APACHE ANT

project build management

tutorialspoint

SIMPLYEASYLEARNING

www.tutorialspoint.com

https://www.facebook.com/tutorialspointindia

https://twitter.com/tutorialspoint

About the Tutorial

Apache ANT is a Java based build tool from Apache Software Foundation. Apache ANT's build files are written in XML and they take advantage of being open standard, portable and easy to understand.

This tutorial will teach you how to use Apache ANT to automate the build and deployment process in simple and easy steps. After completing this tutorial, you will find yourself at a moderate level of expertise in using Apache ANT from where, you can take yourself to next levels.

Audience

This tutorial is prepared for the beginners to help them understand basic functionality of Apache ANT tool to automate the build and deployment process.

Prerequisites

We assume that you have knowledge about the software development using any programming language, especially Java, and are aware about the software build and deployment process.

Copyright & Disclaimer

Table of Contents

1. Apache ANT – Overview

ANT stands for **Another Neat Tool**. It is a Java-based build tool from computer software development company Apache. Before going into the details of Apache Ant, let us first understand why we need a build tool.

Need for a Build Tool

On an average, a developer spends a substantial amount of time doing mundane tasks like build and deployment that include:

- Compiling the code
- Packaging the binaries
- Deploying the binaries to the test server
- Testing the changes
- Copying the code from one location to another

To automate and simplify the above tasks, Apache Ant is useful. It is an Operating System build and deployment tool that can be executed from the command line.

History of Apache Ant

Ant was created by software developer James Duncan Davidson who is also the original creator of webserver application Tomcat.

Ant was originally used to build Tomcat, and was bundled as a part of Tomcat distribution.

It was born out of the problems and complexities associated with the Apache Make tool.

It was promoted as an independent project in Apache in the year 2000. The latest version of Apache Ant as on September 2020 is **1.10.9**.

Features of Apache Ant

The features of Apache Ant are listed below:

- It is the most complete Java build and deployment tool available.
- It is platform neutral and can handle platform specific properties, such as file separators.
- It can be used to perform platform specific tasks such as modifying the modified time of a file using 'touch' command.
- Ant scripts are written using plain XML. If you are already familiar with XML, you can learn Ant pretty quickly.

- Ant is good at automating complicated repetitive tasks.
- Ant comes with a big list of predefined tasks.
- Ant provides an interface to develop custom tasks.
- Ant can be easily invoked from the command line and it can integrate with free and commercial IDEs.

2. Apache ANT — Environment Setup

Apache Ant is distributed under the Apache Software License which is a fully-fledged open source license certified by the open source initiative.

The latest Apache Ant version, including its full-source code, class files, and documentation can be found at http://ant.apache.org.

Installing Apache Ant

It is assumed that you have already downloaded and installed Java Development Kit (JDK) on your computer. If not, please follow the instructions available at file:///C:/java/java_environment_setup.htm

- Ensure that the JAVA_HOME environment variable is set to the folder, where your JDK is installed.
- Download the binaries from http://ant.apache.org
- Unzip the zip file to a convenient location c:\folder by using Winzip, winRAR, 7-zip or similar tools.
- Create a new environment variable called **ANT_HOME** that points to the Ant installation folder. In this case, it is **c:\apache-ant-1.10.9-bin** folder.
- Append the path to the Apache Ant batch file to the PATH environment variable. In our case, this would be the **c:\apache-ant-1.10.9-bin\bin** folder.

Verifying the Installation

To verify the successful installation of Apache Ant on your computer, type ant on your command prompt.

You should see an output as given below:

```
C:\>ant -version

Apache Ant(TM) version 1.10.9 compiled on September 27 2020
```

If you do not see the above output, then please verify that you have followed the installation steps properly.

Installing Eclipse

This tutorial also covers integration of Ant with Eclipse integrated development environment (IDE). Hence, if you have not installed Eclipse, please download and install Eclipse.

Steps to install Eclipse

- Download the latest Eclipse binaries from www.eclipse.org
- Unzip the Eclipse binaries to a convenient location, say c:\folder.
- Run Eclipse from c:\eclipse\eclipse.exe.

3. Apache ANT — Build Files

Typically, Ant's build file, called **build.xml** should reside in the base directory of the project. However, there is no restriction on the file name or its location. You are free to use other file names or save the build file in some other location.

For this exercise, create a file called build.xml anywhere in your computer with the following contents:

```
<?xml version="1.0"?>
   <project name="Hello World Project" default="info">
   <target name="info">
      <echo>Hello World - Welcome to Apache Ant!</echo>
   </target>
</project>
```

Note that there should be no blank line(s) or whitespace(s) before the xml declaration. If you allow them, the following error message occurs while executing the ant build -

```
The processing instruction target matching "[xX][mM][lL]" is not allowed. All
build files require the project element and at least one target element.
```

The XML element **project** has three attributes which are as follows:

Attributes	Description
name	The Name of the project. (Optional)
default	The default target for the build script. A project may contain any number of targets. This attribute specifies which target should be considered as the default. (Mandatory)
basedir	The base directory (or) the root folder for the project. (Optional)

A target is a collection of tasks that you want to run as one unit. In our example, we have a simple target to provide an informational message to the user.

Targets can have dependencies on other targets. For example, a **deploy** target may have a dependency on the **package** target, the **package** target may have a dependency on the **compile** target and so forth. Dependencies are denoted using the **depends** attribute.

