtest todir="${junit.report.dir}">
        <fileset dir="${classes.test}">
          <include name="**/remote/**/*Test*.class"/>
        </fileset>
        <formatter type="xml"/>
        <formatter type="plain"/>
      </batchtest>
    </junit>
```

```
</target>
```

## Using Maven

How does one deploy an ant file and run tests in Maven. It turns out that this is a lot harder than one would expect. As with ant, one is basically dependent for a plugin being available that deploys to a particular application server. Unfortunately, there is no such thing for most application servers. Generally the Maven Build life cycle defines an integration-test-phase that could be run right after packaging the EAR file. However there is no real meaningful default implementation available. There are some open source efforts, like Cargo (http://cargo.codehaus.org/Home) but most of the time you will miss one important feature or support of your respective version is not yet available and so on.

In order to just run the remote Unit Tests, we need to create a pom.xml file that resolves the dependencies to the required server runtime files, in our case to the glassfish libraries. We choose to deploy them into our repository as dependencies, so we can address them properly in Maven. Defining these tests alone, adds another 54 lines to our Maven deployment. Altogether that is 160 lines of configuration in Maven - and this does not yet include deployment to the server.

```
<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/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.iternum.demo</groupId>
  <artifactId>SimpleAccountInfoIntegration</artifactId>
  <version>1.0-SNAPSHOT</version>
  <name>SimpleAccountInfoIntegration</name>
  <url>http://maven.apache.org</url>
  <build>
 <plugins>
 </plugins>
  </build>
  <dependencies>
   <dependency>
     <groupId>com.iternum.demo</groupId>
      <artifactId>SimpleAccountInfoEJB</artifactId>
      <version>1.0-SNAPSHOT</version>
     <type>ejb-client</type>
   </dependency>
   <dependency>
       <groupId>junit</groupId>
       <artifactId>junit</artifactId>
       <version>3.8.1</version>
       <scope>test</scope>
     </dependency>
     <dependency>
       <groupId>javaee</groupId>
       <artifactId>javaee</artifactId>
       <version>5</version>
       <scope>provided</scope>
     </dependency>
     <dependency>
       <groupId>com.sun.appserv</groupId>
       <artifactId>appserv-deployment-client</artifactId>
       <version>1</version>
       <scope>test</scope>
```

```
    </dependency>
    <dependency>
      <groupId>com.sun.appserv</groupId>
      <artifactId>appserv-ext</artifactId>
      <version>1</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>com.sun.appserv</groupId>
      <artifactId>appserv-rt</artifactId>
      <version>1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

Now we need to make sure that the generated EAR is deployed to the server prior to running the tests. One of the most common approaches you come across in a lot in forums and some software application wizards is to create a separate project for deployment to the server. The generated EAR can be deployed to the Server during the compile phase and the tests can then be performed during the normal test phase. Generally this may not the cleverest of ideas. Essentially the deploy phase (the uploading of the files to the Maven repository) of the actual targets - the EAR File as well as the Client Jar File, will run well before the integration tests on the EAR can be performed, which strikes as a bit odd.

