How To:

Build and Run MapReduce Projects using ANT

Ant is a tool to help configure a project to automatically be compiled, built and even run. All that is required is for ant to be installed on your machine (which it should already be for linux machines in the lab), and an xml file ('build.xml') to be located at the root of your project.
For the COMP38120 projects, a build script has been supplied to you so you can get started straight away.

How to run an ANT file

One of the great things about ANT is how easy it is to call when your build.xml file is configured. From the root of your project, simply enter:

```
$ ant
```

in the terminal to run the default target method, which is usually 'main', but can be specified in the 'project' tag. Ideally, you want your default target to contain the process that is the most important, quickest and run the most often. IN the COMP38120 files, the main process cleans up old files, compiles the java files into classes and builds those classes into a jar. If you want to run a different target method, for example 'run', which would run a Cloud10 process in hadoop local mode, you can call that by entering:

```
$ ant run
```

in your terminal. Methods can call other methods they depend on too (which is good if you need to compile and build your jar file before running it)!
Finally, you may wish to know more about what your ant process is doing. In order to launch ant in verbose mode, you can tag it with '-v'. For example:

```
$ ant run -v
```

The main components of a build.xml

There are many components of an ant build script. In this step, we discuss all of the components you will require for when you wish to run your own MapReduce job.

1. Project

```
<project name="Cloud10" basedir="." default="main">
     ...
</project>
```

The project tag is the first tag that should be in any build.xml file. It gives the project its global name, the base (or root) directory of the project relative to this file, and specifies the default tag that is run when ant is run without any arguments.

2. Target

```
<target name=”clean”>
      ...
</target>

<target name=”compile”>
      ...
</target>

<target name=”jar” depends=”compile”>
      ...
</target>

<target name=”run” depends=”main”>
      ...
</target>

<target name=”main” depends=”clean,jar” />
```

Targets are very like methods or functions used in regular programming. They have names, and when they are called, they run a procedure. They can also run other targets, but these must run before the current target. Targets can also be called by running ant with the target name as an argument from the command line, each target serving as an entry point to the procedure.
In the above example, you can see a set of targets. Each of these targets represents a set of steps to achieve an outcome. For example, running the ‘clean’ target should clean up the project directory, whereas ‘jar’ should compile all the code, and then build it into a jar file.
Targets can have a set of dependencies, stated in the ‘depends’ attribute. For example, in order to create the jar file, the code has to be compiled first. ‘Run’ depends on ‘main’ running first, which in turn depends on ‘clean’, then ‘jar’, which needs to compile first, after which the program can be executed. Of course, if you could have various different run targets (each named something unique) that could have the same dependencies and do something different (such as run a different main class in your jar file).

3. Property

```
<property name="build.dir"   value="build" />
<property name="classes.dir" value="${build.dir}/classes" />
```

Properties are much like variables in regular coding and are usually set at the top of a build file (although they can be set wherever you want to use them, for example, a property is set in the target ‘run’).
Properties are referred to as above by wrapping the property name in the `${name}` syntax, and can be nested in other properties.

4. Path

```
<path id="classpath">
      <fileset dir="${lib.dir}" includes="*.jar"/>
</path>

<path id="run.path.id">
      <path refid="${classpath}" />
      <fileset dir="${jar.dir}" />
</path>
```

Paths are similar to properties except they contain a set of paths to given files. These files can be included or excluded given a regular expression. For the above ‘classpath’ example, this specifies only the files that end in jar contained in the lib directory (evaluated from a property above). In the ‘run.path.id’ example, it combines the set of files in classpath with all the files located in the ‘build/jar’ location (which contains the built jar file). It is easy to see the advantages of build scripts from this example as any new libraries added to the directory will become part of the build.

```
<pathconvert property="run.path.id" refid="run.path.id" />
```

In the ‘run’ target, the ‘pathconvert’ tag is used to convert the path ‘run.path.id’ into a property of the same name. The property can be used like any other property by using the `${name}` statement.