For example:

```
<target name="deploy" depends="package">

  ....
```

```
</target>
<target name="package" depends="clean,compile">
  ....
</target>
<target name="clean" >
  ....
</target>
<target name="compile" >
  ....
</target>
```

The target element has the following attributes:

Attributes	Description
name	The name of the target (Required)
depends	Comma separated list of all targets that this target depends on. (Optional)
description	A short description of the target. (optional)
if	Allows the execution of a target based on the trueness of a conditional attribute. (optional)
unless	Adds the target to the dependency list of the specified Extension Point. An Extension Point is similar to a target, but it does not have any tasks. (Optional)

The **echo** task in the above example is a trivial task that prints a message. In our example, it prints the message **Hello World.**

To run the ant build file, open up command prompt and navigate to the folder, where the build.xml resides, and then type **ant info**. You could also type **ant** instead. Both will work, because **info** is the default target in the build file.

You should see the following output:

```
C:\>ant
Buildfile: C:\build.xml


info: [echo] Hello World - Welcome to Apache Ant!


BUILD SUCCESSFUL
Total time: 0 seconds


C:\>
```

4. Apache ANT — Property Task

Ant build files are written in XML, which does not allow declaring variables as you do in your favorite programming language. However, as you may have imagined, it would be useful if Ant allowed declaring variables such as project name, project source directory, etc.

Ant uses the **property** element which allows you to specify the properties. This allows the properties to be changed from one build to another or from one environment to another.

Ant Properties

By default, Ant provides the following pre-defined properties that can be used in the build file:

Properties	Description
ant.file	The full location of the build file.
ant.version	The version of the Apache Ant installation.
basedir	The basedir of the build, as specified in the basedir attribute of the project element.
ant.java.version	The version of the JDK that is used by Ant.
ant.project.name	The name of the project, as specified in the name attribute of the project element.
ant.project.default-target	The default target of the current project.
ant.project.invoked-targets	Comma separated list of the targets that were invoked in the current project.
ant.core.lib	The full location of the Ant jar file.
ant.home	The home directory of Ant installation.
ant.library.dir	The home directory for Ant library files - typically ANT_HOME/lib folder.

Ant also makes the system properties (Example: file.separator) available to the build file.

In addition to the above, the user can define additional properties using the **property** element.

The following example shows how to define a property called **sitename**:

```
<?xml version="1.0"?>
```

```
<project name="Hello World Project" default="info">

   <property name="sitename" value="www.tutorialspoint.com"/>
   <target name="info">
      <echo>Apache Ant version is ${ant.version} - You are
         at ${sitename} </echo>
   </target>
</project>
```

Running Ant on the above build file produces the following output:

```
C:\>ant
Buildfile: C:\build.xml

info: [echo] Apache Ant version is Apache Ant(TM) version 1.10.9
      compiled on September 27 2020 - You are at www.tutorialspoint.com

BUILD SUCCESSFUL
Total time: 0 seconds
C:\>
```

5. Apache ANT — Property Files

Setting properties directly in the build file is fine, if you are working with a handful of properties. However, for a large project, it makes sense to store the properties in a separate property file.

Benefits

Storing the properties in a separate file offers the following benefits:

- It allows you to reuse the same build file, with different property settings for different execution environment. For example, build properties file can be maintained separately for DEV, TEST, and PROD environments.
- It is useful, when you do not know the values for a property (in a particular environment) up-front. This allows you to perform the build in other environments, where the property value is known.

There is no hard and fast rule, but typically the property file is named as **build.properties** and is placed along-side the **build.xml** file. You could create multiple build properties files based on the deployment environments - such as **build.properties.dev** and **build.properties.test.**

The contents of the build property file are similar to the normal java property file. They contain one property per line. Each property is represented by a name and a value pair.

The name and value pairs are separated by an equals (=) sign. It is highly recommended that the properties are annotated with proper comments. Comments are listed using the hash (#) character.

The following example shows a **build.xml** file and its associated **build.properties** file:

build.xml

Given below is an example for build.xml file.

```
<?xml version="1.0"?>
<project name="Hello World Project" default="info">
   <property file="build.properties"/>
      <target name="info">
         <echo>Apache Ant version is ${ant.version} - You are
            at ${sitename} </echo>
      </target>
</project>
```

build.properties

An example for build.properties file is mentioned below:

```
# The Site Name
sitename=www.tutorialspoint.com
buildversion=3.3.2
```

In the above example, **sitename** is a custom property which is mapped to the website name. You can declare any number of custom properties in this fashion.

Another custom property listed in the above example is the **buildversion**, which, in this instance, refers to the version of the build.

In addition to the above, Ant comes with a number of predefined build properties, which are listed in the previous section, but is given below once again for your reference.

Properties	Description
ant.file	The full location of the build file.
ant.version	The version of the Apache Ant installation.
basedir	The basedir of the build, as specified in the basedir attribute of the project element.
ant.java.version	The version of the JDK that is used by Ant.
ant.project.name	The name of the project, as specified in the name attribute of the project element.
ant.project.default-target	The default target of the current project.
ant.project.invoked-targets	Comma separated list of the targets that were invoked in the current project.
ant.core.lib	The full location of the Ant jar file.
ant.home	The home directory of Ant installation.
ant.library.dir	The home directory for Ant library files - typically ANT_HOME/lib folder.

The example presented in this chapter uses the **ant.version** built-in property.

6. Apache ANT — Data Types

Ant provides a number of predefined data types. Do not confuse the term "data types" with those that are available in the programming language. Instead, consider them as a set of services that are built into the product already.

Data Types in Ant

The following data types are provided by Apache Ant.

Fileset

The fileset data type represents a collection of files. It is used as a filter to include or exclude files that match a particular pattern.

For example, refer the following code. Here, the src attribute points to the source folder of the project.

```
<fileset dir="${src}" casesensitive="yes">
   <include name="**/*.java"/>
   <exclude name="**/*Stub*"/>
</fileset>
```

The fileset selects all .java files in the source folder except those contain the word 'Stub'. The case-sensitive filter is applied to the fileset, which means a file with the name Samplestub.java will not be excluded from the fileset.