The size of the configuration for this file is mainly dependent on the number of required dependencies and on the tools available to perform the deployments. A straightforward solution for glassfish uses the antrun plugin for deployment and defines all libraries required by the plugin as dependencies. Also a plugin to copy the target file is needed prior to deployment. Altogether a massive 157 lines to do something fairly trivial. Even if we are solely restricted to the file copying and the ant plugin, we end up with something like 80 lines. With Maven, we end up with about 240 to 320 lines of build files altogether as compared with ant's 100 lines.

```

<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/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.iternum.demo</groupId>
  <artifactId>SimpleAccountInfoIntegration</artifactId>
  <version>1.0-SNAPSHOT</version>
  <name>SimpleAccountInfoIntegration</name>
  <url>http://maven.apache.org</url>

  <build>
    <plugins>
   <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-dependency-plugin</artifactId>
        <executions>
          <execution>
            <id>copy</id>
            <goals>
              <goal>copy</goal>
            </goals>
            <configuration>
        <artifactItems>
                <artifactItem>
                  <groupId>com.iternum.demo</groupId>
                  <artifactId>SimpleAccountInfoEAR</artifactId>
                  <type>ear</type>
                  <overWrite>true</overWrite>
                </artifactItem>
              </artifactItems>
        <stripVersion>true</stripVersion>

<outputDirectory>${project.build.directory}</outputDirectory>
              <includeTypes>ear</includeTypes>
            </configuration>
          </execution>
        </executions>
       </plugin>
       <plugin>
         <groupId>org.apache.maven.plugins</groupId>
         <artifactId>maven-antrun-plugin</artifactId>
         <executions>
           <execution>
             <id>compile</id>
             <phase>compile</phase>
             <configuration>
               <tasks>
                 <taskdef name="sun-appserv-deploy"
classname="org.apache.tools.ant.taskdefs.optional.sun.appserv.Deploy
Task"
                    classpathref="maven.compile.classpath"/>
                  <sun-appserv-deploy asinstalldir="C:\Program
Files\Sun\glassfish\"

file="${project.build.directory}/SimpleAccountInfoEAR.ear"
                   upload="true" force="true"
passwordfile="password.txt"
                  />
               </tasks>
             </configuration>
             <goals>
               <goal>run</goal>
             </goals>
           </execution>
         </executions>
         <dependencies>
           <dependency>
             <groupId>commons-net</groupId>
             <artifactId>commons-net</artifactId>
             <version>1.4.1</version>
           </dependency>
           <dependency>
             <groupId>ant</groupId>
             <artifactId>ant-commons-net</artifactId>
             <version>1.6.5</version>
           </dependency>
           <dependency>
             <groupId>ant</groupId>
             <artifactId>ant-nodeps</artifactId>
             <version>1.6.5</version>
           </dependency>
         </dependencies>
```

```
      </plugin>
    </plugins>
  </build>

  <dependencies>
     <dependency>
    <groupId>com.iternum.demo</groupId>
    <artifactId>SimpleAccountInfoEAR</artifactId>
       <version>1.0-SNAPSHOT</version>
    <type>ear</type>
  </dependency>
  <dependency>
    <groupId>com.iternum.demo</groupId>
    <artifactId>SimpleAccountInfoEJB</artifactId>
       <version>1.0-SNAPSHOT</version>
    <type>ejb-client</type>
  </dependency>
   <dependency>
      <groupId>com.iternum.demo</groupId>
      <artifactId>SimpleAccountInfoEAR</artifactId>
      <version>1.0-SNAPSHOT</version>
      <type>ear</type>
      <scope>compile</scope>
    </dependency>
  <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>javaee</groupId>
      <artifactId>javaee</artifactId>
      <version>5</version>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>com.sun.appserv</groupId>
      <artifactId>appserv-deployment-client</artifactId>
      <version>1</version>
      <scope>compile</scope>
    </dependency>
    <dependency>
      <groupId>com.sun.appserv</groupId>
      <artifactId>appserv-ext</artifactId>
      <version>1</version>
      <scope>compile</scope>
    </dependency>
    <dependency>
      <groupId>com.sun.appserv</groupId>
      <artifactId>appserv-rt</artifactId>
      <version>1</version>
      <scope>compile</scope>
    </dependency>
    <dependency>
      <groupId>com.sun.appserv</groupId>
      <artifactId>appserv-admin</artifactId>
      <version>1</version>
      <scope>compile</scope>
    </dependency>
    <dependency>
      <groupId>com.sun.appserv</groupId>
      <artifactId>sun-appserv-ant</artifactId>
      <version>1</version>
      <scope>compile</scope>
    </dependency>
 <dependency>
      <groupId>com.sun.appserv</groupId>
      <artifactId>admin-cli</artifactId>
      <version>1</version>
      <scope>compile</scope>
    </dependency>
 <dependency>
      <groupId>jmxremote_optional</groupId>
      <artifactId>jmxremote_optional</artifactId>
      <version>1</version>
      <scope>compile</scope>
 </dependency>
</dependencies>
</project>
```

Finally there is something bizarre in the way the Maven multi-module project build works. It runs the same build phase against all its parts. So it is meaningful to run mvn clean or mvn install on the individual modules one after the other. But since the modules usually depend on each other it is rather meaningless to run ant package or ant compile against the entire project – the projects will resolve the dependency not to the current values but to the ones that are already in to local or remote repository. In the effect mvn compile would succeed where one would normally expect it to fail.

## REPORTING

One of the major advantages of Maven is that is offers the site task segment as a very elegant way of building concise documentation for a project. In its plain vanilla version the plugin creates various project related reports, in particular regarding the dependencies of the project, shortcuts into any source code repository and various other things. Its most convenient features are the project reports that can be configured in the pom.xml. This adds another four to five lines per plugin. Some of the most common source code analysis tools are available as maven plugins, in particular Unit Test Reporting, Checkstyle and JavaDoc.

