Technology Conversations

Java Build Tools: Ant vs Maven vs Gradle

In the beginning there was Make as the only build tool available. Later on it was improved with GNU Make. However, since then our needs increased and, as a result, build tools evolved.

JVM ecosystem is dominated with three build tools:

- Apache Ant with Ivy
- Maven
- Gradle

Ant with Ivy

Ant was the first among "modern" build tools. In many aspects it is similar to Make. It was released in 2000 and in a short period of time became the most popular build tool for Java projects. It has very low learning curve thus allowing anyone to start using it without any special preparation. It is based on procedural programming idea.

After its initial release, it was improved with the ability to accept plug-ins.

Major drawback was XML as the format to write build scripts. XML, being hierarchical in nature, is not a good fit for procedural programming approach Ant uses. Another problem with Ant is that its XML tends to become unmanageably big when used with all but very small projects.

Later on, as dependency management over the network became a must, Ant adopted Apache Ivy.

Main benefit of Ant is its control of the build process.

Maven

Maven was released in 2004. Its goal was to improve upon some of the problems developers were facing when using Ant.

Maven continues using XML as the format to write build specification. However, structure is diametrically different. While Ant requires developers to write all the commands that lead to the successful execution of some task, Maven relies on conventions and provides the available targets (goals) that can be invoked. As the additional, and probably most important addition, Maven introduced the ability to download dependencies over the network (later on adopted by Ant through Ivy). That in itself revolutionized the way we deliver software.

However, Maven has its own problems. Dependencies management does not handle conflicts well between different versions of the same library (something Ivy is much better at). XML as the build configuration format is strictly structured and highly standardized. Customization of targets (goals) is hard. Since Maven is focused mostly on dependency management, complex, customized build scripts are actually harder to write in Maven than in Ant.

Maven configuration written in XML continuous being big and cumbersome. On bigger projects it can have hundreds of lines of code without actually doing anything "extraordinary".

Main benefit from Maven is its life-cycle. As long as the project is based on certain standards, with Maven one can pass through the whole life cycle with relative ease. This comes at a cost of flexibility.

In the mean time the interest for DSLs (Domain Specific Languages) continued increasing. The idea is to have languages designed to solve problems belonging to a specific domain. In case of builds, one of the results of applying DSL is Gradle.

Gradle

Gradle combines good parts of both tools and builds on top of them with DSL and other improvements. It has Ant's power and flexibility with Maven's life-cycle and ease of use. The end result is a tool that was released in 2012 and gained a lot of attention in a short period of time. For example, Google adopted Gradle as the default build tool for the Android OS.

Gradle does not use XML. Instead, it had its own DSL based on Groovy (one of JVM languages). As a result, Gradle build scripts tend to be much shorter and clearer than those written for Ant or Maven. The amount of boilerplate code is much smaller with Gradle since its DSL is designed to solve a specific problem: move software through its life cycle, from compilation through static analysis and testing until packaging and deployment.

Initially, Gradle used Apache Ivy for its dependency management. Later own it moved to its own native dependency resolution engine.

Gradle effort can be summed as "convention is good and so is flexibility".

Code examples

We'll create build scripts that will compile, perform static analysis, run unit tests and, finally, create JAR files. We'll do those operations in all three frameworks (Ant, Maven and Gradle) and compare the syntax. By comparing the code for each task we'll be able to get a better understanding of the differences and make an informed decision regarding the choice of the build tool.

First things first. If you'll do the examples from this article by yourself, you'll need Ant, Ivy, Maven and Gradle installed. Please follow installation instructions provided by makers of those tools. You can choose not to run examples by yourself and skip the installation altogether. Code snippets should be enough to give you the basic idea of how each of the tools work.

Code repository https://github.com/vfarcic/JavaBuildTools contains the java code (two simple classes with corresponding tests), checkstyle configuration and Ant, Ivy, Maven and Gradle configuration files.

Let's start with Ant and Ivy.

Ant with Ivy

Ivy dependencies need to be specified in the ivy.xml file. Our example is fairly simple and requires only JUnit and Hamcrest dependencies.

[ivy.xml]

```
<ivy-module version="2.0">
    <info organisation="org.apache" module="java-build-tools"/>
    <dependencies>
        <dependency org="junit" name="junit" rev="4.11"/>
        <dependency org="org.hamcrest" name="hamcrest-all" rev="1.3"/>
    </dependencies>
</ivy-module>
```

Now we'll create our Ant build script. Its task will be only to compile a JAR file. The end result is the following build.xml.

[build.xml]

```
<project xmlns:ivy="antlib:org.apache.ivy.ant" name="java-build-tools" default="jar">

    <property name="src.dir" value="src"/>
    <property name="build.dir" value="build"/>
    <property name="classes.dir" value="${build.dir}/classes"/>
    <property name="jar.dir" value="${build.dir}/jar"/>
    <property name="lib.dir" value="lib" />
    <path id="lib.path.id">
        <fileset dir="${lib.dir}" />
    </path>

    <target name="resolve">
        <ivy:retrieve />
    </target>

    <target name="clean">
        <delete dir="${build.dir}"/>
    </target>

    <target name="compile" depends="resolve">
        <mkdir dir="${classes.dir}"/>
        <javac srcdir="${src.dir}" destdir="${classes.dir}" classpathref="lib.path.id"/>
    </target>

    <target name="jar" depends="compile">
        <mkdir dir="${jar.dir}"/>
        <jar destfile="${jar.dir}/${ant.project.name}.jar" basedir="${classes.dir}"/>
    </target>

</project>
```

First we specify several properties. From there on it is one task after another. We use Ivy to resolve dependencies, clean, compile and, finally, create the JAR file. That is quite a lot of configuration for a task that almost every Java project needs to perform.