Pattern set

A pattern set is a pattern that allows to filter files or folders easily based on certain patterns. The patterns can be created using the following meta characters:

- **?** - Matches one character only.
- - Matches zero or many characters.
- ****** - Matches zero or many directories recursively.

The following example depicts the usage of a pattern set.

```
<patternset id="java.files.without.stubs">
   <include name="src/**/*.java"/>
   <exclude name="src/**/*Stub*"/>
</patternset>
```

The patternset can then be reused with a fileset as follows:

```
<fileset dir="${src}" casesensitive="yes">
```

```
   <patternset refid="java.files.without.stubs"/>
</fileset>
```

File list

The filelist data type is similar to the file set except the following differences:

- It contains explicitly named lists of files and it does not support wild cards.
- This data type can be applied for existing or non-existing files.

Let us see the following example of the filelist data type. Here, the attribute **webapp.src.folder** points to the web application source folder of the project.

```
<filelist id="config.files" dir="${webapp.src.folder}">
   <file name="applicationConfig.xml"/>
   <file name="faces-config.xml"/>
   <file name="web.xml"/>
   <file name="portlet.xml"/>
</filelist>
```

Filter set

By using a filterset data type along with the copy task, you can replace certain text in all the files that matches the pattern with a replacement value.

A common example is to append the version number to the release notes file, as shown in the following code.

```
<copy todir="${output.dir}">
   <fileset dir="${releasenotes.dir}" includes="**/*.txt"/>
   <filterset>
      <filter token="VERSION" value="${current.version}"/>
   </filterset>
</copy>
```

In the above mentioned code:

- The attribute **output.dir** points to the output folder of the project.
- The attribute **releasenotes.dir** points to the release notes folder of the project.
- The attribute **current.version** points to the current version folder of the project.
- The copy task, as the name suggests, is used to copy files from one location to another.

Path

The **path** data type is commonly used to represent a class-path. Entries in the path are separated using semicolons or colons. However, these characters are replaced at the run-time by the executing system's path separator character.

The classpath is set to the list of jar files and classes in the project, as shown in the example below.

```
<path id="build.classpath.jar">
   <pathelement path="${env.J2EE_HOME}/${j2ee.jar}"/>
   <fileset dir="lib">
      <include name="**/*.jar"/>
   </fileset>
</path>
```

In the code given above:

- The attribute **env.J2EE_HOME** points to the environment variable **J2EE_HOME**.
- The attribute **j2ee.jar** points to the name of the J2EE jar file in the J2EE base folder.

7. Apache ANT — Building Projects

Now that we have learnt about the data types in Ant, it is time to put that knowledge into practice. We will build a project in this chapter. The aim of this chapter is to build an Ant file that compiles the java classes and places them in the WEB-INF\classes folder.

Consider the following project structure:

- The database scripts are stored in the **db** folder.
- The java source code is stored in the **src** folder.
- The images, js, META-INF, styles (css) are stored in the **war** folder.
- The Java Server Pages (JSPs) are stored in the **jsp** folder.
- The third party jar files are stored in the **lib** folder.
- The java class files are stored in the **WEB-INF\classes** folder.

This project forms the **Hello World** Fax Application for the rest of this tutorial.

```
C:\work\FaxWebApplication>tree
Folder PATH listing
Volume serial number is 00740061 EC1C:ADB1
C:.
+---db
+---src
.   +---faxapp
.   +---dao
.   +---entity
.   +---util
.   +---web
+---war
    +---images
    +---js
    +---META-INF
    +---styles
    +---WEB-INF
        +---classes
        +---jsp
        +---lib
```

Here is the **build.xml** required for this project. Let us consider it piece by piece.

```
<?xml version="1.0"?>
<project name="fax" basedir="." default="build">
   <property name="src.dir" value="src"/>
   <property name="web.dir" value="war"/>
   <property name="build.dir" value="${web.dir}/WEB-INF/classes"/>
   <property name="name" value="fax"/>

   <path id="master-classpath">
      <fileset dir="${web.dir}/WEB-INF/lib">
         <include name="*.jar"/>
      </fileset>
      <pathelement path="${build.dir}"/>
   </path>

   <target name="build" description="Compile source tree java files">
      <mkdir dir="${build.dir}"/>
      <javac destdir="${build.dir}" source="1.5" target="1.5">
         <src path="${src.dir}"/>
         <classpath refid="master-classpath"/>
      </javac>
   </target>

   <target name="clean" description="Clean output directories">
      <delete>
         <fileset dir="${build.dir}">
            <include name="**/*.class"/>
         </fileset>
      </delete>
   </target>
</project>
```

First, let us declare some properties for the source, web, and build folders.

```
<property name="src.dir" value="src"/>
<property name="web.dir" value="war"/>
<property name="build.dir" value="${web.dir}/WEB-INF/classes"/>
```

In the above mentioned example:

- **src.dir** refers to the source folder of the project, where the java source files can be found.
- **web.dir** refers to the web source folder of the project, where you can find the JSPs, web.xml, css, javascript and other web related files
- **build.dir** refers to the output folder of the project compilation.

Properties can refer to other properties. As shown in the above example, the **build.dir** property makes a reference to the **web.dir** property.

In this example, the **src.dir** refers to the source folder of the project.

The default target of our project is the **compile** target. But first, let us look at the **clean** target.

