



JavaOneSM
Sun's 2002 Worldwide Java Developer Conference™

Using the Apache Group's Ant Build Tool

David S. Read, Chief Technologist,
Blue Slate Solutions

Paul Evans, Senior Java Developer,
Blue Slate Solutions

Goal

Ant, from the Apache Foundation, is a powerful tool for simplifying the application build and deployment process — we will introduce you to this tool and demonstrate its inherent flexibility and extendibility



Learning Objectives

During this presentation we will define and demonstrate:

- The purpose of Ant

- The utilization of Ant as an application build tool

- The extensibility of Ant for various requirements



Speaker's Qualifications

David is a Sun Certified Programmer for the Java™ 2 Platform and a Red Hat Certified Engineer. He has been leading IT teams for over 10 years. Dave has also been using technologies for the Java platform since 1996. In addition he has led Java technology developer training for several offshore development companies

Paul is a Sun Certified Programmer for the Java 2 Platform. He is also a Sun Certified Developer for the Java 1.1 Platform. He has worked as a consultant in a Fortune 10 company for 3 years building Java technology-based systems

Both speakers have worked with geographically dispersed development teams in Fortune 10 companies



IDEs, Shell Scripts, Makefiles, ...

As project teams change, would you rather your developers:

Spend time focused on design and coding; or
Waste time configuring their particular development environment for each project?

Ant enhances developer productivity



Presentation Agenda

What is Ant?

Installation and Configuration

The Build Script

- Environment

- Targets

- Tasks

Extending Ant

Making Ant Work for You



What Is Ant?

A Java™ technology-based tool for building applications for the Java platform

An open architecture allowing you to add functionality



What Can Ant Do?

Automate those tasks that are necessary to build and deploy Java technology-based applications

- Compile Java technology Source

- Package into JAR and WAR Files

- Run Test Scripts

- Interact with SCCS Systems

- Deploy to Clients and Servers

With automation, development time is reduced



Benefits of Ant

Platform independent

Performs any Java technology-based task

Creates consistent processes

Automates repetitive tasks

Reuses tasks and scripts between projects

Removes ties to IDEs



Installation and Overview

Installation of Ant

Java™ Development Kit (JDK™) software, at least version 1.1, must be installed

Download the latest production version from:
<http://jakarta.apache.org/ant/>

You will want the base installation file and the optional JAR

Set `ANT_HOME` to the directory where Ant is installed and make sure `JAVA_HOME` points

to your installation of JDK software

Add `ANT_HOME/bin` to your path

Place the optional JAR in `ANT_HOME/lib`



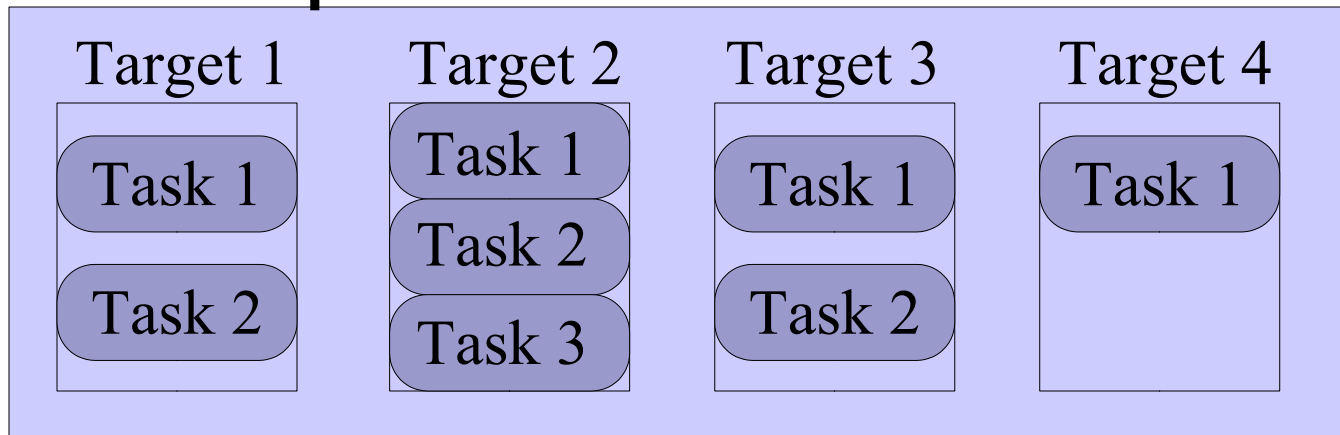
Ant's Build Script: build.xml

Defines one or more targets and determines tasks to be carried out to achieve each target

