LEARN 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 you have knowledge of software development using any programming language, especially Java, and the software build and deployment process.

Disclaimer & Copyright

© Copyright 2014 by Tutorials Point (I) Pvt. Ltd.

All the content and graphics published in this e-book are the property of Tutorials Point (I) Pvt. Ltd. The user of this e-book is prohibited to reuse, retain, copy, distribute or republish any contents or a part of contents of this e-book in any manner without written consent of the publisher.

We strive to update the contents of our website and tutorials as timely and as precisely as possible, however, the contents may contain inaccuracies or errors. Tutorials Point (I) Pvt. Ltd. provides no guarantee regarding the accuracy, timeliness or completeness of our website or its contents including this tutorial. If you discover any errors on our website or in this tutorial, please notify us at contact@tutorialspoint.com.

Table of Contents

About the Tutorial i

Audience i

Prerequisites i

Disclaimer & Copyright i

Table of Contents ii

1. ANT INTRODUCTION 1

Need for a Build Tool 1

History of Apache ANT 1

Features of Apache ANT 1

2. ANT ENVIRONMENT SETUP 3

Installing Apache ANT 3

Verifying Apache ANT Installation 3

Installing Eclipse 4

3. ANT BUILD FILES 5

4. ANT PROPERTY TASK 8

5. ANT PROPERTY FILES 10

build.xml 10

build.properties 11

6. ANT SERVICES 13

Fileset 13

Patternset 13

Filelist 14

Filterset 14

Path 15

7. ANT BUILDING PROJECTS....16

8. ANT BUILD DOCUMENTATION....20

Attributes....20

Putting it All Together....20

9. ANT CREATING JAR FILES....23

10. ANT PACKAGING APPLICATIONS....29

build.properties....29

build.xml....29

11. ANT FOR DEPLOYING APPLICATIONS....33

build.properties....33

build.xml....33

12. EXECUTING JAVA CODE....40

13. ECLIPSE INTEGRATION....42

14. ANT JUNIT INTEGRATION....44

15. EXTENDING ANT....46

1. ANT INTRODUCTION

ANT stands for Another Neat Tool. Before going into details of Apache ANT, you must understand the need for a build tool.

Need for a Build Tool

Apart from coding and testing, the JAVA developers are engaged in the following tasks:

- 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 command line.

History of Apache ANT

- ANT was created by James Duncan Davidson, the original author of Tomcat.
- It was originally used to build Tomcat, and was bundled as part of Tomcat distribution.
- ANT was born out of the problems and complexities associated with Apache Make tool.
- ANT was promoted as an independent project in Apache in the year 2000.
- The latest version of Apache ANT as on May 2014 is 1.9.4.
- NAnt is a .NET build tool that is similar to ANT, except that is used to build .NET apps.

Features of Apache ANT

- ANT is the most complete Java build and deployment tool available.
- ANT is platform neutral and can handle platform-specific properties such as file separators.

tutorialspoint
SIMPLYEASYLEARNING

- ANT 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. ANT ENVIRONMENT SETUP

Apache ANT is distributed under the Apache Software License, a full-fledged open source license certified by the open source initiative.

The latest Apache Ant version, including 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 here.

- 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 using Winzip, winRAR, 7-zip or similar tools, on say c:\folder.
- Create a new environment variable called **ANT_HOME** that points to the ANT installation folder, in this case, **c:\apache-ant-1.9.4-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-1.9.4-bin\bin** folder.

Verifying Apache ANT Installation

To verify the successful installation of Apache ANT on your computer, start command prompt and type ant. You should see an output similar to:

```
C:\>ant -version
Apache Ant(TM) version 1.9.4 compiled on December 20 2014
```

If you do not see the above reply, then please confirm if you followed the installation steps properly.

Installing Eclipse

This tutorial also covers integration of ANT with Eclipse IDE. Hence, if you have not installed Eclipse already, please download and install Eclipse as given below:

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

3. ANT BUILD FILES

Typically, ANT's build file called **build.xml** should reside in the base directory of the project. Although, you are free to use other file name or place for the build file.

