Hello, I'm Steve Loughran.

I'm here to talk about Ant1.7. I'm also going to cover stuff that has been in Ant1.6 for a while, but not been broadly noticed.

This is not an introductory talk. If you are new to Ant, this talk will probably scare you off. Leave the room now!

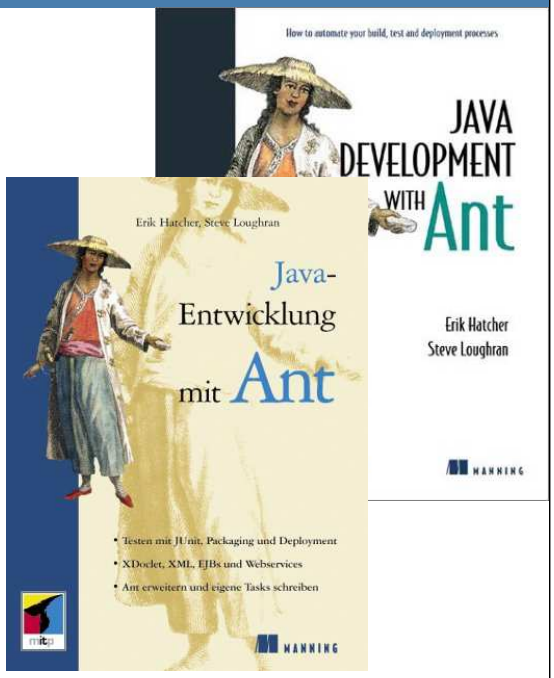
Me:

Researcher at HP Laboratories on *Grid-Scale Deployment*

Ant team member

Co-author of *Java Development with Ant*

Writing the 2nd "Ant1.7" edition; due late 2005



All about me. By day: a research scientist at HPLabs, Bristol, UK. Out of hours Ant dev team (it's a long story), and co-author of Java Dev with Ant. I am now busy writing the second edition, which is due later this year, and targeting Ant1.7

State of the Nation: Ant

- Standard way to build Java code. (90%*)
- 100% IDE coverage
- Foundation for testing, automated builds
- So successful, MSBuild is a strategic product
- Only Java alternative is Maven, our sibling tool

Ant is Java's primary build tool.

* syscon survey; possible participant bias

We are the build tool. That gives us responsibility more than power.