Tasks are the workhorses—They are the “Java classes” that carry out a specific operation

The build file may use properties which promote reusability of build files between projects

Build Script



A build.xml Example

```
<project name="HelloWorld" default="comp" basedir=".">
  <!-- set global properties for this build -->
  <property name="srcDir" value="Source"/>
  <property name="bldDir" value="build"/>

  <!-- setup steps required before compilation -->
  <target name="init">
    <!-- Create the time stamp -->
    <tstamp/>
    <!-- Create the build dir for compile -->
    <mkdir dir="${bldDir}"/>
  </target>

  <!-- compilation -->
  <target name="comp" depends="init">
    <javac srcdir="${srcDir}" destdir="${bldDir}"/>
  </target>
</project>
```

Environment

Environment

Within Ant you can define properties and more complex collections, like a set of files to be placed on the classpath

Properties can be self-contained within the build script or externally defined

Some properties are built-in and supply information such as the path to the build file, Ant version, name of the project, etc.



Properties

A property is a named variable within the build script

A property is defined with the `<property>` element or with the “`-D`” argument on the command line

```
<property name="projJAR" value="KYN" />  
$ ant -DprojJAR=KYN
```

A property is used anywhere a literal is expected by placing the property name within “`${`” and “`}`”

```
<jar jarfile="${projJAR}.jar">...</jar>
```



<pathelement>

The **<pathelement>** element is the most basic building block for defining a path; It accepts either a location or path attribute

The location attribute expects a single file or directory, which must be a relative or absolute pathname

The path attribute is typically used for including predefined paths; It expects a colon or semi-colon separated list of pathnames

```
<pathelement location="/jars/jdbc.jar" />  
<pathelement path="{classpath}" />
```



<classpath> and <path>

Using the **<classpath>** element, you define a classpath to be used within a task

The **<classpath>** element supports **<pathelement>** as a nested element

You can define multiple paths, using the **<path>** element; These paths can be given unique names and used as the classpath for different tasks



Example <classpath> and <path>

Here we define a classpath using two directories and a JAR

```
<classpath>
  <pathelement path="/classes1:/classes2"/>
  <pathelement location="/jars/special.jar"/>
</classpath>
```

Here we define a named path containing 3 jars and a directory

```
<path id="proj.class.path">
  <pathelement location="/jars/special.jar"/>
  <pathelement location="/jars/special2.jar"/>
  <pathelement location="/jars/special3.jar"/>
  <pathelement location="/classes1"/>
</path>
```



<fileset>

We can define a group of files using absolute filenames and wildcards with the **<fileset>** element

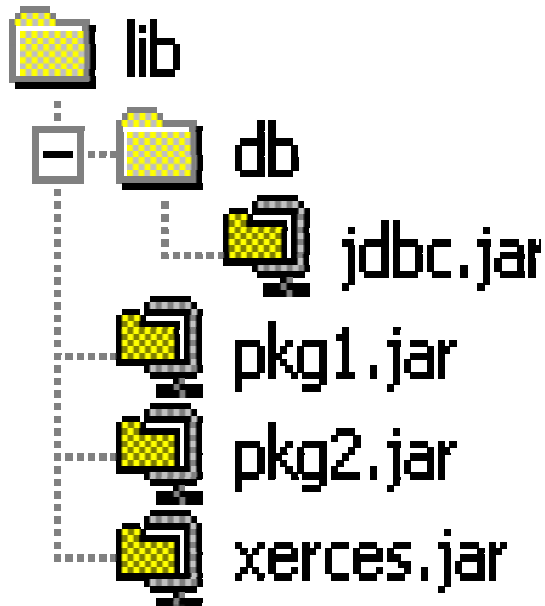
The **<include>** nested element filters the matching files within the fileset

The **<exclude>** nested element filters out matching files

```
<path id="proj.class.path">  
  <fileset dir="/jars">  
    <include name="**/*.jar"/>  
    <exclude name="**/Debug*.jar"/>  
  </fileset>  
</path>
```