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

```
<?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 must 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:

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 **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, start command prompt and navigate to the folder where the build.xml resides, and type **ant info**. You can also type **ant** instead. Both will

tutorialspoint
SIMPLYEASYLEARNING

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. ANT PROPERTY TASK

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

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. For Example, file.separator is 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 1.9.4
     compiled on December 20 2014 - You are at www.tutorialspoint.com

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


tutorialspoint
SIMPLYEASYLEARNING

5. 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.

Storing the properties in a separate file is advantageous as:

- 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 **build.properties** and is placed alongwith the **build.xml** file. You can create multiple build properties files based on the deployment environment - 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-value pair. The name-value pairs are separated by equals (=) signs. It is highly recommended that the properties are annotated with proper comments. Comments are listed using the hash (#) character.

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

build.xml

```
<?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>
```


tutorialspoint
SIMPLYEASYLEARNING

build.properties

```
# 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 represented below once again.

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.

In the example presented in this chapter, we use the **ant.version** built-in property.

tutorialspoint
SIMPLYEASYLEARNING

6. ANT SERVICES

ANT provides a number of predefined Services. It provides services such as File set (we can include/exclude set of files), Pattern Set (easily filter files or folders), File List (explicit set of files), Filter Set (pattern matching for text/file/folder), Path (Represents the class path), etc. The following services are provided by Apache ANT:

Fileset

The fileset 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 (${src}) points to the source folder of the project. The fileset selects all .java files in the source folder except those, which contain the word 'Stub'. The **casesensitive** filter is applied to the fileset which means that a file with the name **Samplestub.java** will not be excluded from the fileset.

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

Patternset

A patternset is a pattern that allows to filter the files or folders easily, based on certain patterns. 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 patternset:

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


tutorialspoint
SIMPLYEASYLEARNING

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

```
<fileset dir="${src}" casesensitive="yes">
  <patternset refid="java.files.without.stubs"/>
 </fileset>
```

Filelist

The filelist service is similar to the fileset except the following differences:

- Filelist contains explicitly named lists of files and it does not support wild cards.
- Filelist tag can be applied for existing or non-existing files.

Let us see the following example of filelist . 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>
```

Filterset

Using filterset service along with the copy task, enables you to replace certain text in all 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. Here, 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.

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


tutorialspoint
SIMPLYEASYLEARNING

```
</copy>
```

Path

The **path** service 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. Here, 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.

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


tutorialspoint
SIMPLYEASYLEARNING

7. ANT BUILDING PROJECTS

Now that you have learnt basic services in ANT, it is time to put your knowledge into action. 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 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.

Let us assume the following folder hierarchy of a sample project. The project is the **Hello World Fax Web Application**. Follow the same folder structure 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. The **src.dir** refers to the source folder of the project, where the java source files can be found. The **web.dir** refers to the web source folder of the project. This is where you can find the JSPs, web.xml, css, javascript, and other web related files. Finally, the **build.dir** refers to the output folder of the project compilation.

```
<?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>
```

Let us disscuss above given build.xml in detail 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"/>
```

Properties can refer to other properties. The **build.dir** property makes a reference to the **web.dir** property. 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 target 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">
```


tutorialspoint
SIMPLYEASYLEARNING