MapReduce build.xml Targets

Rather than introducing each and every possible component of a target, which are plentiful, each of the current targets shall now be discussed. The build.xml file can be found in the appendix.

Clean

The clean target is intended to remove old output files. This is an easy solution to most build issues. This ensures that only new files are run. The method to doing this is very simple: using the `<delete>` component, remove each folder that contains output.

Compile

The compile target is substantially more complicated than the ‘Clean’ target, as you can see.

1. `<mkdir>` Creates a directory in which to put the compiled classes

2. *<javac>* Compiles all of the specified java files. It takes all of the java files from the src directory, compiles them into the build/classes directory with the classpath specified in the classpath path.
3. Finally, *<copy>* and *<fileset>* copies all the non-java files into the build/classes directory. This could be input files, libraries or property files.

Jar

A Jar file must be created from the compiled classes for use with Hadoop (both locally or on Amazon). This target depends on the 'compile' target, as the compiled classes are required in order to build a jar file.

1. Again, *<mkdir>* creates an output directory for the jar file.
2. *<jar>* takes all the classes in the build/compile directory and builds them into build/jar/Cloud10.jar.
3. The *<manifest>* component specifies the main class of this jar file.

Once these steps have run, there is a jar file ready to go.

Javadoc

This target builds a javadoc from the files in the src directory. The *<include>* and *<exclude>* tags inside the *<fileset>* tag specify which files are to be used to build the javadoc (regular java files are to be included, but any test files are not to be). There are also a set of options set in the *<javadoc>* tag.

Main

The main target has no steps! All this target does is depend on other targets in order to specify a main entry point to the build.

Run

The run target is the most complicated, interesting and non-standard of the build targets. This is because a hadoop jar has to be run in the hadoop environment. The easiest way to do this in ant is to echo out a script to run the program, and then run the script:

1. *<pathconvert>* creates properties from the paths.
2. *<mkdir>* creates a directory for the script to end up in.
3. The two *<echo>* lines echo the script into the etc/run.sh file. The second *<echo>* component has append set to true so as not to overwrite the first line.
4. *<chmod>* sets the permissions on run.sh so that the file can be run, both by the user and by the ant script.
5. Finally, the script gets executed with the *<exec>* component and *<arg>* sets any command line arguments. For these Jobs '-input' and '-output' must be defined. These directories can be set at the top of the file.

Components that were not used in the supplied build.xml (but you might want to use)

There are a few additional components that should be mentioned as they occur frequently in other build scripts. They are by no means required, but you may feel the need to include them either now or in the future.

<get>

```
    <get src="[file stored on the web]" dest="[output folder]" />
```

The <get> component allows files to be downloaded and placed into a directory. This can be very useful for getting required libraries so you don't have to store them if you, for example, put your code into a repository.

<junit>

The `<junit>` and `<junitreport>` tags can be used to run junit tests and generate reports. A link to more information about the `<junit>` tags can be found at the end of this document.

<java>

Java files can be natively run using the <java> tag. This is the regular way to run java programs, but wasn't used due to the limitations of the hadoop environment. As with the <junit> tag, a link to more information can be found at the end of the document.

Additional Reading

- A quick and simple tutorial to composing an ant file: http://ant.apache.org/manual/tutorial-HelloWorldWithAnt.html
- More information on the junit tag http://ant.apache.org/manual/Tasks/junit.html
- More information on the Java tag http://ant.apache.org/manual/Tasks/java.html