The clean target, as the name suggests, deletes the files in the build folder.

```
<target name="clean" description="Clean output directories">
   <delete>
      <fileset dir="${build.dir}">
         <include name="**/*.class"/>
      </fileset>
   </delete>
</target>
```

The master-classpath holds the classpath information. In this case, it includes the classes in the build folder and the jar files in the lib folder.

```
<path id="master-classpath">
   <fileset dir="${web.dir}/WEB-INF/lib">
      <include name="*.jar"/>
   </fileset>
   <pathelement path="${build.dir}"/>
</path>
```

Finally, the build targets to build the files.

First of all, we create the build directory, if it does not exist, then, we execute the javac command (specifying jdk1.5 as our target compilation). We supply the source folder and the classpath to the javac task and ask it to drop the class files in the build folder.

```
<target name="build" description="Compile main source tree java files">
   <mkdir dir="${build.dir}"/>
   <javac destdir="${build.dir}" source="1.5" target="1.5" debug="true"
      deprecation="false" optimize="false" failonerror="true">
```

```
      <src path="${src.dir}"/>
      <classpath refid="master-classpath"/>
   </javac>
</target>
```

Executing Ant on this file compiles the java source files and places the classes in the build folder.

The following outcome is the result of running the Ant file:

```
C:\>ant
Buildfile: C:\build.xml

BUILD SUCCESSFUL
Total time: 6.3 seconds
```

The files are compiled and placed in the **build.dir** folder.

8. Apache ANT — Build Documentation

Documentation is a must in any project. Documentation plays a great role in the maintenance of a project. Java makes documentation easier by the use of the in-built **javadoc** tool. Ant makes it even easier by generating the documentation on demand.

As you know, the javadoc tool is highly flexible and allows a number of configuration options. Ant exposes these configuration options via the javadoc task. If you are unfamiliar with javadocs, we suggest that you start with this Java Documentation Tutorial.

The following section lists the most commonly used javadoc options that are used in Ant.

Attributes

Source can be specified using **sourcepath**, **sourcepathref** or **sourcefiles**.

- **sourcepath** is used to point to the folder of the source files (e.g. src folder).
- **sourcepathref** is used to refer a path that is referenced by the path attribute (e.g, delegates.src.dir).
- **sourcefiles** is used when you want to specify the individual files as a comma separated list.

Destination path is specified using the **destdir** folder (e.g build.dir).

You could filter the **javadoc** task by specifying the package names which are to be included. This is achieved by using the **packagenames** attribute, a comma separated list of package files.

You could filter the javadoc process to show only the public, private, package, or protected classes and members. This is achieved by using the **private**, **public**, **package** and **protected** attributes.

You could also tell the javadoc task to include the author and version information by using the respective attributes.

You could also group the packages together using the **group** attribute, so that it becomes easy to navigate.

Putting it all together

Let us continue our theme of the **Hello world** Fax application and add a documentation target to our Fax application project.

Given below is an example javadoc task used in our project. In this example, we have specified the javadoc to use the **src.dir** as the source directory, and **doc** as the target.

We have also customised the window title, the header, and the footer information that appear on the java documentation pages.

Also, we have created three groups:

- one for the utility classes in our source folder,
- one for the user interfaces classes, and
- one for the database related classes.

You may notice that the data package group has two packages -- faxapp.entity and faxapp.dao.

```
<target name="generate-javadoc">
   <javadoc packagenames="faxapp.*" sourcepath="${src.dir}"
      destdir="doc" version="true" windowtitle="Fax Application">
      <doctitle><![CDATA[= Fax Application =]]></doctitle>
      <bottom>
         <![CDATA[Copyright © 2011. All Rights Reserved.]]>
      </bottom>
      <group title="util packages" packages="faxapp.util.*"/>
      <group title="web packages" packages="faxapp.web.*"/>
      <group title="data packages"
         packages="faxapp.entity.*:faxapp.dao.*"/>
   </javadoc>
   <echo message="java doc has been generated!" />
</target>
```

Let us execute the javadoc Ant task. It generates and places the java documentation files in the doc folder.

When the **javadoc target** is executed, it produces the following outcome:

```
C:\>ant generate-javadoc
Buildfile: C:\build.xml

java doc has been generated!

BUILD SUCCESSFUL
Total time: 10.63 second
```

The java documentation files are now present in the **doc** folder.

Typically, the javadoc files are generated as a part of the release or package targets.

9. Apache ANT — Creating JAR files

The next logical step after compiling your java source files, is to build the java archive, i.e., the Java Archive (JAR) file. Creating JAR files with Ant is quite easy with the **jar** task.

Attributes

The commonly used attributes of the jar task are as follows:

Attributes	Description
basedir	The base directory for the output JAR file. By default, this is set to the base directory of the project.
compress	Advises Ant to compress the file as it creates the JAR file.
keepcompression	While the compress attribute is applicable to the individual files, the keepcompression attribute does the same thing, but it applies to the entire archive.
destfile	The name of the output JAR file.
duplicate	Advises Ant on what to do when duplicate files are found. You could add, preserve, or fail the duplicate files.
excludes	Advises Ant to not include these comma separated list of files in the package.
excludesfile	Same as above, except the exclude files are specified using a pattern.
inlcudes	Inverse of excludes.
includesfile	Inverse of excludesfile.
update	Advises Ant to overwrite files in the already built JAR file.

Continuing our **Hello World** Fax Application project, let us add a new target to produce the jar files.

But before that, let us consider the jar task given below.

```
<jar destfile="${web.dir}/lib/util.jar"
   basedir="${build.dir}/classes"
```

```
   includes="faxapp/util/**"
   excludes="**/Test.class"
/>
```

Here, the **web.dir** property points to the path of the web source files. In our case, this is where the util.jar will be placed.

The **build.dir** property in this example, points to the build folder, where the class files for the util.jar can be found.

In this example, we create a jar file called **util.jar** using the classes from the **faxapp.util.*** package. However, we are excluding the classes that end with the name Test. The output jar file will be placed in the web application lib folder.