Limits of Ant: scale & change

1. *Defaults*
2. *Custom tasks*
3. *Big projects*
4. *Libraries*
5. *Deployment*

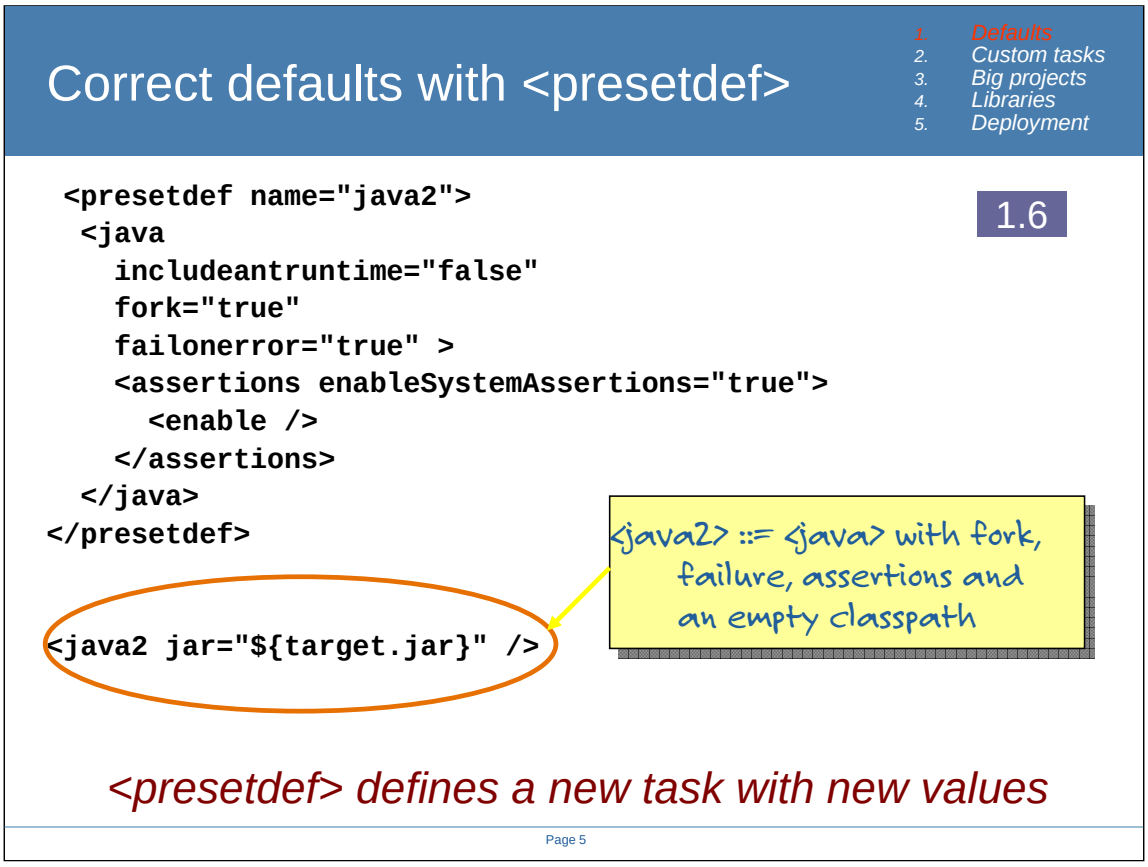
- Bad defaults on <javac>, <java>, <exec>, …
- Custom tasks generally written in Java (=extra work, complexity)
- Big projects get complex fast
- Library management
- Not that good at workflow or deployment

How are we going to fix these?

So where is Ant bad?