```
   <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 result of executing an ANT file is as shown:

```
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. ANT BUILD DOCUMENTATION

Documentation is necessary task in the any project execution. Documentation plays a great role in the project maintenance. Java makes documentation easier by the use of the inbuilt **javadoc** tool. The javadoc tool is highly flexible and allows a number of configuration options. You can visit Java Documentation Tutorial for more information.

Apache ANT can generate documents on demand. ANT exploits the configuration options via the javadoc task.

Attributes

You can specify source using the attributes sourcepath, sourcepathref, or sourcefiles.

You can use the attributes in the following ways:

- the attribute **sourcepath** to point to the folder of the source files (for example, src folder).
- the attribute **sourcepathref** to refer to a path, which is referenced by the path attribute (for example, delegates.src.dir).
- the attribute **sourcefiles** when you want to specify the individual files as a comma separated list.
- You can specify destination path using the **destdir** folder (for example, build.dir)
- You can filter the **javadoc** task by specifying the package names to be included. You can achieve this by using the **packagenames** attribute, a comma separated list of package files.
- You can filter the javadoc process to show only the public, private, package, or protected classes, and members. You can achieve this by using the **private**,**public**,**package** and **protected** attributes.
- You can also tell the javadoc task to include the author and version information using the respective attributes.
- You can also group the packages together using the **group** attribute, so that it is easy to navigate.

Putting it All Together

Let us continue with the **Hello World Fax Web Application** folder structure. Let us add a documentation target to our Fax application project build.xml.

The following example depicts javadoc task used in our project. Here, we specified the javadoc to use the **src.dir** as the source directory, and **doc** as the target

directory. We also customized the window title, the header, and footer information that appears on the java documentation pages.

We created three groups for classes as follows:

1. Utility classes in our source folder
2. User interfaces classes
3. Database related classes.

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. This 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
```


tutorialspoint
SIMPLYEASYLEARNING

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

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

tutorialspoint
SIMPLYEASYLEARNING

9. ANT CREATING JAR FILES

The next logical step after compiling your java source files, is to build the java archive, i.e. the JAR file. Creating JAR files with ANT is quite easy with the **jar** task. The commonly used attributes of the jar task are:

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 Web Application** project folder structure, let us add a new target to produce the jar files. But before that let us consider the jar task given below. Here, the **web.dir** property points to the path of the web source

files. In our case, this is where the util.jar is placed. The **build.dir** property 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 is place in the web application lib folder.

```
<jar destfile="${web.dir}/lib/util.jar"
     basedir="${build.dir}/classes"
     includes="faxapp/util/**"
     excludes="**/Test.class"
/>
```

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 is updated as:

```
<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.

On execution of the ANT file, the following result is generated:

```
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.

tutorialspoint
SIMPLYEASYLEARNING

Ant creating war files

Creating 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 Web Application** project folder structure, 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.

tutorialspoint
SIMPLYEASYLEARNING

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 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:

```
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
```


tutorialspoint
SIMPLYEASYLEARNING

10. ANT PACKAGING APPLICATIONS

We 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 build.properties is used to mention user defined variables which are helpful to build the project such as file path, classpath, dburls, etc. The following example defines **deploy.path** variable.

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

build.xml

Refer the following code. In this 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 **src.dir** variable. Then we declare the source folder for the web files in **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. Then we declare the name of the web application, which is **fax** in our case.

We define the master class path which conains the JAR files present in the WEB-INF/lib folder of the project. We 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.

```
<?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"/>
```


tutorialspoint
SIMPLYEASYLEARNING

```
<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>
```


tutorialspoint
SIMPLYEASYLEARNING

```
        </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>
```

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

tutorialspoint
SIMPLYEASYLEARNING

11. ANT FOR DEPLOYING APPLICATIONS

You 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, then we are going to add few ANT targets to start and stop the services. Let us continue with the **Hello World** Fax Application folder structure.

build.properties

```
# 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

```
<?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"/>
```


tutorialspoint
SIMPLYEASYLEARNING

```
<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"/>
```


tutorialspoint
SIMPLYEASYLEARNING

```
        <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">
```


tutorialspoint
SIMPLYEASYLEARNING

```
            <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 exercise, we used Tomcat as our application server. First, in the build properties file, we defined some additional properties.

