Maven Kicking the tyres without changing your project structure

So you're interested in *kicking the maven tyres*, so to speak, or just want to see what it's all about. The following provides hints on how to try maven with your current WebObjects projects, if say you're using the standard WOLips ant builds, without having to adopt a different file structure. Whilst this is not the recommended approach for the long term it allows you to try things out side-by-side with your current build system.

> **Recommended Homework (or pre-requisites)**
>
> It's *really* worth doing your homework on maven in order to understand it. The place to start is Learning Maven found at http://maven.apache.org. Various guides are also found at http://maven.apache.org/guides/.
>
> At the very least you want to have read through, and understood, the Getting Started Tutorial.
>
> The maven user mailing list is also recommended for getting help. It's quite active and, as you find for the WebObjects mailing lists, is an invaluable resource. See Getting Help.

> **Hang in there**
>
> This particular guide might look long but some of the xml is duplicated a few times to show differing examples.

Why Maven

This might be stating the obvious, but an OO developer will, in the course of time (or is *supposed* to anyway), build up various encapsulated, *reus able*, libraries or frameworks that can be tapped into for differing projects. In addition, those frameworks or libraries will themselves often depend on third party frameworks like apache commons, log4j, or WebObjects.

It can be extraordinarily tedious to manage downloading, installing, compiling, and packaging these dependencies. Just finding a particular version of commons-logging-1.1.jar can take 20 minutes. Then everyone in your workgroup has to agree where to put it, and copy it over. If you decide to update to 1.1.1, you have to talk to everyone in your workgroup again, remember to put it into production when you deploy, etc.

This is not a new problem in computer science. There are other tools that attempt to solve this problem, maven just takes it beyond just the build stage into nightly builds, running tests, packaging, deploying, etc.

So in essence, the goal of maven is to automate even more of the whole build/test/install process then is currently done, even to the point of downloading software needed as part of the build. In addition, maven emphasizes *standards* over *configuration*. In WebObjects terms, that's a fancy way of saying that if you put your .wo files in Components, maven will know they need to go into the Resources folder in the .woa.

So while you still have to provide maven with information on the dependencies, if you use the standard locations for things, you won't have to specify much else.

A sample build

So let's assume we have multiple frameworks and applications in our build. Each of these has some common ground, such as their dependencies on certain WebObjects frameworks, or the file layout, and of course they each may have something distinctive about them.

The layout of the frameworks and applications might look like this:

```
/trunk/
/trunk/apps/
/trunk/apps/ApplicationA/
/trunk/apps/ApplicationB/
/trunk/frameworks/
/trunk/frameworks/CustomExtensions/
/trunk/frameworks/CustomBusinessLogic/
/trunk/frameworks/etc/
```

This is a pretty standard way for many WO developers to group their projects. Framework projects go into frameworks, apps into apps. We can

leverage that standard layout to accomplish two things:

1. put as much configuration as possible that's shared between all frameworks, for example, into /frameworks/pom.xml so we only have to define it once. The configuration is inherited by a child pom. This makes the child pom.xml files simpler.
2. Be able to issue a single command that will package each and every framework and application.

Key Concepts

Typical things that make up a pom are as follows. (Note: only pom identification is mandatory. All the others have defaults.)

1. pom identification (who am I?)
   The base triplet used to identify an artifact (i.e., something you need to build/package/install)

```
<artifactId>CustomExtensions</artifactId>
  <groupId>com.mywostuff.frameworks</groupId>
  <version>0.0.1-SNAPSHOT</version>
```

2. pom packaging (i.e., what are we building?)
   The default value for the packaging element is JAR if not specified. For the purposes of this exercise, we'll use JAR for the frameworks and woapplication for the applications, which requires the woproject maven plugin (TODO revisit this scenario with the apple maven plugin)

```
<packaging>woapplication</packaging>
```

3. pom parent identification (who do I belong to?)

```
<parent>
    <artifactId>frameworks</artifactId>
    <groupId>com.mywostuff</groupId>
  </parent>
```

4. modules (a.k.a kids; who belongs to me?)

```
<modules>
    <module>CustomExtensions</module>
    <module>CustomBusinessLogic</module>
  </modules>
```

5. dependencies (what do I need?)

```
<dependencies>
    <dependency>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
      <version>1.2.12</version>
      <scope>compile</scope>
    </dependency>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.4</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
```

6. build sources/resources (what do I have?)
7. properties and filtering resources (variable definitions)
8. dependency/plugin management (shared configuration and versioning)
9. repositories (where to find dependencies and plugins)

Of course, with the plethora of plugins available for maven, this is only the tip of the iceberg. However, these main concepts will suffice for now.