If you've been in a big project, you'll know. If not, ask the maven team for a list :)

It comes down to the classic problem of scale. If one project succeeds, you try harder the following time round.



the trouble with bad defaults is that you have to remember to correct them everywhere you use them.

With <presetdef> you can declare a new task, which declares new default values for an existing task (or presetdef).

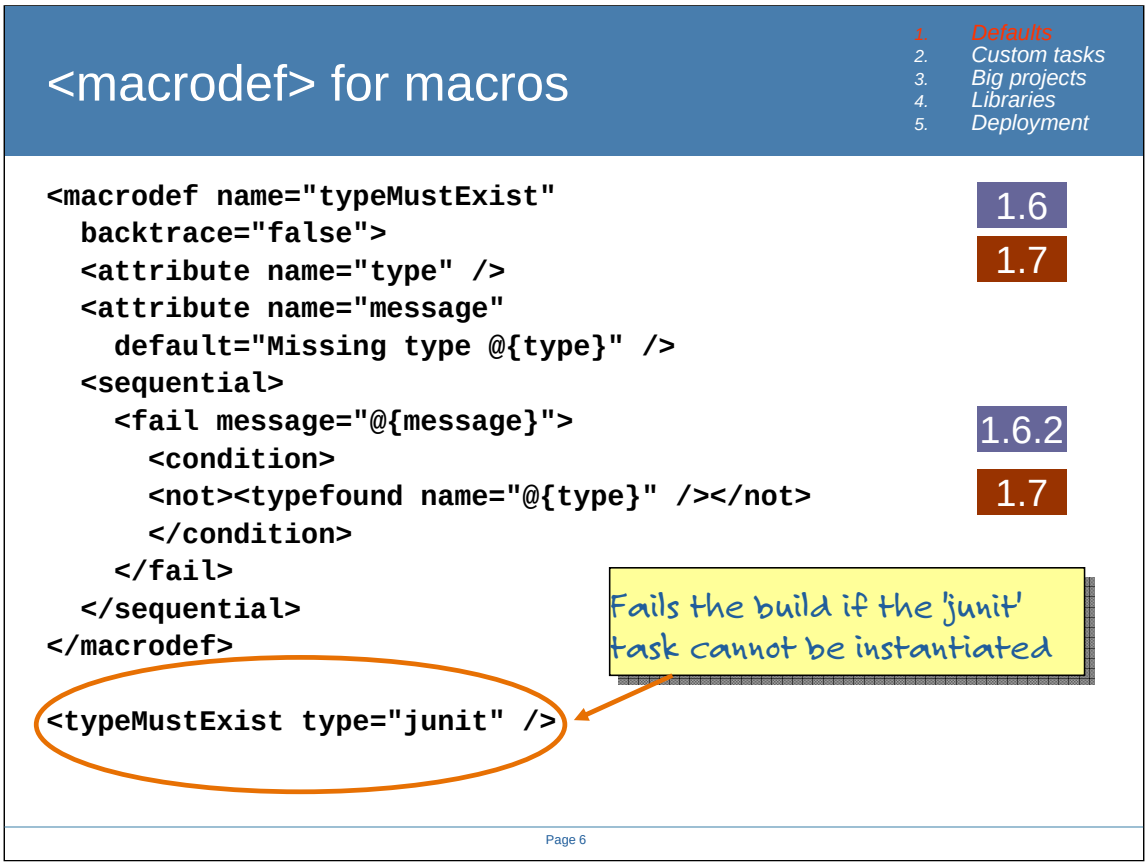
DO NOT REDEFINE EXISTING TASKS.