Something like this can be achieved using ant as well, with about the same number of lines per report, however it will normally require great effort to be able to create a layout that is as concise as the one created with Maven.

```

<reporting>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-surefire-report-plugin</artifactId>
            <version>2.4.3</version>
        </plugin>
        <plugin>
             <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-checkstyle-plugin</artifactId>
         </plugin>
```

```
          <plugin>
             <groupId>org.apache.maven.plugins</groupId>
             <artifactId>maven-javadoc-plugin</artifactId>
         </plugin>
     </plugins>
</reporting>
```

## DEPENDENCY MANAGMENT AND REPOSITORIES

Of course the ant build process does not offer any of the Maven dependency management. Assuming that the dependency resolution works properly - which it does for a lot of circumstances - this seems an overwhelming argument to use Maven. It may require some more configurations, but it saves you the effort of finding all your required libraries piecemeal on the internet.

Unfortunately, the real world is somewhat different: Not all your required dependencies are going to be in public repositories so you'll have to download them manually anyway and

- Either install them or
- Deploy them in your own repository

Installing dependencies locally seems like a good idea but it will get you nowhere of course. It is definitely more efficient and less error prone to check in the required files with the project itself rather than rely on each developer to download and deploy it separately. You need to keep them in a safe place anyway. Which leads to the following statement.

**If you use Maven in anything less than a little library project you most likely must maintain your own repository**

The caveat with this approach is of course that any central resource needs some maintenance effort, storage and network resources, backup processes and so on. Moreover the repository itself can introduce subtle inconsistencies into your build process. For example we had to track down a problem with a popular Maven repository where the timestamps of the snapshot builds got set incorrectly, which in turn made it very hard to work on a large scale project simultaneously with multiple developers. They would just not get the required version of the dependencies.

### Dependency management

When you read about Maven one of its greatest strengths in most people’s opinion is that it provides this cool feature called dependency management. And as I said above, it is a good feature that works well. But do you really need it? Most projects will never change their dependencies during the course of the project nor will they be updated to a newer version of a dependency regularly. They will be built against a particular dependency configuration, tested against it and released against it. Even in large organizations, dependencies are reluctantly updated to newer versions, since they require a retest of the entire dependent application. So for small to medium sized projects, you might be better off putting your dependencies into the source repository. It might seem like a productivity loss at first but at the time of this writing Maven Builds are slow and are only weaky integrated into some of the most popular IDEs (Eclipse to name one). So the productity that you may gain using Maven can quickly be lost in your day-to-day development work. Also, building projects with ant is still significantly faster than doing the same in Maven. And one of the greatest perils in software development are quick turnaround times for build – test – debug cycles.

An interesting aspect about quality management and dependency management enters the scene with Maven. Strictly speaking, when I run two successive builds of the same application on different targets, I would assume that the result is identical as long as I have not changed any of my application settings. Using Maven, you can't strictly guarantee that unless you run your entire repository yourself. Because Maven uses dependencies in its own builds system, it is by definition not stable. I have never seen that this is a practical problem – and it shouldn't be if the repository is used as it is supposed to, but still it is a weakness by design.

## Self supporting projects

As a consultant, I like to be as self sufficient as possible. Consider another project that depends upon the project discussed here.

Now with Maven things get complicated. I like to take projects home from the office to work on them on the train or at home. In the old days, all I did was check out the project and of I went with all the required stuff to build the project.

I might not have some backend functionality, but at the very least all my code would compile and various kinds of unit tests could be run. Not anymore! Because I cannot connect my (private) laptop to a company or customer site, I will need to copy all the required files around. Yet, copying entire local repositories can be messy. And unfortunately you can't be sure that your local repository is up to date, before you have built it at least once on a local machine.

## Multiple dependency trees per project

Whenever remoting is performed, in particular with EJBs, strange things can happen to the dependency tree that is maintained by Maven. The Client EJB will have the same dependencies than the server EJB. In our example, the clients will be dependent on commons-logging, even though the library is only used on the server.

```
[INFO] [dependency:list]
[INFO]
[INFO] The following files have been resolved:
[INFO]    com.iternum.demo:SimpleAccountInfoEJB:ejb-
client:client:1.0-SNAPSHOT:c
ompile
[INFO]    com.sun.appserv:appserv-deployment-client:jar:1:test
[INFO]    com.sun.appserv:appserv-ext:jar:1:test
[INFO]    com.sun.appserv:appserv-rt:jar:1:test
[INFO]    commons-logging:commons-logging:jar:1.1.1:compile
[INFO]    javaee:javaee:jar:5:provided
[INFO]    junit:junit:jar:3.8.1:test
```

