Ant Abilities for Building Large Projects

Version V0-0-4

Ivan Ivanov - rambiusparkisanius@gmail.com,
Petar Tahchiev - paranoiabla@gmail.com

October 25, 2005

Contents

1. Apache Ant: Introductory Words

Apache Ant is a tool for building (compiling, packaging, testing) projects. Its main features are:

- platform independence - as it is pure-Java application;
- extensible - lots of new components can be added either by coding them in Java, or in other script languages as JavaScript, Python;
- easy, but powerful XML syntax that is easy to master;
- can build not only Java projects, but projects in other languages as well.

1.1. Structure of the build scripts

- Ant uses **`build files`**, which describe how an application is built, or packaged or tested. A build script is an XML file, containing the steps of the application build process. Big projects can consist of smaller subprojects, each of which can have its own build script. Another main build script can coordinate the build processes of the subprojects.
- Every build script contains several **`targets`**, that can depend on each other. In most cases we have target for compiling, target for packaging and target for testing, and the targets for testing and packaging naturally depends on the target for compiling.
- Targets consist of **`tasks`**, which perform some operation. For example, there is task **`<javac>`**, which compiles Java files, as well as task **`<copy>`**, which copies files.

Here is a simple build script:

```
<project name="timestamp" basedir="." default="all">
    <description>Prints time and date</description>

    <tstamp/>

    <target name="print-date"
        description="Prints the current date">
        <echo>Current date is ${DSTAMP}</echo>
    </target>

    <target name="print-time"
        description="Prints the current time">
        <echo>Current time is ${TSTAMP}</echo>
    </target>

    <target name="all" depends="print-date,print-time"
        description="Prints the current date and time"/>
</project>
```

We call a target from this build script in the following manner:

```
$ ant print-date
```

The result is:

```
Buildfile: build.xml

print-date:
     [echo] Current date is 20051006

BUILD SUCCESSFUL
Total time: 0 seconds
```

We can call ant without providing a target. In this case the default target is called, as specified in attribute **`default`** of the project:

```
$ ant
Buildfile: build.xml

print-date:
     [echo] Current date is 20051006

print-time:
     [echo] Current time is 1919

all:

BUILD SUCCESSFUL
Total time: 0 seconds
```

The example shows:

- The usage of task `<echo>`, which print the given message. In this case parts of the message are the values of `properties`, called `DSTAMP`, `TSTAMP`. A property is name-value pair and the value can be retrieved by the name with the notation `${prop_name}`; once the value of a property is set it cannot be changed;
- The usage of task `<tstamp>`, which sets the values of properties named `DSTAMP`, `TSTAMP` to the current date and time respectively;
- It is not mandatory every task to be contained by a target. In our case task `tstamp` is out of a target. Such tasks are always executed, regardless of the invoked target;
- The dependencies of a target on other targets are specified by attribute `depends`, for example target `all` depends on `print-date` and `print-time` and they will be executed automatically when all is invoked;
- The project can have nested tag `<description>`, that describes its usage; targets can have attribute `description` as well, that describes its action; this give us ability to document the script and to dump its documentation later with command `ant -projecthelp`.

Ant has some built-in properties, i.e. they are populated automatically without being declared by the user. These are:

- the system properties of JVM - the same properties which we access with System.getProperties() method;
- Ant specific properties as:
  - **basedir** - the absolute path of the project's basedir as defined in basedir attribute of ¡project¿ tag;
  - **ant.file** - the abosulte path of the current build script;
  - **ant.version** - the Ant versiopn;
  - **ant.project.name** - the name of the current project as specified in name attributeof the ¡project¿ tag;
  - **ant.java.version** - the version of the JVM, that is executing the build script.

Besides we have access to all environment variables, for example if we want to take the value of an environment variable called CATALINA_HOME we use the following construction:

```
<project>
    <property environment="env"/>
    <echo>$${env.CATALINA_HOME} is ${env.CATALINA_HOME}</echo>
</project>
```

1.2. Building and testing a HelloWorld application

The next task is to create a HelloWorld application, that will be compiled, tested and packaged. The application consist of:

- Java code:

```
package net.sourceforge.emaria.openfest;

public class Message {

    public String getMessage(String name) {
        return "Hello World, " + name;
    }
}
```

```
package net.sourceforge.emaria.openfest;

public class Main {

    public static void main(String[] args) {
        Message message = new Message();
        System.out.println(message.getMessage("Sharo"));
    }
}
```

- Unit test code:

```
package net.sourceforge.emaria.openfest;

import junit.framework.TestCase;
import junit.textui.TestRunner;
import net.sourceforge.emaria.openfest.Message;

public class MessageTest extends TestCase {

    public void testHelloWorld() {
        Message helloWorld = new Message();
        assertEquals(helloWorld.getMessage("Sharo"),
            "Hello World, Sharo");
        assertNotNull(helloWorld.getMessage(null));
    }

    public void testHelloWorldFails() {
        Message helloWorld = new Message();
        assertEquals(helloWorld.getMessage("Sharo"),
            "Hello World, OpenFest");
    }
}
```