If we want to make the util.jar an executable jar file, we need to add the **manifest** with the **Main-Class** meta attribute.

Therefore, the above example will be updated as follows:

```
<jar destfile="${web.dir}/lib/util.jar"
   basedir="${build.dir}/classes"
   includes="faxapp/util/**"
   excludes="**/Test.class">
   <manifest>
      <attribute name="Main-Class" value="com.tutorialspoint.util.FaxUtil"/>
   </manifest>
</jar>
```

To execute the jar task, wrap it inside a target, most commonly, the build or package target, and execute them.

```
<target name="build-jar">
<jar destfile="${web.dir}/lib/util.jar"
   basedir="${build.dir}/classes"
   includes="faxapp/util/**"
   excludes="**/Test.class">
   <manifest>
      <attribute name="Main-Class" value="com.tutorialspoint.util.FaxUtil"/>
   </manifest>
</jar>
</target>
```

Running Ant on this file creates the util.jar file for us.

The following outcome is the result of running the Ant file:

```
C:\>ant build-jar
Buildfile: C:\build.xml

BUILD SUCCESSFUL
Total time: 1.3 seconds
```

The util.jar file is now placed in the output folder.

10. Apache ANT — Create WAR Files

Creating Web Archive (WAR) files with Ant is extremely simple, and very similar to the creating JAR files task. After all, WAR file, like JAR file is just another ZIP file.

The WAR task is an extension to the JAR task, but it has some nice additions to manipulate what goes into the WEB-INF/classes folder, and generating the web.xml file. The WAR task is useful to specify a particular layout of the WAR file.

Since, the WAR task is an extension of the JAR task, all attributes of the JAR task apply to the WAR task.

Attributes	Description
webxml	Path to the web.xml file.
lib	A grouping to specify what goes into the WEB-INF\lib folder.
classes	A grouping to specify what goes into the WEB-INF\classes folder.
metainf	Specifies the instructions for generating the MANIFEST.MF file.

Continuing our **Hello World** Fax Application project, let us add a new target to produce the jar files. But before that, let us consider the war task.

Consider the following example:

```
<war destfile="fax.war" webxml="${web.dir}/web.xml">
   <fileset dir="${web.dir}/WebContent">
      <include name="**/*.*"/>
   </fileset>
   <lib dir="thirdpartyjars">
      <exclude name="portlet.jar"/>
   </lib>
   <classes dir="${build.dir}/web"/>
</war>
```

As per the previous examples, the **web.dir** variable refers to the source web folder, i.e., the folder that contains the JSP, css, javascript files etc.

The **build.dir** variable refers to the output folder. This is where the classes for the WAR package can be found. Typically, the classes will be bundled into the WEB-INF/classes folder of the WAR file.

In this example, we are creating a war file called **fax.war**. The WEB.XML file is obtained from the web source folder. All files from the 'WebContent' folder under web are copied into the WAR file.

The WEB-INF/lib folder is populated with the jar files from the thirdpartyjars folder. However, we are excluding the portlet.jar as this is already present in the application server's lib folder. Finally, we are copying all classes from the build directory's web folder and putting them into the WEB-INF/classes folder.

Wrap the war task inside an Ant target (usually package) and run it. This will create the WAR file in the specified location.

It is entirely possible to nest the classes, lib, metainf and webinf directors, so that they live in scattered folders anywhere in the project structure. But, best practices suggest that your Web project should have the Web Content structure that is similar to the structure of the WAR file. The Fax Application project has its structure outlined using this basic principle.

To execute the war task, wrap it inside a target, most commonly, the build or package target, and run them.

```
<target name="build-war">
   <war destfile="fax.war" webxml="${web.dir}/web.xml">
   <fileset dir="${web.dir}/WebContent">
      <include name="**/*.*"/>
   </fileset>
   <lib dir="thirdpartyjars">
      <exclude name="portlet.jar"/>
   </lib>
   <classes dir="${build.dir}/web"/>
   </war>
</target>
```

Running Ant on this file will create the **fax.war** file for us.

The following outcome is the result of running the Ant file:

```
C:\>ant build-war
Buildfile: C:\build.xml

BUILD SUCCESSFUL
Total time: 12.3 seconds
```

The fax.war file is now placed in the output folder. The contents of the war file will be as mentioned below:

```
fax.war:
   +---jsp             This folder contains the jsp files
   +---css             This folder contains the stylesheet files
```

```
+---js              This folder contains the javascript files

+---images          This folder contains the image files
+---META-INF        This folder contains the Manifest.Mf
+---WEB-INF
   +---classes   This folder contains the compiled classes
   +---lib       Third party libraries and the utility jar files
   WEB.xml       Configuration file that defines the WAR package
```

11. Apache ANT — Packaging Applications

We have learnt the different aspects of Ant using the **Hello World** Fax web application in bits and pieces.

Now, it is time to put everything together to create a full and complete build.xml file. Consider **build.properties** and **build.xml** files as follows:

build.properties

The file is given below for build.properties.

```
deploy.path=c:\tomcat6\webapps
```