This is of course plain wrong. Any client will now download the server side dependencies. Obviously this can cause various unnecessary conflicts and on top of it will create client packages of unnecessarily large size. The obvious solution would be to be able to define the dependencies per artefact, but that turns out not to be an option. The only way around this effect that I am aware of is to effectively prevent the dependencies when building the ejb itself and to include them again when building the deployable EAR file. Which is acceptable since both are within the same parent project. We do this by moving all dependencies that are only required on the server from scope compile or runtime to scope provided in the pom.xml for the EJB subproject. In the given example this is just a single dependency - in any real situation it will be a lot more of course.

```
<dependency>
      <groupId>commons-logging</groupId>
      <artifactId>commons-logging</artifactId>
      <version>1.1.1</version>
      <scope>provided</scope>
</dependency>
```

To properly build the ear, the dependency must be added to the respective build file of course.

```
<dependency>
      <groupId>commons-logging</groupId>
      <artifactId>commons-logging</artifactId>
      <version>1.1.1</version>
      <scope>package</scope>
</dependency>
```

Altogether this will increase the overall size of the Maven build configuration by six lines per dependency, since all dependencies must be repeated in the EAR build file. And of course it creates a little additional management overhead since the client and server dependencies must be managed on both sides.

## Viral effects

As has been shown, Maven does not offer the freedom that Ant offers when building your software projects. In fact it is a bit patronizing. It won't built, if not connected to the internet, it wants you to but your files where it wants them and if not you will have to do a lot of explanation (as configuration). But most importantly it tends to patronize your friends as well.

Once a project has been built and published with Maven by far the easiest way to anything dependent on the project to use it is to turn into a Maven project by itself. The existence of the Maven repository is a string incentive for other projects to be set up or changed to be Maven projects as well. This is to a certain extent a "viral" effect as it promotes the spread of a certain development model throughout the organization. And it is of course intended and would generally be considered a good thing. But as we have seen, Maven build configuration can be rather complicated and verbose for complex project structures, so in the end the intended streamlining of the build process may turn out to have made it simply more complex.

## CONCLUSION

By now it should has become clear that the perception that Maven build files are smaller and vastly easier to handle than Ant build files is just that – a perception. The table below shows the approximate number of lines of Ant and Maven Configuration that is required in each phase of the build.

| Application Type | Maven config lines | Ant config lines |
|---|---|---|
| **Standalone** | 26 | 60 |
| **Standalone (with JDK)** | 39 | 60 |
| **EJB** | 160 | 68 |
| **EJB with Deployment and Integration test** | 240 | 97 |

TABLE 2 APPROXIMATE LINES OF CONFIG FOR VARIOUS BUILD TYPES

So when starting a new project, should you use Maven? If it is a standalone library or application, by all means, do! Just don't believe you will end up with shorter and easier to read build files or faster build cycles or better IDE integration than with tradition Ant scripted builds. However you will end up with managed dependencies, which can be worth the effort. And you will be able to create human readable concise documentation for your project more or less out of the box, which, if nothing else, may impress management and allow you to fulfil your documentation quotas.

If your application is a multi component application running inside a particular infrastructure, the decision is not always quiet as easy. In particular for the integration testing phase of the application, you will quite often have to rely on some existing Ant tasks and the antrun plugin. Also, the build cycle may tend to stray to something rather illogical in a multi module project - such as having to deploy an artefact like the EJB JAR file, even though it will only ever be needed within the scope of the build itself.

One caveat though: If you cannot rely entirely on public repositories – which is the normal case for all in house developments in large organizations – you will need to deploy and maintain a central repository. A lot of development departments do this anyway, so it may turn out not be an issue but it is overhead – and overhead that is frequently underestimated.

Should you port your existing projects to Maven? Tempting as it may be, normally the answer is no. *If it ain' t broken, don't fix it* is a common saying in software development and it holds especially true when porting a complicated project from an existing build system – most likely Ant – to Maven. The main reason is that your current build architecture may just not fit in with Mavens approach.

Making your structure fit usually means changing your whole source layout, breaking your project into multiple modules and quiet often loosing traceability in your source code management system. You will often end up running a lot of antrun tasks within your build without a proper chance of ever really switching. There are however good reasons of using Mavens dependency management in existing projects. This is particularly interesting if your project has quickly changing dependencies on other in house projects. Here, the usage of the maven-ant-tasks can be a good compromise where you use Maven solely for dependency management and possible reporting while the actual application build can be performed by whatever structure you have in place with minimal changes.

Finally, keep in mind that there are different ways to skin a cat. The examples presented here can be refactored to cut down the configuration sizes to some extent. For example, the integration phase that is bundled in its own Maven project can be moved to the project that builds the the EAR and be bound to the integration test phase. Or the rather verbose ant code that performs the testing can be refactored into a common reusable task.