Here is the very code of build.xml file, which:

- compiles the application:

```
<target name="compile" depends="-init"
    description="Compiles the project's sources">
    <mkdir dir="${project.classes}"/>
    <javac srcdir="${project.src}"
        destdir="${project.classes}"/>
</target>
```

- compiles the unit tests:

```
    <delete dir="${project.tests.classes}"/>
    <delete dir="${project.tests.data}"/>
    <delete dir="${project.tests.reports}"/>
</target>

<target name="test.compile" depends="-test.init,compile"
    description="Compiles the project's unit tests">
    <mkdir dir="${project.tests.classes}"/>
```

- starts the unit tests:

```
            destdir="${project.tests.classes}">
            <classpath refid="compile.tests.cp"/>
        </javac>
    </target>

    <target name="test" depends="test.compile"
        description="Runs the project's unit tests">
        <mkdir dir="${project.tests.reports}"/>
        <mkdir dir="${project.tests.data}"/>
        <junit printsummary="yes" haltonfailure="no"
                haltonerror="no" errorproperty="test.error"
                failureproperty="test.error">
            <classpath refid="runtime.tests.cp"/>
            <formatter type="plain" usefile="false"/>
            <formatter type="xml"/>
            <batchtest fork="yes" todir="${project.tests.data}">
                <fileset dir="${project.tests.classes}">
                    <include name="**/*Test.class"/>
                </fileset>
            </batchtest>
        </junit>
        <junitreport todir="${project.tests.data}">
            <fileset dir="${project.tests.data}">
                <include name="TEST-*.xml"/>
            </fileset>
            <report format="frames"
```

The examples demonstrate:

- Task `<junit>`, that executes all unit tests, picked up by nested element `<batchtest>`;
- Element `formatter`, that transforms the test result in the form specified by attribute `type`;
- Attributes `haltonerror`, `haltonfailure`, that specify whether the build execution will be stopped, if the tests are not successful;
- Attributes `failureproperty` and `errorproperty` specify the names of the properties, that are set if the tests are not successful;
- Task `<junitreport>` transforms the test results from XML format to HTML format via XSLT transformation;
- Task `<fail>`, that stops the build as unsuccessful if a given condition is met;
- In our example we specify the build to not stop if the tests fail, but we specify when they fail to initialize property `test.error`, which we will help us to manually fail the build after the HTML results are generated with <junitreport>.

1.3. First Usage of <presetdef>

Task **<presetdef>** creates a new definition of an already existing task, by predefining some of its attributes or elements. A possible scenario of <presetdef> usage is:

1. Task <javac> by default do not generate debug information, which is needed during development for error tracing; as we saw in HelloWorld application that <javac> is used at least twice in the build script: for compiling the very application and for unit tests compiling;
2. We can cite every time we use <javac>, to cite one and the same values of attributes **debug**, **debuglevel** and **deprecation**, but this leads to literal repetition of code;
3. The solution is to use <presetdef>:

```
<presetdef name="javac">
    <javac debug="on" debuglevel="vars,lines,source"
        deprecation="on"/>
</presetdef>
```

   In this way, every time we use <javac>, we use the above predefined version.
4. This leads to the following problem: debug information increases the size of the compiled classes, which is not desired for the official builds, and every use of <javac> generates this information;

5. The solution is to use properties instead of fixed values for the attributes. Since the value of the properties cannot be changed, all we need to do is to ensure that when we create official builds, the values of the debug properties should be set to off before everything else;
6. In our case we fulfill this by adding the target

```
<target name="javadoc" depends="-init"
    description="Generates project's javadoc">
</target>

<target name="doc" depends="javadoc"
```

and we specify the target for creating the official distribution `dist` to depend on `-dist.init`:

```
</target>

<target name="dist.clean" depends="-init"
```

2. Description of tasks <subant>, <import>, <macrodef>

These tasks, as well as <presetdef> were introduced in Ant 1.6 to ease the create new tasks without coding them in Java, to simplify the build process in large projects and the reuse of existing build scripts.

2.1. Defining new tasks with <macrodef>

Task **<macrodef>** defines new tasks and the input data of these new tasks task-are specified with elements **<attribute>** **<element>**.
Here is a useful <macrodef>, that opens a given HTML file in the default browser, by using Linux command **htmlview**:

```
<macrodef name="browse">
    <attribute name="file"/>
    <sequential>
        <exec executable="htmlview" spawn="yes">
            <arg file="@{file}"/>
        </exec>
    </sequential>
</macrodef>
```

It is used as all other tasks:

```
<target name="test-browser" depends="browserDef"
    description="Tests the macrodef">
    <browse file="test-macrodef.html"/>
</target>
```

We can write a Windows version of this <macrodef>

```
<macrodef name="browse">
    <attribute name="file"/>
    <sequential>
        <exec executable="rundll32.exe">
            <arg line="url.dll, FileProtocolHandler"/>
            <arg file="@{file}"/>
        </exec>
    </sequential>
</macrodef>
```