build.xml

The build.xml file is as follows:

```
<?xml version="1.0"?>

<project name="fax" basedir="." default="usage">
   <property file="build.properties"/>
   <property name="src.dir" value="src"/>
   <property name="web.dir" value="war"/>
   <property name="javadoc.dir" value="doc"/>
   <property name="build.dir" value="${web.dir}/WEB-INF/classes"/>
   <property name="name" value="fax"/>

   <path id="master-classpath">
      <fileset dir="${web.dir}/WEB-INF/lib">
         <include name="*.jar"/>
      </fileset>
      <pathelement path="${build.dir}"/>
   </path>

   <target name="javadoc">
      <javadoc packagenames="faxapp.*" sourcepath="${src.dir}"
         destdir="doc" version="true" windowtitle="Fax Application">
      <doctitle><![CDATA[<h1>= Fax Application =</h1>]]>
```

```
      </doctitle>
      <bottom><![CDATA[Copyright © 2011. All Rights Reserved.]]>
      </bottom>
      <group title="util packages" packages="faxapp.util.*"/>
      <group title="web packages" packages="faxapp.web.*"/>
      <group title="data packages"
         packages="faxapp.entity.*:faxapp.dao.*"/>
      </javadoc>
   </target>

   <target name="usage">
      <echo message=""/>
      <echo message="${name} build file"/>
      <echo message="-----------------------------------"/>
      <echo message=""/>
      <echo message="Available targets are:"/>
      <echo message=""/>
      <echo message="deploy    --> Deploy application as directory"/>
      <echo message="deploywar --> Deploy application as a WAR file"/>
      <echo message=""/>
   </target>

   <target name="build" description="Compile main
      source tree java files">
   <mkdir dir="${build.dir}"/>
      <javac destdir="${build.dir}" source="1.5"
         target="1.5" debug="true"
         deprecation="false" optimize="false" failonerror="true">
      <src path="${src.dir}"/>
      <classpath refid="master-classpath"/>
      </javac>
   </target>

   <target name="deploy" depends="build"
      description="Deploy application">

   <copy todir="${deploy.path}/${name}"
```

```
      preservelastmodified="true">
      <fileset dir="${web.dir}">
         <include name="**/*.*"/>
      </fileset>
   </copy>
   </target>

   <target name="deploywar" depends="build"
      description="Deploy application as a WAR file">
      <war destfile="${name}.war"
         webxml="${web.dir}/WEB-INF/web.xml">
         <fileset dir="${web.dir}">
            <include name="**/*.*"/>
         </fileset>
      </war>
      <copy todir="${deploy.path}" preservelastmodified="true">
         <fileset dir=".">
            <include name="*.war"/>
         </fileset>
      </copy>
   </target>


   <target name="clean" description="Clean output directories">
      <delete>
         <fileset dir="${build.dir}">
            <include name="**/*.class"/>
         </fileset>
      </delete>
   </target>
</project>
```

In the above mentioned example:

- We first declare the path to the webapps folder in Tomcat in the build properties file as the **deploy.path** variable.
- We also declare the source folder for the java files in the **src.dir** variable.

- Then, we declare the source folder for the web files in the **web.dir** variable. **javadoc.dir** is the folder for storing the java documentation, and **build.dir** is the path for storing the build output files.
- After that, we declare the name of the web application, which is **fax** in our case.
- We also define the master class path, which contains the JAR files present in the WEB-INF/lib folder of the project.
- We also include the class files present in the **build.dir** in the master class path.
- The Javadoc target produces the javadoc required for the project and the usage target is used to print the common targets that are present in the build file.

The above example shows two deployment targets: **deploy** and **deploywar.**

The deploy target copies the files from the web directory to the deploy directory preserving the last modified date time stamp. This is useful, when deploying to a server that supports hot deployment.

The clean target clears all the previously built files.

The deploywar target builds the war file and then, copies the war file to the deploy directory of the application server.

12. Apache ANT — Deploying Applications

In the previous chapter, we have learnt how to package an application and deploy it to a folder.

In this chapter, we are going to deploy the web application directly to the application server deploy folder and then, we are going to add a few Ant targets to start and stop the services.

Let us continue with the **Hello World** fax web application. This is a continuation of the previous chapter; the new components are highlighted in **bold.**

build.properties

The file for build.properties is given below:

```
# Ant properties for building the springapp

appserver.home=c:\\install\\apache-tomcat-7.0.19
# for Tomcat 5 use $appserver.home}/server/lib
# for Tomcat 6 use $appserver.home}/lib
appserver.lib=${appserver.home}/lib

deploy.path=${appserver.home}/webapps

tomcat.manager.url=http://www.tutorialspoint.com:8080/manager
tomcat.manager.username=tutorialspoint
tomcat.manager.password=secret
```