Build.xml for WordCount

```
<project name="COMP38120-Lab1.1" basedir="." default="main">

       <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" />
       <property name="script.dir"  value="etc" />
       <property name="javadoc.dir" value="javadoc" />

       <property name="input.dir"  value="input" />
       <property name="output.dir"  value="output" />

       <property name="main-class"
value="uk.ac.man.cs.comp38120.exercise.WordCount" />
```

```
        <property                                              name="hadoop-location"
value="/Users/norm/Desktop/Workfiles/teach/COMP38120/Kristian/ModifiedHadoop/hadoop-
1.0.3/bin/" />

    <path id="classpath">
        <fileset dir="${lib.dir}" includes="*.jar" excludes="*.txt" />
    </path>

        <path id="lib.path.id">
                <fileset dir="${lib.dir}" />
        </path>

        <path id="run.path.id">
                <path refid="lib.path.id" />
                <fileset dir="${jar.dir}" />
        </path>

        <path id="jar.id">
                <fileset dir="${jar.dir}" />
        </path>

        <path id="application" location="${jar.dir}/${ant.project.name}.jar" />

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

    <target name="compile">
        <mkdir dir="${classes.dir}" />
        <javac  includeantruntime="false"  srcdir="${src.dir}"  destdir="${classes.dir}"
classpathref="classpath" />
        <copy todir="${classes.dir}">
                <fileset dir="${src.dir}" excludes="*.java" />
        </copy>
    </target>

        <target name="jar" depends="compile">
                <mkdir dir="${jar.dir}" />
                <mkdir dir="${classes.dir}/${lib.dir}" />
                <copy todir="${classes.dir}/${lib.dir}">
                                <path refid="lib.path.id" />
                </copy>

                <path id="jar.lib.path.id">
                        <fileset dir="${classes.dir}/${lib.dir}" includes="*.jar" />
                </path>

                <manifestclasspath                         property="manifest.classpath"
jarfile="${jar.dir}/${ant.project.name}.jar">
                        <classpath refid="jar.lib.path.id" />
                </manifestclasspath>
```

```
             <jar                        destfile="${jar.dir}/${ant.project.name}.jar"
basedir="${classes.dir}">
                    <manifest>
                          <attribute name="Main-Class" value="${main-class}" />
                          <attribute  name="Class-Path"  value="${manifest.classpath}"
/>
                    </manifest>
             </jar>
       </target>

       <target name="javadoc" depends="jar">
           <javadoc destdir="${javadoc.dir}" access="public" use="true" notree="false"
nonavbar="false"   noindex="false"   splitindex="no"   author="true"   version="true"
nodeprecatedlist="false" nodeprecated="false" classpathref="lib.path.id">
               <fileset dir="${src.dir}">
                    <include name="*.java" />
                    <exclude name="*Test.java" />
               </fileset>
               <link href="http://download.oracle.com/javase/6/docs/api/" />
               <link href="http://hadoop.apache.org/docs/r1.0.4/api/" />
           </javadoc>
       </target>

    <target name="run" depends="main">

       <pathconvert property="run.path.id" refid="run.path.id" />
       <pathconvert property="lib.path.id" refid="lib.path.id" />
       <pathconvert property="jar.id" refid="jar.id" />

       <mkdir dir="${script.dir}" />
       <echo file="${script.dir}/run.sh" message="#!/bin/bash${line.separator}" />
       <echo    file="${script.dir}/run.sh"    message="${hadoop-location}hadoop    jar
${jar.id} -D mapreduce.framework.name=local -D mapreduce.jobtracker.address=local -D
fs.default.name=file:///    -D    mapreduce.cluster.local.dir=/tmp/mapred/local    -D
mapreduce.cluster.temp.dir=/tmp/mapred/temp                                        -D
mapreduce.jobtracker.staging.root.dir=/tmp/mapred/staging                          -D
mapreduce.jobtracker.system.dir=/tmp/mapred/system  $1  $2  $3  $4  $5  $6  $7  $8  $9"
append="true" />
       <chmod dir="${script.dir}" perm="ugo+rx" includes="*.sh" />

       <exec executable="${script.dir}/run.sh">
             <arg value="-input ${input.dir}" />
             <arg value="-output ${output.dir}" />
       </exec>

    </target>

       <target name="main" depends="clean,jar" />

</project>
```