Alternate File System Layout Concepts

As you would (i.e., should) have read by now, Maven has what it calls *standards*. One such standard is the standard directory layout. One of the advantages of following the standards is that you get something for free: you have less to configure (or even almost nothing) in order to build a jar, for example, from your sources and resources. When that's not possible, options are available that allow you to *subvert* these standards or provide extra resouces.

> **Mavan Model Reference Doco**
>
> To see what built-in options are available for maven see Maven Model.

In this case though, we're just trying to "kick the tyres", so we don't want to have to move our files around. The following roughly resembles the current WebObjects WOLips produced project layout (a.k.a Fluffy Bunny layout).

```
/MyProject
/MyProject/Components
/MyProject/Resources
/MyProject/Sources
/MyProject/Tests
/MyProject/WebServerResources
```

Assuming your building a framework, for example, the following is an extract from the relevant pom.xml. It specifies where to find your java source files and resources, we can put this in /trunk/pom.xml and then all the child pom.xml files will know we're using Fluffy Bunny Layout. Notice we've also defined the target path for each resource. (See the WOL:Maven Model#class_resource for a definition of targetPath)

pom.xml

```
<...>
  <build>
    <sourceDirectory>Sources</sourceDirectory>
    <testSourceDirectory>Tests</testSourceDirectory>
    <resources>
      <resource>
        <targetPath>Resources</targetPath>
        <filtering>false</filtering>
        <directory>Components</directory>
      </resource>
      <resource>
        <targetPath>Resources</targetPath>
        <filtering>false</filtering>
        <directory>Resources</directory>
      </resource>
      <resource>
        <targetPath>WebServerResources</targetPath>
        <filtering>false</filtering>
        <directory>WebServerResources</directory>
      </resource>
    </resources>
    <...>
  </build>
<...>
```

So, concentrating on our frameworks alone for the moment, assuming all of your frameworks share the above project layout the above can happily go into your /frameworks/pom.xml file and as such be shared by all sub-modules (i.e., frameworks).

Project Dependencies Concepts

Most projects, of course, have dependencies on other libraries or frameworks. See the WOL:Maven Getting Started#How_do_I_use_external_dependencies.

The following shows the mixture of third party dependencies and custom framework dependencies. Notice that the scope element determines the life cycle phase each dependency is relevant for. See WOL:Maven Model#class_dependency for specific definitions.

```
<dependencies>
    <dependency>
      <artifactId>CustomExtensions</artifactId>
      <groupId>com.mywostuff.frameworks</groupId>
      <version>0.0.1-SNAPSHOT</version>
      <scope>compile</scope>
    </dependency>
    <dependency>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
      <version>1.2.12</version>
      <scope>compile</scope>
    </dependency>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.4</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
```

Project Inheritance

It naturally gets a bit boring having to define the same things over and over again. So, you can utilise a parent pom file specifying its packaging as 'pom'. Dependencies, plugins and executions, resources specifications and so forth can be defined once and shared by any sub-modules. See http://maven.apache.org/guides/introduction/introduction-to-the-pom.html#Project_Inheritance and Java World's The Maven 2 POM demystified for further information and examples.

For our example we'll have trunk/pom.xml which will define everything common to any and all modules in the hierarchy. Likewise, trunk/frameworks/pom.xml and trunk/apps/pom.xml will define everything common to frameworks and applications respecively.