These two version can be combined in one build script so that task <browse> will work both on Windows and Linux:

```
<target name="checkOS"
    description="Checks the operating system">
    <condition property="isWindows">
        <os family="windows"/>
    </condition>
    <condition property="isLinux">
        <os name="Linux"/>
    </condition>
</target>

<target name="winBrowserDef" depends="checkOS" if="isWindows">
    <macrodef name="browse">
        <attribute name="file"/>
        <sequential>
            <exec executable="rundll32.exe">
                <arg line="url.dll, FileProtocolHandler"/>
                <arg file="@{file}"/>
            </exec>
        </sequential>
    </macrodef>
</target>

<target name="linBrowserDef" depends="checkOS" if="isLinux">
    <macrodef name="browse">
        <attribute name="file"/>
        <sequential>
            <exec executable="htmlview" spawn="yes">
                <arg file="@{file}"/>
            </exec>
        </sequential>
    </macrodef>
</target>

<target name="browserDef" depends="winBrowserDef,linBrowserDef"/>
```

2.2. Reuse of the build scripts via <import>

In the example above we saw that after calling target **browserDef** macrodef <browse> is defined for the corresponding operating system. It is natural to wish to use this macrodef in other build scripts.
One of the way to reuse Ant build scripts (or parts of them) is via XML entities. It is not related with Ant, but with the XML format of the build scripts.
The other way is to use task **<import>**. An example of the syntax of this task is:

```
<import file="imported.xml" optional="true"/>,
```

where attribute **file** points another build file, which we want to include in the current build scripts, and attribute **optional** specifies whether the build will stop, if the imported file does not exist.

The build script, containing macrodef <browse>, is imported in the following way:

```
<project name="test-import" default="test-browser" basedir=".">

    <import file="../macrodef/browser.macrodef" optional="false"/>

    <target name="test-browser" depends="browserDef">
        <browse file="test-import.html"/>
    </target>
</project>
```

Target test-browser depends on target browserDef (the latter defined in the imported file), because macrodef <browse> is created after target browserDef is executed. In this way <import> allows the reuse of both macrodefs and targets.
In its foundation it works in the same way the XML entities work, i.e. <import> adds in the importing file everything from the imported without tag <project>. The differences are that <import> target overriding and some special property overriding.

2.3. Executing several build scripts with <subant>

Consider the following situation: let's have a given product, which consists of several projects, and building of the product requires building of the projects. In most cases, every project has its own build script and the product's build script calls the build scripts of the projects.
In the ancient ages before Ant 1.6 this is achieved with task **<ant>**. Here is a piece of pseudocode that implements this idea:

```
<target name="compile-product">
    <ant target="compile" antfile="project1/build.xml"/>
    <ant target="compile" antfile="project2/build.xml"/>
    ...
    <ant target="compile" antfile="projectN/build.xml"/>
</target>
```

A convenient replacement of task <ant> is task <subant>, which looks in the following way:

```
<target name="compile-product">
    <subant target="compile">
        <dirset dir="${basedir}">
            <include name="project*"/>
        </dirset>
    </subant>
</target>
```

3. Demonstration of a project using <subant>, <import> and <macrodef>

4. <subant>, <import> <macrodef>

The project, which will illustrate the ideas above is called **antutils**. It implements some Ant extensions, s Apache Ant, separated in components, that are:

- FileInfo - an Ant task, that extracts information about a file - its length, if it is hidden, etc.;
- JazzyTask - an Ant task, that spell checks files, using Java library Jazzy;
- Input - this component allows masking of passwords entered in a text console;
- MailAttachLogger - an Ant listener, that extends the original MailLogger and allows to attach files to the message with the build result.

Each of these components should be compiled, tested and packaged. These three operations can be isolated in one common build script - `common.xml`, that can be ¡import¿ed in the build script of each component. If necessary the build script of a given component may predefine a property or a task from common.xml. We have also one main build script `build.xml`, that uses ¡subant¿ to call the build scripts of the components.
The example with antutils shows the case, when the componets are independent of each other. The same technique, however, can be applied with some modifications when building projects, that depend on each other.

5. How to access the current presentation and the sample code

You can download the presentation as an archive called `OFP2005.tar.bz2` from

http://sourceforge.net/project/showfiles.php?group_id=103509

Its latest version can be downloaded from the CVS repository of SF.net via the commands:

```
cvs -d:pserver:anonymous@cvs.sourceforge.net:/cvsroot/emaria login
cvs -d:pserver:anonymous@cvs.sourceforge.net:/cvsroot/emaria co -P \
OFP2005
```

The source code of **`antutils`** project can be downloaded again from the CVS repository of SF.net via the commands:

```
cvs -d:pserver:anonymous@cvs.sourceforge.net:/cvsroot/emaria login
cvs -d:pserver:anonymous@cvs.sourceforge.net:/cvsroot/emaria co -P
antutils
```

When a password is requested for the CVS server, give an empty one, i.e. just press Enter.