That is the C++ evil -redefining existing behaviour.

1. you can't copy and paste stuff into different builds without them behaving slightly differently, which is hard to track down. Not having a task called java2 is obvious, but having java's failonerror change is not

2. any code that goes (Java)Project.createTask("java") crashes at runtime. I stripped all of them out of our codebase, but third party tasks are vulnerable.

Instead give tasks a new name/prefix/namespace and use the new name



Preset sets defaults; <macrodef> lets you define new tasks. This page shows a lot of new features

1. Macrodef: new for ant1.6
2. Backtrace: a 1.7 flag that says 'don't trace into the macro on a failure'. It makes a macro look more like a real task -leave it off during debugging, turn it on when finished.
3. Nested conditions in a <fail> task. This is ideal for macros, but useful everywhere.
4. <typefound> a condition that tests that a task is declared and can be instantiated.

Hello, Scripting!

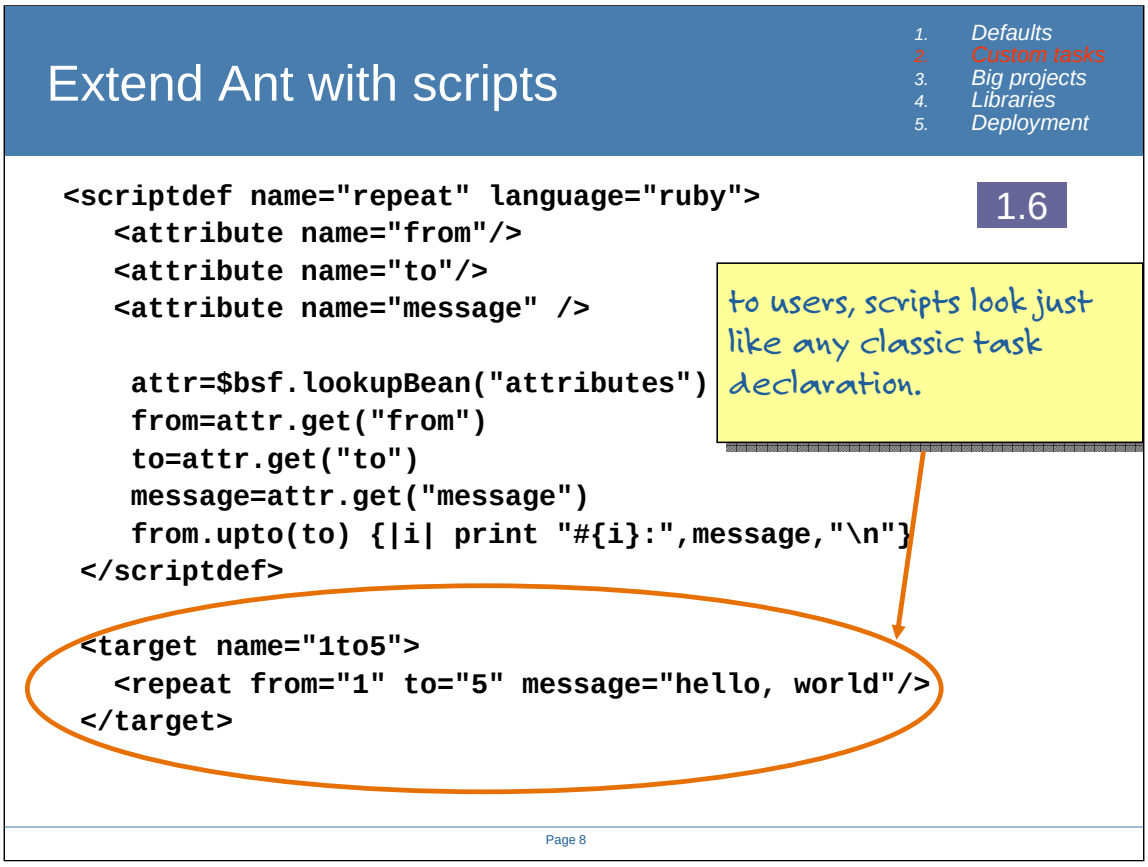
1. Defaults
2. Custom tasks
3. Big projects
4. Libraries
5. Deployment

<script>	Inline script	1.1?
<scriptdef>	Define a task	1.6
<scriptfilter>	Transform chars in a filter chain	1.6
<scriptcondition>	Implement a condition	1.7
<scriptselector>	Select files	1.7
<scriptmapper>	Map filenames	1.7

BSH, Jython, Groovy, JavaScript, JRuby, Rexx …

Look at the <script> task doc for all the main details on scripting within Ant.

These are all the ways that Ant1.7 lets you write scripts (inline or in files) to do bits of your build. Its easier than writing java code, but relies on the relevant JARs being on the classpath. As scripts get used more, this may become less of an issue.



This is an task declared inline in ruby.

You can also refer to declarations in files, which would seem a better approach for anything complex. To callers (with the right library), a scripted task behaves just like a 'legacy' java task.

Use <import> for project re-use

1. Defaults
2. Custom tasks
3. Big projects
4. Libraries
5. Deployment

```
<import file="${home}/common.xml">
```

1.6

- Imported file is appended to the current document *not where declared*
- Special target handling logic
- Filename in `ant.file.`*`project-name`*

Ant-specific replacement for XML &includes;

Ant1.6 added something new, a way to import existing build files. This is an Ant-specific replacement for XML &includes;.

Imported target logic

1. Defaults
2. Custom tasks
3. Big projects
4. Libraries
5. Deployment

main.xml

```
<project name="main">
<import file="import.xml" />

<target name="a">
  <echo>main#a</echo>
</target>



<target name="c"
     depends="a,1" />

</project>
```

import.xml

```
<project name="import">

<target name="a">
  <echo>import.a</echo>
</target>

<target name="1" depends="a" />

<target name="2"
  depends="import.1,import.a"
  />
</project>
```

Use the prefixed name to prevent overrides

This is the Ant1.7 import logic

1. Import files are effectively appended to the tail of the build file (it's a depth-first import, BTW)
2. Every imported target is prefixed with its project name, so can be explicitly addressed
3. If there is not yet a target with the unprefixed name in the project, it is also imported with the unprefixed name

Ant 1.6 only renamed targets if they clashed with one in the base class. Because Ant1.7 prefixes every file, you can use prefixed references to prevent overrides.