<pathelement> and <fileset> Example



Equivalent Paths

```
<path id="proj.class.path">
  <pathelement
    location="/lib/pkg1.jar"/>
  <pathelement
    location="/lib/pkg2.jar"/>
  <pathelement
    location="/lib/db/jdbc.jar"/>
</path>
```

```
<path id="proj.class.path">
  <fileset dir="/lib">
    <include name="**/*.jar"/>
    <exclude
      name="**/xerces*.jar"/>
  </fileset>
</path>
```



Targets

Targets

Control which sets of tasks are required

Similar concept as a makefile target

Can include dependencies for other targets

Can be made optional through the use of the if and unless attributes

The default target is assumed if no target name is specified on the command line



Basic Target Example

Here the default target is **dist**

The **dist** target requires the **compile** target

The **compile** target requires the **init** target

```
<project name="HelloWorld" default="dist" basedir=". ">
  <target name="init"><mkdir dir="{bDir}" /></target>

  <target name="compile" depends="init">
    <javac srcdir="{sDir}" destdir="{bDir}" >
      <classpath refid="proj.class.path" />
    </javac>
  </target>

  <target name="dist" depends="compile">
    <jar jarfile="{projJAR}.jar">
      <fileset dir="{bDir}" />
    </jar>
  </target>
</project>
```



Optional Target Example

Given a request to build the compile target, the test-module target is built, unless the parameter **prod-version** is defined

```
<target name="compile" depends="init, test-module">  
  <javac srcdir="{sDir}" destdir="{bDir}">  
    <classpath refid="proj.class.path"/>  
  </javac>  
</target>
```

```
<target name="test-module" unless="prod-version">  
  <javac srcdir="{tDir}" destdir="{bDir}">  
    <classpath refid="proj.class.path"/>  
  </javac>  
</target>
```

The following command line would not include the test modules

```
ant compile -Dprod-version=anything
```



Tasks

Tasks

Code that is executed by using the appropriate task element

Many built-in and optional tasks including:

`<javac>`, `<java>`, `<jar>`, `<war>`,
`<ftp>`, `<telnet>`

Tasks can have multiple attributes, or support nested elements, supplying “arguments” to the task

Easy to add new tasks by building a “Java class” that extends `org.apache.tools.ant.Task`



<javac>

Compile “Java source code”
(Java™ programming language source code)

By default it will recurse through all subdirectories in its source tree locating “Java files”; These are checked against its destination tree for outdated or missing class files

Since it only uses the directory tree and file names (.java vs .class), it will always build “Java files” containing differently named classes or source files not arranged to match the package structure



<javac> Options

The behavior of **<javac>** can be altered through the use of attributes and nested elements

A few attributes and nested elements of note:

srcdir: the relative root of the source tree—
Can also be **<src>** nested element

destdir: the relative root of the destination tree

excludes/includes: file name patterns to exclude/include

classpath: classpath to use. Can also be **<classpath>** nested element

deprecation: report deprecated methods



<javac> Sample

```
<javac srcdir="{srcDir}"
      destdir="{buildDir}"
      deprecation="yes">
  <classpath>
    <fileset dir="{standardJARDir}">
      <include name="**/*.jar"/>
    </fileset>
  </classpath>
</javac>
```

<jar>

Creates a JAR file

Similar functionality as the jar utility

May use nested **<fileset>** elements to support more complex inclusion rules

Relative root of each fileset is the same

```
<jar jarfile="${dist}/${jarFile}.jar"  
      manifest="manifest.txt">  
  <fileset dir="${buildDir}" />  
  <fileset dir="${srcDir}/resources" />  
</jar>
```



<war>

Builds a Web Archive (WAR) file

Similar to **<jar>** but understands the special WEB-INF, WEB-INF/lib and WEB-INF/classes directories

Supports the nested elements **<webinf>**, **<lib>**, and **<classes>** to control content placed in those locations

```
<war warfile="${distDir}/${appNm}.war"
      webxml="${srcDir}/${appNm}.xml" >
  <classes dir="${buildDir}" />
  <fileset dir="${srcDir}/resources" />
  <lib dir="${thirdPartyLibs}" />
  <webinf dir="${addlWebInf}" />
</war>
```



<delete>

Remove a specific file or a set of files within a directory tree

To also have empty directories removed use the **includeEmptyDirs** attribute set to “true”

Useful for a cleanup target

```
<delete includeEmptyDirs="true">  
  <fileset dir="${buildDir}" />  
</delete>
```