To run the Ant task that creates the JAR file, execute following.

```
ant jar
```

Let's see how would Maven does the same set of tasks.

Maven

[pom.xml]

```
<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.technologyconversations</groupId>
    <artifactId>java-build-tools</artifactId>
    <packaging>jar</packaging>
    <version>1.0</version>

    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.11</version>
        </dependency>
        <dependency>
            <groupId>org.hamcrest</groupId>
            <artifactId>hamcrest-all</artifactId>
            <version>1.3</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>2.3.2</version>
            </plugin>
        </plugins>
    </build>

</project>
```

To run the Maven goal that creates the JAR file, execute following.

```
mvn package
```

The major difference is that with Maven we don't need to specify what should be done. We're not creating tasks but setting the parameters (what are the dependencies, what plugins to use...). This shows the major difference between Ant and Maven. Later promotes the usage of conventions and provides goals (targets) out-of-the-box. Both Ant and Maven XML files tend to grow big with time. To illustrate that, we'll add Maven CheckStyle, FindBugs and PMD plugins that will take care of static analysis. All three are fairly standard tools used, in one form or another, in many Java projects. We want all static analysis to be executed as part of a single target **verify** together with unit tests. Moreover, we should specify the path to the custom checkstyle configuration and make sure that it fails on error. Additional Maven code is following:

[pom.xml]

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-checkstyle-plugin</artifactId>
    <version>2.12.1</version>
    <executions>
        <execution>
            <configuration>
                <configLocation>config/checkstyle/checkstyle.xml</configLocation>
                <consoleOutput>true</consoleOutput>
                <failsOnError>true</failsOnError>
            </configuration>
            <goals>
                <goal>check</goal>
            </goals>
        </execution>
    </executions>
</plugin>
<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>findbugs-maven-plugin</artifactId>
    <version>2.5.4</version>
    <executions>
        <execution>
            <goals>
                <goal>check</goal>
            </goals>
        </execution>
    </executions>
</plugin>
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-pmd-plugin</artifactId>
    <version>3.1</version>
    <executions>
        <execution>
            <goals>
                <goal>check</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

To run the Maven goal that runs both unit tests and static analysis with CheckStyle, FindBugs and PMD, execute following.

```
mvn verify
```

We had to write a lot of XML that does some very basic and commonly used set of tasks. On real projects with a lot more dependencies and tasks, Maven pom.xml files can easily reach hundreds or even thousands of lines of XML.

Here's how the same looks in Gradle.

Gradle

[build.gradle]

```
apply plugin: 'java'
apply plugin: 'checkstyle'
apply plugin: 'findbugs'
apply plugin: 'pmd'

version = '1.0'

repositories {
    mavenCentral()
}

dependencies {
    testCompile group: 'junit', name: 'junit', version: '4.11'
    testCompile group: 'org.hamcrest', name: 'hamcrest-all', version: '1.3'
```

```
15 }
```

Not only that the Gradle code is much shorter and, to those familiar with Gradle, easier to understand than Maven, but it actually introduces many useful tasks not covered with the Maven code we just wrote. To get the list of all tasks that Gradle can run with the current configuration, please execute the following.

```
1 gradle tasks --all
```

Clarity, complexity and the learning curve

For newcomers, Ant is the clearest tool of all. Just by reading the configuration XML one can understand what it does. However, writing Ant tasks easily gets very complex. Maven and, specially, Gradle have a lot of tasks already available out-of-the-box or through plugins. For example, by seeing the following line it is probably not clear to those not initiated into mysteries of Gradle what tasks will be unlocked for us to use.

[build.gradle]

```
1 apply plugin: 'java'
```

This simple line of code adds 20+ tasks waiting for us to use.

Ant's readability and Maven's simplicity are, in my opinion, false arguments that apply only during the short initial Gradle learning curve. Once one is used to the Gradle DSL, its syntax is shorter and easier to understand than those employed by Ant or Maven. Moreover, only Gradle offers both conventions and creation of commands. While Maven can be extended with Ant tasks, it is tedious and not very productive. Gradle with Groovy brings it to the next level.

This entry was posted in Java and tagged Ant, build, Gradle, Groovy, Java, Maven on June 18, 2014 [https://technologyconversations.com/2014/06/18/build-tools/] by Viktor Farcic.

---

55 thoughts on "Java Build Tools: Ant vs Maven vs Gradle"

Foobo
June 18, 2014 at 7:00 pm

Gradle is nice if your developers' time is worth nothing.
Yes, the example code looks nice, but it's a lot less declarative than maven, simply because the kind of developer that wants to write scripts instead of XML WILL write scripts, and they make your build unreadable (I'm in a gradle project right now, but I don't know it too well; I only know what the build does because gradle prints out the tasks it executes).

Also, while ant is quite fast (though still slower than Eclipse) and maven is – by now – slower but ok, gradle is incredibly, annoyingly, very very painfully abysmally slow. Yes, I know the daemon. It doesn't work, it only causes the "clean" goal