Effective <import>

1. *Defaults*
2. *Custom tasks*
3. *Big projects*
4. *Libraries*
5. *Deployment*

1. Use the task with care and caution
2. Give every project a unique `name` attribute
3. Have a common.xml for common stuff
4. In the `<import>`, define `<presetdef>`, `<macrodef>` tasks, scripts, shared types
5. Use prefixed dependencies to avoid overrides
6. Remember imports are imported to the tail

I think we are all still learning how to use this properly. I have used it in very large projects, and while it helped us to scale, it was still fairly brittle.

antlib: automatic task binding

1. Defaults
2. Custom tasks
3. Big projects
4. Libraries
5. Deployment

```
<target xmlns:t1="antlib:org.antbook.tasks">
  <delete dir="${build.dir}" />
  <t1:delete user="steve" />
</target>
```

1.6

- Since Ant1.6; <typedef> has let you declare the URI of an XML namespace, into which the tasks live
- If the uri begins antlib: Ant autoloads a descriptor, here `org/antbook/tasks/antlib.xml`
- Antlib descriptors can declare: tasks, types, presetdefs, macrodefs, scripted tasks.
- Useful with `-lib` on the command line

- This has been in ant1.6, but only now are people taking use of it
- This means they have to embrace XML namespaces :(
- But on a positive note, antlib provides an extensibility point, which, through the miracle that is XML namespaces, is very unique.

Library management

1. Defaults
2. Custom tasks
3. Big projects
4. Libraries
5. Deployment

Ant1.7 does *not* provide any built in library management tasks.

Maven2 <artifact:> tasks are the “Apache” means of retrieving libraries/managing dependencies

http://maven.apache.org/maven2/ant-tasks.html

Consider also *Ivy*

http://jayasoft.org/ivy

CVS_HEAD Ant had a <libraries> task that was layout agnostic, protocol independent. But it was complex, and didn’t work perfectly. Rather than get it right, we have chose

—works today, single codebase, can use .pom dependency rules; Maybe even makes it easier to move to maven…

There is also Ivy; an Ant extension which is not from Apache, but can also work with the Ivy repository. Arguably it is more mature.

Maven2 <artifacts:dependencies>

1. Defaults
2. Custom tasks
3. Big projects
4. Libraries
5. Deployment

```
<property file="libraries.properties"/>

<artifact:dependencies pathID="xerces.classpath"
 xmlns:artifact="antlib:org.apache.maven.artifact.ant"
  >
  <dependency groupID="xerces"
    artifactID="xercesImpl"
    version="${xerces.version}"/>
  <dependency groupID="xerces"
    artifactID="xmlParserAPIs"
    version="${xerces.version}"/>
</artifact:dependencies>

<javac classpathref="xerces.classpath">
  …
</javac>
```

define versions

path to create

declare artifacts (fetched on demand)

use the path in tasks

1.6

Page 14

This is the Maven2 task. It lves in its own antlib URI: antlib:org.apache.maven.artifact.ant
This is all under development; here we are using Maven2.0 Alpha 2

We are importing and using Xerces, by declaring a dependency on two Xerces artifacts, the API and the parser.

The task actually does transitive dependencies, which are powerful, but a bit troublesome. Sometimes you get more than you can expect. A solution to this is planned.

Repositories

1. Defaults
2. Custom tasks
3. Big projects
4. Libraries
5. Deployment

- http://ibiblio.org/maven2 is the primary server
- Mirrors round the world
- Only does OSS libraries, not those with click-through licenses (like Sun's)
- Open security issues
- Intranet projects may want to have their own repository for private and Sun libraries:

```
<artifact:remoteRepository
   id="internal.repository"
   url="http://internal/maven2"/>
```


The ibiblio.org lib is the centre of the universe, but you can tell the tasks to fetch from any of a list of sources. Future versions of the tasks may use mirrors automatically, but don't bet on it. Better to locate your local mirror and list it first.