Ant In Practice

A Web Application: The “CodeWorld Registration System” Demo

A Java™ technology-based web application used by individuals registering to attend the “CodeWorld event”

Multiple developers located at various sites are building components of the application

Developers test on their local systems

QA deploys to the production system



Defining the Build Script: Produce Two WAR Files

Requirements for build tool:

Produce test and production versions of the WAR file

Deploy application to local test server

Deploy application to remote production server

Send an email to developers, providing compilation error messages, if build fails

Produce
WAR

Deploy
Test

Deploy
Prod

Email



What Do We Mean by Test and Production Versions

The production WAR will not have debug code compiled into the .class files; the test WAR will

This is accomplished by placing debug code within 'if' statements controlled by a 'static final' boolean

```
public final static boolean DEBUG = true;
if (DEBUG) {
    System.out.println("Debug message");
}
```

Produce
WAR

Deploy
Test

Deploy
Prod

Email



Which Ant Tasks?

To accomplish the goal of building a production and test WAR, the `<replace>`, `<touch>`, and `<antcall>` tasks will be used

`<replace>` will be used to toggle the value of the DEBUG boolean between true and false

`<touch>` will be used to update the last modified timestamp on the java source files

`<antcall>` will be used to call a target which builds the WAR—The target will name the WAR based on a passed parameter

Produce
WAR

Deploy
Test

Deploy
Prod

Email



<replace>

The **<replace>** task allows for string replacements within files

This is necessary to put the source files into a “debug” state

```
<replace
  file="${srcDir}/${javaconstants}
  token="DEBUG = false"
  value="DEBUG = true" />
```

Produce
WAR

Deploy
Test

Deploy
Prod

Email



<touch>

The **<touch>** task updates the last modified timestamp on files

This is necessary in this situation to force Ant to recompile the entire source tree for the next build

```
<touch>  
  <fileset dir="${srcDir}" />  
</touch>
```

Produce
WAR

Deploy
Test

Deploy
Prod

Email



<antcall>

The **<antcall>** task allows Ant to be invoked from within a running Ant process

```
<target name="buildwar" depends="compile">  
  <war warfile="${distDir}/${warname}"  
    webxml="${srcDir}/web.xml" />  
</target>
```

```
<target name="buildDebugWar">  
  <antcall target="buildwar">  
    <param name="warname"  
      value="${projectWARName}Debug.war" />  
  </antcall>  
</target>
```

Produce
WAR

Deploy
Test

Deploy
Prod

Email



Creating the Target to Build the Test and Production WARs

```
<!-- Make Project WAR File -->
<target name="dist">
  <!-- prepare for debug build -->
  <replace file="${srcDir}/${javaconstants}"
    token="DEBUG = false" value="DEBUG = true"/>
  <touch><fileset dir="${srcDir}"/></touch>
  <antcall target="buildwar">
    <param name="warname"
      value="${projectWARName}Debug.war"/>
  </antcall>
  <!-- prepare for production build -->
  <replace file="${srcDir}/${javaconstants}"
    token="DEBUG = true" value="DEBUG = false"/>
  <touch><fileset dir="${srcDir}"/></touch>
  <antcall target="buildwar">
    <param name="warname"
      value="${projectWARName}.war"/>
  </antcall>
</target>
```

Deploying to the Local Server: A Generic JRun Command Target

Our local server is running JRun. Here is what the target looks like for executing a command on the local JRun server:

```
<!-- Execute a JRun Command -->  
<target name="jrun.cmd">  
  <java  
    classname="allaire.jrun.tools.WarDeploy"  
    fork="true">  
    <classpath refid="jrun.class.path"/>  
    <arg value="-${cmd}"/>  
    <arg value="-config=deploy.properties"/>  
  </java>  
</target>
```

Produce
WAR

Deploy
Test

Deploy
Prod

Email



Deploying to the Local Server: The Deployment Target

Here is the target to deploy the WAR file using the generic target from the previous example:

```
<!-- Deploy the WAR file locally on JRun -->
<target name="jrun.deploy" depends="dist">
  <!-- Remove the old application -->
  <antcall target="jrun.cmd">
    <param name="cmd" value="remove"/>
  </antcall>

  <!-- Deploy the new application -->
  <antcall target="jrun.cmd">
    <param name="cmd" value="deploy"/>
  </antcall>
</target>
```