- 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.

tutorialspoint
SIMPLYEASYLEARNING

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 following additional parameters:–

1) URL to the manager application

2) Username to restart the web application

3) Password to restart the web application

4) 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. On executing the ANT file, the following result is seen:

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

BUILD SUCCESSFUL
Total time: 6.3 seconds
```


tutorialspoint
SIMPLYEASYLEARNING

```
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.

tutorialspoint
SIMPLYEASYLEARNING

12. EXECUTING JAVA CODE

You can use ANT to execute java code. In this example below, the java class takes in an argument (administrator's email address) and sends 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
```


tutorialspoint
SIMPLYEASYLEARNING

```
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 - to send an email. We could have used the built in Ant task to do that. However, now that you have got the idea you can extend your build file to call java code that performs complicated things, for example: encrypts your source code.

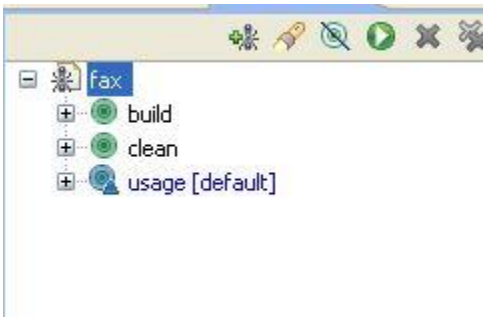
13. ECLIPSE INTEGRATION

Eclipse comes pre bundled with the ANT plugin, ready for you to use.

Follow the simple steps, to integrate ANT into Eclipse.

- Make sure that the build.xml is 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:

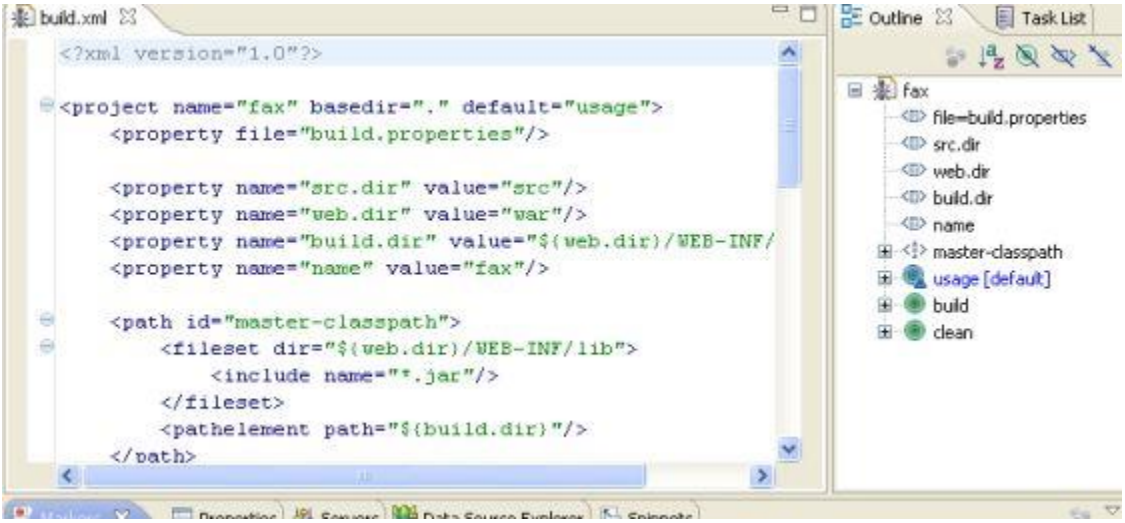


Click on the targets, build / clean / usage to run ANT with the target.

Click on fax to 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.

14. ANT JUNIT INTEGRATION

Junit is the commonly used unit testing framework for development based on Java. It is easy to use and easy to extend. There are a number of JUnit extensions available. If you are not much familiar with Junit, download junit from www.junit.org and read the junit manual.

This tutorial discusses about executing the junit tests using ANT. ANT makes this straight forward through the Junit task.

The attribute of the junit task are:

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	Advices Ant to display simple statistics for each test
showoutput	Adivces Ant tosend 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 folder structure 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
```

15. EXTENDING ANT

ANT comes with a predefined set of tasks, however you are not limited to using only the available tasks. 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. See the following 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 build.xml:

```
<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 improve your build and deployment process.