build.xml

The file for build.xml is as follows:

```
<?xml version="1.0"?>

<project name="fax" basedir="." default="usage">
   <property file="build.properties"/>
   <property name="src.dir" value="src"/>
   <property name="web.dir" value="war"/>
   <property name="javadoc.dir" value="doc"/>
```

```
<property name="build.dir" value="${web.dir}/WEB-INF/classes"/>
<property name="name" value="fax"/>

<path id="master-classpath">
   <fileset dir="${web.dir}/WEB-INF/lib">
      <include name="*.jar"/>
   </fileset>
<pathelement path="${build.dir}"/>
</path>

<target name="javadoc">
<javadoc packagenames="faxapp.*" sourcepath="${src.dir}"
   destdir="doc" version="true" windowtitle="Fax Application">
   <doctitle><![CDATA[<h1>= Fax Application
      =</h1>]]></doctitle>
   <bottom><![CDATA[Copyright © 2011. All
      Rights Reserved.]]></bottom>
   <group title="util packages" packages="faxapp.util.*"/>
   <group title="web packages" packages="faxapp.web.*"/>
   <group title="data packages" packages="faxapp.entity.*:faxapp.dao.*"/>
</javadoc>
</target>

<target name="usage">
<echo message=""/>
<echo message="${name} build file"/>
<echo message="-----------------------------------"/>
<echo message=""/>
<echo message="Available targets are:"/>
<echo message=""/>
<echo message="deploy    --> Deploy application as directory"/>
<echo message="deploywar --> Deploy application as a WAR file"/>
<echo message=""/>
</target>


<target name="build" description="Compile main
```

```
   source tree java files">
<mkdir dir="${build.dir}"/>
<javac destdir="${build.dir}" source="1.5" target="1.5" debug="true"
   deprecation="false" optimize="false" failonerror="true">
   <src path="${src.dir}"/>
   <classpath refid="master-classpath"/>
</javac>
</target>

<target name="deploy" depends="build" description="Deploy application">
   <copy todir="${deploy.path}/${name}"
      preservelastmodified="true">
      <fileset dir="${web.dir}">
         <include name="**/*.*"/>
      </fileset>
   </copy>
</target>

<target name="deploywar" depends="build"
   description="Deploy application as a WAR file">
   <war destfile="${name}.war"
      webxml="${web.dir}/WEB-INF/web.xml">
      <fileset dir="${web.dir}">
         <include name="**/*.*"/>
      </fileset>
   </war>
   <copy todir="${deploy.path}" preservelastmodified="true">
      <fileset dir=".">
         <include name="*.war"/>
      </fileset>
   </copy>
</target>



<target name="clean" description="Clean output directories">
```

```
      <delete>
         <fileset dir="${build.dir}">
            <include name="**/*.class"/>
         </fileset>
      </delete>
   </target>
<!-- ================================================================= -->
<!-- Tomcat tasks -->
<!-- ================================================================= -->

<path id="catalina-ant-classpath">
<!-- We need the Catalina jars for Tomcat -->
<!--  * for other app servers - check the docs -->
   <fileset dir="${appserver.lib}">
      <include name="catalina-ant.jar"/>
   </fileset>
</path>

<taskdef name="install" classname="org.apache.catalina.ant.InstallTask">
   <classpath refid="catalina-ant-classpath"/>
</taskdef>
<taskdef name="reload" classname="org.apache.catalina.ant.ReloadTask">
   <classpath refid="catalina-ant-classpath"/>
</taskdef>
<taskdef name="list" classname="org.apache.catalina.ant.ListTask">
   <classpath refid="catalina-ant-classpath"/>
</taskdef>
<taskdef name="start" classname="org.apache.catalina.ant.StartTask">
   <classpath refid="catalina-ant-classpath"/>
</taskdef>
<taskdef name="stop" classname="org.apache.catalina.ant.StopTask">
   <classpath refid="catalina-ant-classpath"/>

</taskdef>
```

```
<target name="reload" description="Reload application in Tomcat">

   <reload url="${tomcat.manager.url}"username="${tomcat.manager.username}"
      password="${tomcat.manager.password}" path="/${name}"/>
</target>
</project>
```

In this example, we have used Tomcat as our application server.

First, in the build properties file, we have defined some additional properties which are explained below:

- The **appserver.home** points to the installation path to the Tomcat application server.
- The **appserver.lib** points to the library files in the Tomcat installation folder.
- The **deploy.path** variable now points to the webapp folder in Tomcat.

Applications in Tomcat can be stopped and started using the Tomcat manager application. The URL for the manager application, username and password are also specified in the build.properties file.

Next, we declare a new CLASSPATH that contains the **catalina-ant.jar**. This jar file is required to execute Tomcat tasks through Apache Ant.

Tasks

The catalina-ant.jar provides the following tasks:

Properties	Description
InstallTask	Installs a web application. Class Name: org.apache.catalina.ant.InstallTask
ReloadTask	Reload a web application. Class Name: org.apache.catalina.ant.ReloadTask
ListTask	Lists all web applications. Class Name: org.apache.catalina.ant.ListTask
StartTask	Starts a web application. Class Name: org.apache.catalina.ant.StartTask
StopTask	Stops a web application. Class Name: org.apache.catalina.ant.StopTask
ReloadTask	Reloads a web application without stopping. Class Name: org.apache.catalina.ant.ReloadTask

The reload task requires the additional parameters which are as follows:

- URL to the manager application.
- Username to restart the web application.
- Password to restart the web application.

- Name of the web application to be restarted.

Let us issue the **deploy-war** command to copy the webapp to the Tomcat webapps folder and then, let us reload the Fax Web application. The following outcome is the result of running the Ant file:

```
C:\>ant deploy-war
Buildfile: C:\build.xml

BUILD SUCCESSFUL
Total time: 6.3 seconds

C:\>ant reload
Buildfile: C:\build.xml

BUILD SUCCESSFUL
Total time: 3.1 seconds
```

Once the above task is run, the web application is deployed and the web application is reloaded.

13. Apache ANT — Executing Java code

You can use Ant to execute the Java code. In the following example, the java class takes in an argument (administrator's email address) and send out an email.

```
public class NotifyAdministrator
{
   public static void main(String[] args)
   {
      String email = args[0];
      notifyAdministratorviaEmail(email);
      System.out.println("Administrator "+email+" has been notified");
   }
   public static void notifyAdministratorviaEmail(String email
   {
        //......
   }
}
```

Here is a simple build that executes this java class.

```
<?xml version="1.0"?>
<project name="sample" basedir="." default="notify">
   <target name="notify">
      <java fork="true" failonerror="yes" classname="NotifyAdministrator">
         <arg line="admin@test.com"/>
      </java>
   </target>
</project>
```

When the build is executed, it produces the following outcome:

```
C:\>ant
Buildfile: C:\build.xml

notify: [java] Administrator admin@test.com has been notified
```

```
BUILD SUCCESSFUL

Total time: 1 second
```

In this example, the java code does a simple thing which is, to send an email. We could have used the built in the Ant task to do that.

However, now that you have got the idea, you can extend your build file to call the java code that performs complicated things. For example: encrypts your source code.

14. Apache ANT — Eclipse Integration

If you have downloaded and installed Eclipse already, you have very little to do to get started. Eclipse comes pre bundled with the Ant plugin, ready to use.