Repositories

So far we have assumed that maven just knows where to find third party libraries. There is the default local repository (e.g., ~/.m2/repository) and a remote one at ibiblio.org or a mirror of the same. See http://maven.apache.org/guides/introduction/introduction-to-repositories.html. Repositories are what lets you specify, "my app needs commons-logging-1.1.1" and maven can then pull it into the build as needed. Here we're adding some additional repositories to the defaults. You might want to setup one for your workgroup, and then there are some useful WO-related ones as well. We can include this in the master trunk/pom.xml file, then all the children can use it.

```
<repositories>
    <repository>
        <id>system-repo</id>
        <name>internal repository</name>
        <!-- TODO switch over to your intranet.domain -->
        <url>file://${user.home}/.myarchiva</url>
        <snapshots>
            <enabled>true</enabled>
            <updatePolicy>always</updatePolicy>
            <checksumPolicy>ignore</checksumPolicy>
        </snapshots>
        <releases>
            <enabled>true</enabled>
            <updatePolicy>always</updatePolicy>
```

```
            <checksumPolicy>ignore</checksumPolicy>
        </releases>
    </repository>
    <repository>
        <id>maven2-repository.dev.java.net</id>
        <name>Java.net Repository for Maven</name>
        <url>http://download.java.net/maven/2</url>
        <snapshots>
            <enabled>true</enabled>
            <updatePolicy>daily</updatePolicy>
        </snapshots>
        <releases>
            <enabled>true</enabled>
            <updatePolicy>daily</updatePolicy>
        </releases>
    </repository>
    <repository>
        <id>webobjects.mdimension.com/releases</id>
        <name>mdimension maven 2 releases repo</name>
        <url>http://webobjects.mdimension.com/maven2/releases</url>
        <snapshots>
            <enabled>false</enabled>
        </snapshots>
        <releases>
            <enabled>true</enabled>
        </releases>
    </repository>
    <repository>
        <id>webobjects.mdimension.com/snapshots</id>
        <name>mdimension maven 2 snapshots repo</name>
        <url>http://webobjects.mdimension.com/maven2/snapshots</url>
        <snapshots>
            <enabled>true</enabled>
        </snapshots>
        <releases>
            <enabled>false</enabled>
        </releases>
    </repository>
    <repository>
        <id>objectstyle-maven2</id>
        <name>objectstyle maven2 repo</name>
        <url>http://objectstyle.org/maven2</url>
        <snapshots>
            <enabled>false</enabled>
        </snapshots>
        <releases>
            <enabled>true</enabled>
        </releases>
```

```
    </repository>
</repositories>
```

Note: A remote repository is not guaranteed to keep older versions of libraries, for example, indefinitely. This is why it's recommended that you set up one for your intranet which stores what you need for longevity. See both the above intro to repositories and http://www.theserverside.com/tt/articles/article.tss?l=SettingUpMavenRepository.

Packaging Frameworks as Jars

Here's the definition for /frameworks/pom.xml, definitions here will be shared by all of the individual framework pom.xml files. Note that it depends on the following Info.plist file being located under trunk/frameworks/src/main/resources (maven builds can use files stored in common off of a shared structure):

File	Modified
File Info.plist frameworks/src/main/resources/Info.plist	Jun 23, 2008 by Lachlan Deck

Labels

- No labels

/frameworks/pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<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>

    <!-- parent artifact -->
    <parent>
        <artifactId>mywostuff</artifactId>
        <groupId>com</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>

    <!-- artifact identity -->
    <artifactId>frameworks</artifactId>
    <groupId>com.mywostuff</groupId>
    <packaging>pom</packaging>

    <!-- framework relevant properties -->
    <properties>
        <!-- NS related properties fills in Info.plist etc-->
        <CFBundleDevelopmentRegion>English</CFBundleDevelopmentRegion>
        <CFBundleGetInfoString></CFBundleGetInfoString>
        <CFBundlePackageType>FMWK</CFBundlePackageType>
        <CFBundleIconFile>WOAfile.icns</CFBundleIconFile>
        <CFBundleInfoDictionaryVersion>6.0</CFBundleInfoDictionaryVersion>
        <CFBundleVersion>5.3.1</CFBundleVersion>
        <Has_WOComponents>true</Has_WOComponents>
```

```
    <NSPrincipalClass>${mainclass}</NSPrincipalClass>
    <NSResourcesBundlePath></NSResourcesBundlePath>
</properties>

<!-- modules -->
<modules>
    <module>CustomExtensions</module>
    <module>CustomBusinessLogic</module>
</modules>

<!-- specific dependencies (for modules) -->
<dependencies>
    <dependency>
        <artifactId>ERExtensions</artifactId>
        <groupId>${wonder.common.groupId}</groupId>
    </dependency>
    <dependency>
        <artifactId>JavaWOExtensions</artifactId>
        <groupId>${wonder.common.groupId}</groupId>
    </dependency>
    <dependency>
        <artifactId>JavaFoundation</artifactId>
        <groupId>${webobjects.groupId}</groupId>
    </dependency>
    <dependency>
        <artifactId>JavaJDBCAdaptor</artifactId>
        <groupId>${webobjects.groupId}</groupId>
    </dependency>
    <dependency>
        <artifactId>JavaWebObjects</artifactId>
        <groupId>${webobjects.groupId}</groupId>
    </dependency>
    <dependency>
        <artifactId>JavaEOControl</artifactId>
        <groupId>${webobjects.groupId}</groupId>
    </dependency>
    <dependency>
        <artifactId>JavaEOAccess</artifactId>
        <groupId>${webobjects.groupId}</groupId>
    </dependency>
    <dependency>
        <artifactId>JavaWebObjects</artifactId>
        <groupId>${webobjects.groupId}</groupId>
    </dependency>
    <dependency>
        <artifactId>JavaXML</artifactId>
        <groupId>${webobjects.groupId}</groupId>
    </dependency>
</dependencies>