Produce
WAR

Deploy
Test

Deploy
Prod

Email



Deploying to a Remote Server

The `<telnet>` and `<ftp>` tasks allow for a straightforward deployment to the remote server

To accomplish the remote deployment we will:

Stop the Tomcat server and erase the exploded copy of the WAR;

FTP the WAR to the deployment directory; and

Restart the Tomcat server

Produce
WAR

Deploy
Test

Deploy
Prod

Email



<telnet>

The **<telnet>** task requires a server attribute

The user id and password may be provided as attributes, or for complete control of the login process, the login steps can be part of the script

The nested elements **<read>** and **<write>** are used to build an interactive script of responses and commands

Produce
WAR

Deploy
Test

Deploy
Prod

Email



Sending Telnet Commands

We built a “tomcat.telnet” target with a cmd property to supply a remote command

```
<target name="tomcat.telnet">
  <telnet server="${tomcat.server}"
    userid="${tomcat.svr.telnet.id}"
    password="${tomcat.svr.telnet.pw}">
    <!-- Wait for command prompt, timeout
      after 5 seconds -->
    <read string="$" timeout="5"/>
    <!-- Output the command -->
    <write string="${cmd}"/>
  </telnet>
</target>
```

Produce
WAR

Deploy
Test

Deploy
Prod

Email



Stopping Tomcat and Removing the Exploded WAR

For stopping the server the cmd property is defined as:

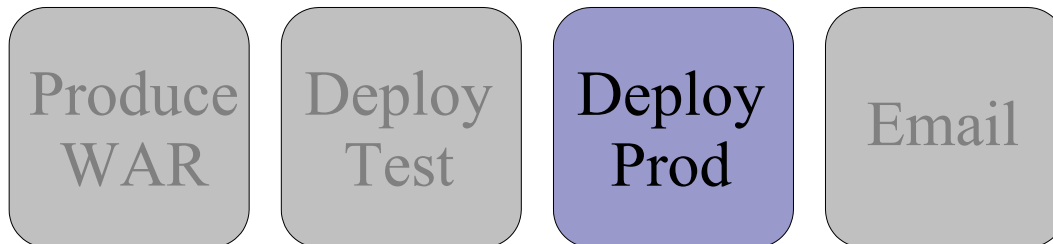
```
tomcat stop
```

For erasing the exploded WAR the cmd property is defined as:

```
rm -Rf ${tomcat.server.deploy.dir}/${projectWARName}
```

For restarting the server the cmd property is defined as:

```
tomcat start
```



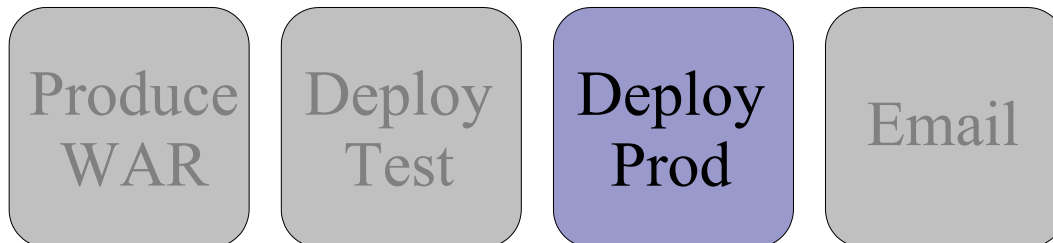
<ftp>

The **<ftp>** task requires the server, userid, and password as attributes

The action parameter determines what FTP operation will be run—Actions such as **get**, **put** and **ls** are defined

You cannot carry out more than one action within a single FTP task

<fileset> elements are used to determine what files are sent or retrieved



FTP the New WAR

We call the target “tomcat.ftpwar” to send the new WAR to the server

```
<target name="tomcat.ftpwar">
  <ftp server="${tomcat.server}"
        userid="${tomcat.server.ftp.id}"
        password="${tomcat.server.ftp.pw}"
        action="put"
        remotedir="${tomcat.server.deploy.dir}"
        binary="yes" verbose="yes">
    <fileset dir="${distDir}">
      <include name="${projectWARName}.war" />
    </fileset>
  </ftp>
</target>
```