Follow the simple steps, to integrate Ant into Eclipse.

- Make sure that the build.xml is a part of your java project, and does not reside at a location that is external to the project.
- Enable Ant View by following **Window > Show View > Other > Ant > Ant.**
- Open Project Explorer, drag the build.xml into the Ant View.

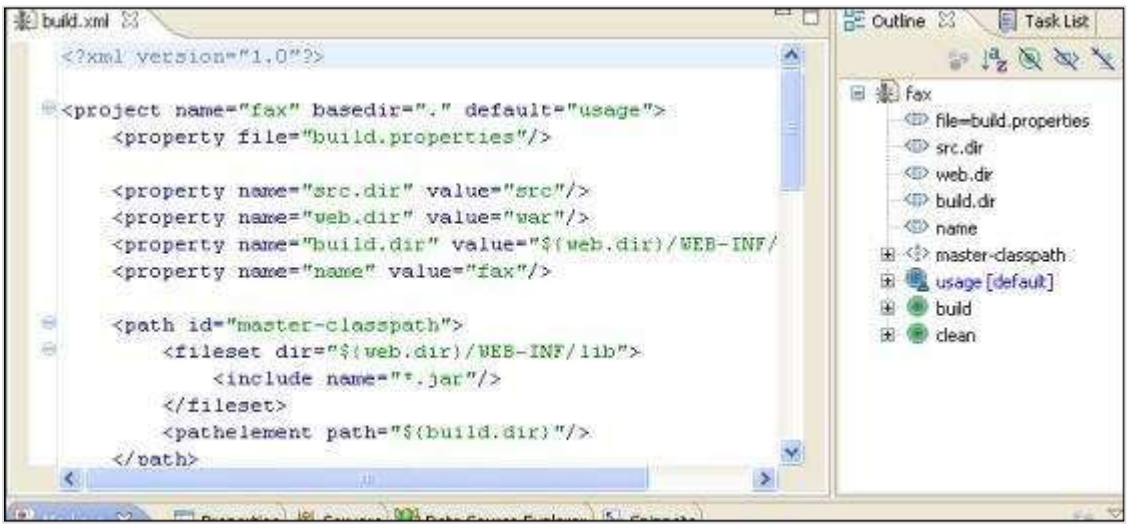
Your Ant view looks similar to the one given below:

Clicking on the targets, build / clean / usage will run Ant with the target.

Clicking "fax" will execute the default target - **usage.**

The Ant Eclipse plugin also comes with a good editor for editing build.xml files. The editor is aware of the build.xml schema and can assist you with code completion.

To use the Ant editor, right click your build.xml (from the Project Explorer) and select Open with > Ant Editor. The Ant editor should look something similar to:



The Ant editor lists the targets on the right hand side. The target list serves as a bookmark that allows you to jump straight into editing a particular target.

15. Apache ANT — JUnit Integration

JUnit is the commonly used unit testing framework for Java-based developments. It is easy to use and easy to extend. There are a number of JUnit extensions available. If you are unfamiliar with JUnit, you should download it from www.junit.org and read its manual.

This chapter shows how to execute JUnit tests by using Ant. The use of Ant makes it straight forward through the JUnit task.

The attributes of the JUnit task are presented below:

Properties	Description
dir	Where to invoke the VM from. This is ignored when fork is disabled.
jvm	Command used to invoke the JVM. This is ignored when fork is disabled.
fork	Runs the test in a separate JVM.
errorproperty	The name of the property to set if there is a JUnit error.
failureproperty	The name of the property to set if there is a JUnit failure.
haltonerror	Stops execution when a test error occurs.
haltonfailure	Stops execution when a failure occurs.
printsummary	Advises Ant to display simple statistics for each test.
showoutput	Advises Ant to send the output to its logs and formatters.
tempdir	Path to the temporary file that Ant will use.
timeout	Exits the tests that take longer to run than this setting (in milliseconds).

Let us continue the theme of the **Hello World** Fax web application and add a JUnit target.

The following example shows a simple JUnit test execution:

```
<target name="unittest">
   <junit haltonfailure="true" printsummary="true">
      <test name="com.tutorialspoint.UtilsTest"/>
   </junit>
</target>
```

This example shows the execution of JUnit on the com.tutorialspoint.UtilsTest junit class.

Running the above code produces the following output:

```
test:
```

```
[echo] Testing the application
[junit] Running com.tutorialspoint.UtilsTest
[junit] Tests run: 12, Failures: 0, Errors: 0, Time elapsed: 16.2 sec
BUILD PASSED
```

16. Apache ANT — Extending Ant

Ant comes with a predefined set of tasks, however you can create your own tasks, as shown in the example below.

Custom Ant Tasks should extend the **org.apache.tools.ant.Task** class and should extend the execute() method.

Below is a simple example:

```
package com.tutorialspoint.ant;
import org.apache.tools.ant.Task;
import org.apache.tools.ant.Project;
import org.apache.tools.ant.BuildException;
public class MyTask extends Task {
   String message;
   public void execute() throws BuildException {
      log("Message: " + message, Project.MSG_INFO);
   }
   public void setMessage(String message) {
      this.message= message;
   }
}
```

To execute the custom task, you need to add the following to the **Hello World** Fax web application:

```
<target name="custom">
   <taskdef name="custom" classname="com.tutorialspoint.ant.MyTask" />
   <custom message="Hello World!"/>
</target>
```

Executing the above custom task prints the message 'Hello World!'

```
c:\>ant custom
test:
[custom] Message : Hello World!
elapsed: 0.2 sec
BUILD PASSED
```

This is just a simple example. You can use the power of Ant to do whatever you want to improve your build and deployment process.