<!-- build config (for modules) -->
<build>
```

```
<sourceDirectory>src</sourceDirectory>
<testSourceDirectory>tests</testSourceDirectory>
<resources>
    <!-- relative dir for Info.plist -->
    <resource>
        <targetPath>Resources</targetPath>
        <filtering>true</filtering>
        <directory>../src/main/resources</directory>
    </resource>
    <resource>
        <targetPath>Resources</targetPath>
        <filtering>false</filtering>
        <directory>Components</directory>
    </resource>
    <resource>
        <targetPath>Resources</targetPath>
        <filtering>false</filtering>
        <directory>Resources</directory>
    </resource>
    <resource>
        <targetPath>WebServerResources</targetPath>
        <filtering>false</filtering>
        <directory>WebServerResources</directory>
    </resource>
</resources>
<plugins>
    <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
            <source>${java.target}</source>
            <target>${java.target}</target>
        </configuration>
    </plugin>
</plugins>
```

```
    </build>
</project>
```

Since our CustomExtensions has no further dependencies, its pom.xml merely specifies its parent and its identity.

/frameworks/CustomBusinessLogic/pom.xml

```
<?xml version="1.0"?>
<project>
    <!-- parent artifact -->
    <parent>
        <artifactId>frameworks</artifactId>
        <groupId>com.mywostuff</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>

    <!-- artifact identity -->
    <artifactId>CustomBusinessLogic</artifactId>
    <groupId>com.mywostuff.frameworks</groupId>
</project>
```

CustomBusinessLogic has a further dependency on CustomExtensions, so it specifies its parent, its identity, and the dependency.

/frameworks/CustomBusinessLogic/pom.xml

```
<?xml version="1.0"?>
<project>
    <!-- parent artifact -->
    <parent>
        <artifactId>frameworks</artifactId>
        <groupId>com.mywostuff</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>

    <!-- artifact identity -->
    <artifactId>CustomBusinessLogic</artifactId>
    <groupId>com.mywostuff.frameworks</groupId>