Produce
WAR

Deploy
Test

Deploy
Prod

Email



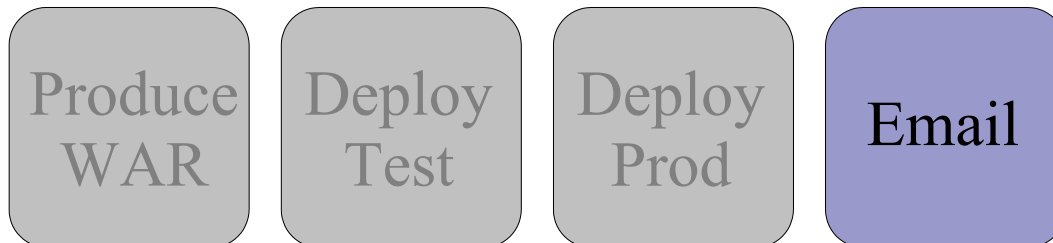
Using a BuildListener to Send an Email if the Build Fails

A build listener is a Java™ programming language class (“Java class”) that implements Ant’s BuildListener interface

It receives messages when compilation starts and stops

The build listener can determine if the build process succeeded or failed

A build listener is registered with Ant through command-line parameters when Ant is executed



BuildListener Code

```
public void buildFinished(BuildEvent evtaEvent) {
    String slTemp;
    StringBuffer sblMsg;
    BufferedReader brlBuildLog;
    Address[] objlTo;
    if (evtaEvent.getException() != null) {
        // Unsuccessful! Send email to developers...
        try {
            objlTo = parseAddress(objcProp.getProperty(EMAIL_TO_KEY));
            brlBuildLog = new BufferedReader(
                new FileReader(objcProp.getProperty(BUILD_LOG)));
            sblMsg = new StringBuffer();
            while ((slTemp = brlBuildLog.readLine()) != null) {
                sblMsg.append(slTemp).append("\n");
            }
            brlBuildLog.close();
            sendEmail(objlTo, "Build failed!, sblMsg.toString());
        }
        catch (Throwable objaThrowable) {
            // log the Throwable
        }
    }
}
```

Produce
WAR

Deploy
Test

Deploy
Prod

Email



Executing Ant With a BuildListener

Here is a command line that executes Ant with a BuildListener attached to the build process

The value of the logfile parameter gives the location where Ant will write the build log— this will be where standard output messages are written

```
$ ant -listener \  
net.blueslate.ant.listener.BuildMonitor \  
-logfile build.log
```

Produce
WAR

Deploy
Test

Deploy
Prod

Email



Demo

Making Ant Work for You

Standardized Build File

Plug-and-Play build files

Modifications to build files from one project to the next become minimal

Allows for new applications to be developed with minimal time put into source arrangement and build file creation

Usage of a standardized build file is dependent on a standardized application source layout