    <!-- specific dependencies -->
    <dependencies>
        <dependency>
            <artifactId>CustomExtensions</artifactId>
            <groupId>${pom.groupId}</groupId>
        </dependency>
    </dependencies>
</project>
```

Packaging Applications

Here's the definition for /apps/pom.xml which is shared by any sub-modules (i.e., ApplicationA and ApplicationB). Both apps need certain

WebObjects frameworks, so we specify those only once for both, here in the parent pom. We also specify Fluffy Bunny Layout, and some maven plugins we want to use. Again, this is for both applications.

/apps/pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<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>

    <!-- parent artifact -->
    <parent>
        <groupId>com</groupId>
        <artifactId>mywostuff</artifactId>
        <version>1.0-SNAPSHOT</version>
    </parent>

    <!-- artifact identity -->
    <artifactId>apps</artifactId>
    <groupId>com.mywostuff</groupId>
    <packaging>pom</packaging>

    <!-- modules -->
    <modules>
        <module>ApplicationA</module>
        <module>ApplicationB</module>
    </modules>

    <!-- specific dependencies (for modules) -->
    <dependencies>
        <!-- wonder frameworks -->
        <dependency>
            <artifactId>ERExtensions</artifactId>
            <groupId>${wonder.common.groupId}</groupId>
        </dependency>
        <dependency>
            <artifactId>JavaWOExtensions</artifactId>
            <groupId>${wonder.common.groupId}</groupId>
        </dependency>

        <!-- project libs -->
        <dependency>
            <artifactId>CustomExtensions</artifactId>
            <groupId>${my.frameworks.groupId}</groupId>
        </dependency>
        <dependency>
            <artifactId>CustomBusinessLogic</artifactId>
            <groupId>${my.frameworks.groupId}</groupId>
        </dependency>

        <!-- webobjects dependencies -->
```

```
        <dependency>
            <artifactId>JavaFoundation</artifactId>
            <groupId>${webobjects.groupId}</groupId>
        </dependency>
        <dependency>
            <artifactId>JavaJDBCAdaptor</artifactId>
            <groupId>${webobjects.groupId}</groupId>
        </dependency>
        <dependency>
            <artifactId>JavaWebObjects</artifactId>
            <groupId>${webobjects.groupId}</groupId>
        </dependency>
        <dependency>
            <artifactId>JavaEOControl</artifactId>
            <groupId>${webobjects.groupId}</groupId>
        </dependency>
        <dependency>
            <artifactId>JavaEOAccess</artifactId>
            <groupId>${webobjects.groupId}</groupId>
        </dependency>
        <dependency>
            <artifactId>JavaWebObjects</artifactId>
            <groupId>${webobjects.groupId}</groupId>
        </dependency>
        <dependency>
            <artifactId>JavaXML</artifactId>
            <groupId>${webobjects.groupId}</groupId>
        </dependency>
    </dependencies>

    <!-- build config (for modules) -->
    <build>
        <sourceDirectory>src</sourceDirectory>
        <testSourceDirectory>tests</testSourceDirectory>
        <resources>
            <resource>
                <targetPath>Resources</targetPath>
                <filtering>false</filtering>
                <directory>Components</directory>
            </resource>
            <resource>
                <targetPath>Resources</targetPath>
                <filtering>false</filtering>
                <directory>Resources</directory>
            </resource>
            <resource>
                <targetPath>WebServerResources</targetPath>
                <filtering>false</filtering>
                <directory>WebServerResources</directory>
            </resource>
        </resources>
        <plugins>
            <plugin>
```

```
            <artifactId>maven-wolifecycle-plugin</artifactId>
            <groupId>org.objectstyle.woproject.maven2</groupId>
            <version>2.0.15</version>
            <extensions>true</extensions>
            <configuration>
                <source>${java.target}</source>
                <target>${java.target}</target>
            </configuration>
        </plugin>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-javadoc-plugin</artifactId>
            <configuration>
                <javadocVersion>${java.target}</javadocVersion>
                <locale>en-AU</locale>
                <minmemory>128m</minmemory>
                <maxmemory>512m</maxmemory>
            </configuration>
        </plugin>
        <!--
        TODO build numbering
        <plugin>
            <groupId>org.codehaus.mojo</groupId>
            <artifactId>maven-buildnumber-plugin</artifactId>
            <version>0.9.6</version>
            <executions>
                <execution>
                    <phase>validate</phase>
                    <goals>
                        <goal>create</goal>
                    </goals>
                </execution>
            </executions>
            <configuration>
                <doCheck>true</doCheck>
                <doUpdate>true</doUpdate>
            </configuration>
        </plugin>
        -->
    </plugins>
    <pluginManagement>
        <plugins>
            <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <configuration>
                <source>${java.target}</source>
                <target>${java.target}</target>
            </configuration>
        </plugin>
        </plugins>
    </pluginManagement>
```

```
    </build>
</project>
```

With most stuff specified in the parent pom, ApplicationA needs only to specify its parent, its idenity, and add a couple of extra specific dependencies to those inherited from its parent.

/apps/ApplicationA/pom.xml

```
<?xml version="1.0"?>
<project>
    <modelVersion>4.0.0</modelVersion>

    <!-- parent artifact -->
    <parent>
        <artifactId>apps</artifactId>
        <groupId>com.mywostuff</groupId>
        <version>1.0-SNAPSHOT</version>
        <relativePath>../apps</relativePath> <!-- e.g., (optional) if your
app is under /trunk -->
    </parent>

    <!-- artifact identity -->
    <artifactId>ApplicationA</artifactId>
    <groupId>com.mywostuff.apps</groupId>
    <packaging>woapplication</packaging> <!-- woproject specific packaging
-->

    <!-- specific properties -->
    <properties>
        <!-- general properties -->
        <mainclass>your.app.Application</mainclass>
    </properties>

    <!-- specific dependencies -->
    <dependencies>
        <!-- wonder frameworks -->
        <dependency>
            <artifactId>Ajax</artifactId>
            <groupId>${wonder.ajax.groupId}</groupId>
        </dependency>
        <dependency>
            <artifactId>ERCaptcha</artifactId>
            <groupId>${wonder.common.groupId}</groupId>
            <!-- requires jcaptcha-all below -->
        </dependency>
        <dependency>
            <artifactId>WOOgnl</artifactId>
            <groupId>${wonder.common.groupId}</groupId>
        </dependency>

        <!-- general libs -->
        <dependency>
```

```
    <artifactId>jcaptcha-all</artifactId>
    <groupId>com.octo.captcha</groupId>
</dependency>
<dependency>
    <artifactId>commons-collections</artifactId>
    <groupId>commons-collections</groupId>
</dependency>
<dependency>
    <groupId>ognl</groupId>
    <artifactId>ognl</artifactId>
</dependency>
```

```
    </dependencies>
</project>
```

Packaging Applications as True WAR

You can find steps to package WO Applications as True WAR here.

Eclipse Integration

details to come...

Putting It All Together

details to come...