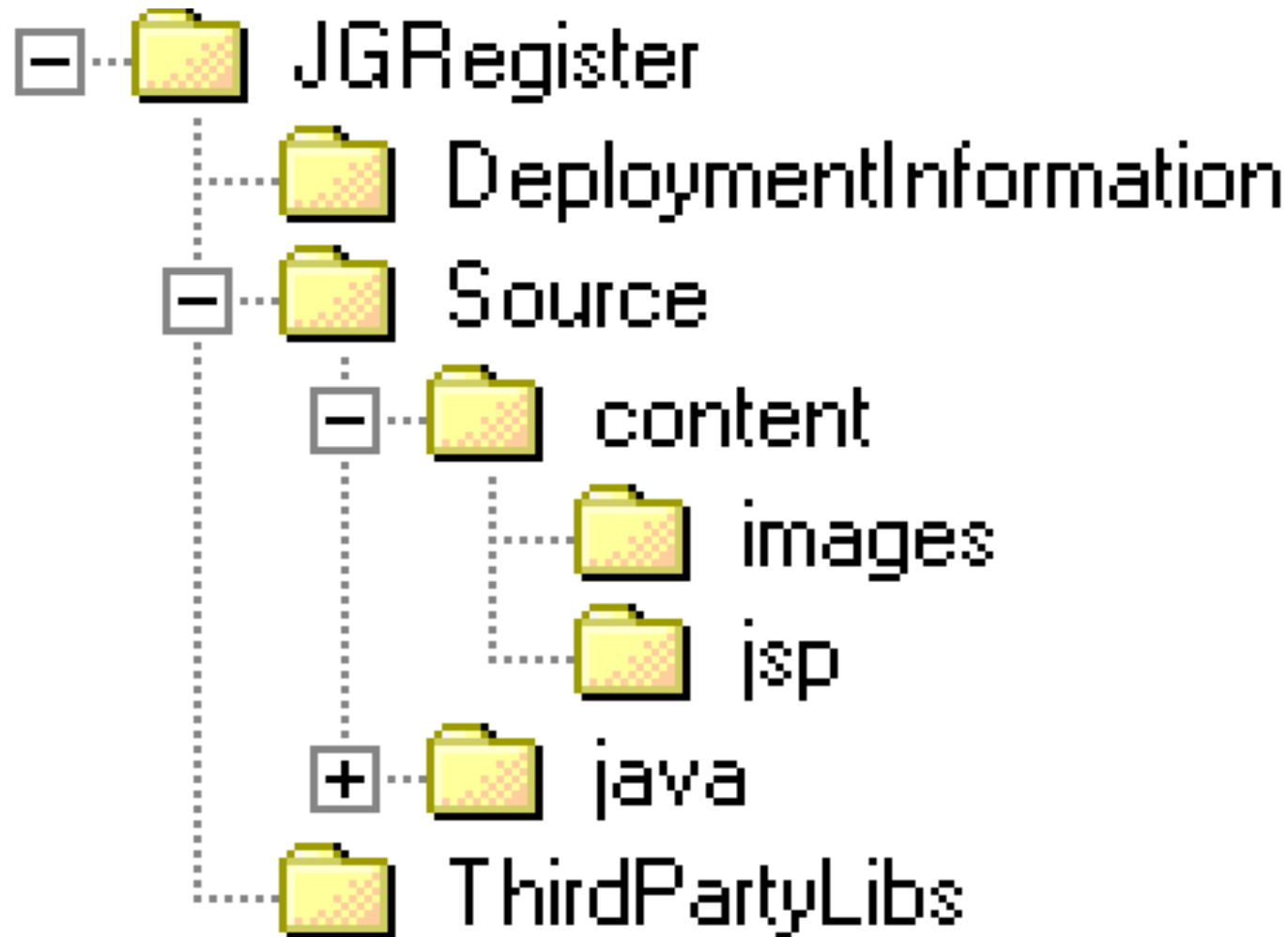


Benefits of Standardized Application Source Layout

Several developers can work on the same project without worrying about source layout for build purposes

Application source will be stored in a source control system using the standardized layout

Sample Source Arrangement



IDE Independence

Building applications using Ant allows developers to use any IDE

Many IDEs have the ability to integrate with Ant either through Ant tasks or command lines accessed through menu items

Ant has built-in support for several IDEs and add-ons, like AntRunner, integrate still others

Tools such as Codewarrior and UltraEdit allow command lines to be assigned to menu items



Common Challenges, 1

Don't reinvent the wheel—Keep it simple and try to use tasks already developed and tested by others

Make sure you have installed all required libraries and that they are on the classpath

Optional tasks may have additional dependencies; For example the `<ftp>` and `<telnet>` tasks require the NetComponents JAR (URL given at the end of this presentation)



Common Challenges, 2

Place task code in the **execute()** method, **not** in the constructor

Use the **<taskdef>** tag to associate a task to its implementation:

```
<taskdef name="myTask"  
         classname="net.blueslate.myTask" />
```

Use **-verbose** flag to see a trace of execution

Search the apache bug database:
<http://nagoya.apache.org/bugzilla/>



Summary

Ant is an effective tool for unifying your shop's build process

Ant provides a flexible design for handling any application build and deployment task

Ant's extensibility allows you to add capabilities unique to your environment's requirements

Ant allows developers to continue to use their preferred development tools for personal productivity



Resources

Ant is available from the Apache Foundation, as part of the Jakarta project:

<http://jakarta.apache.org/ant/>

The build file and source code for our “CodeWorld” demonstration application is available on our website:

<http://www.blueslate.net/javaone2002/>

The netcomponents.jar file, which is required for ftp and telnet support, is available from:

<http://www.savarese.org/oro/downloads/>

The AntRunner add-on for JBuilder is located at:

<http://antrunner.sourceforge.net/>



Contact Information

David Read

david.read@blueslate.net

(518) 443-0430 ext.108

Paul Evans

paul.evans@blueslate.net

(518) 443-0430 ext.111



Q&A



JavaOneSM

Sun's 2002 Worldwide Java Developer Conference™

BEYOND
BOUNDARIES