Using internal repositories is good for security. You can control which versions of libraries it is possible to depend on, and only release versions you are happy with. You could even sign the jars, though that has other consequences;

Embrace Repositories

1. Defaults
2. Custom tasks
3. Big projects
4. Libraries
5. Deployment

- Filename of JAR files should include the version
- Get used to signing JARs: <signjar>
- Provide a Maven POM for every JAR created
- Use the <artifact:dependencies> task to set up the classpaths
- Define the versions of libraries you need in a properties file.
- OSS developers: publish to the repository! (file request on maven JIRA)

From now on, all JAR files should have a version marker: artifact-1.6.jar

Write a POM file too. Even a simple one will suffice.

Use the dependencies task, but provide an override point of a properties file. This is essential for letting people override your decisions.

Finally, all OSS developers should publish to the repository. The Ant/Maven teams will help you do this securely.

Deployment

1. Defaults
2. Custom tasks
3. Big projects
4. Libraries
5. Deployment

Stop using Ant as the deployment engine!

- No complex workflow (concurrency, failure)
- No automatic roll-back/undeploy
- No integrated monitoring/liveness
- No remote process management
- No security
- Little dynamic application configuration
- Ant was never designed to run for days

Use Ant to start deployment, not describe it

Steve's opinions only

Little bit of controversy here.

Ant is better than deployment than its predecessors. But that only gives people ideas, and they start using ant as generic workflow engine (c.f. gridant), or for complex deployments (c.f chapter "advanced deployment" in java dev with ant).

But it isnt really the right tool for the job. It cannot handle failure, rollback, or coordinated deployment to multiple systems.

SmartFrog: a deployment system

1. *Defaults*
2. *Custom tasks*
3. *Big projects*
4. *Libraries*
5. *Deployment*

- Deployment counterpart to Ant
- Ant tasks to initiate deployment across machines
- Integrated configuration, undeploy, liveness checks
- Template-driven
- LPGL Java library; sourceforge hosted.
- HPLabs research project;

http://smartfrog.org/

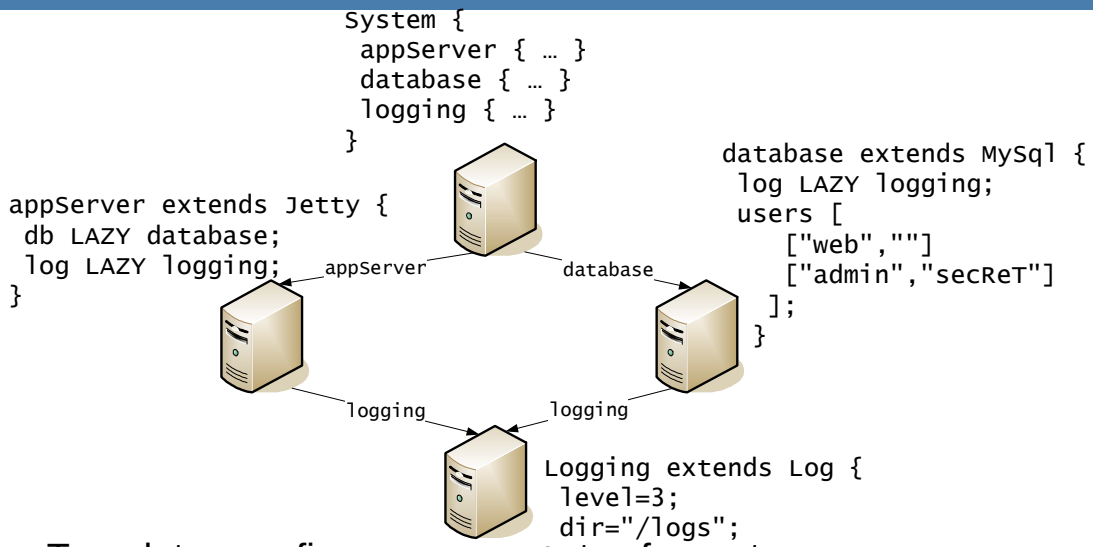
This is a personal opinion. I am on the team that works on this, so am biased. Other ant developers have the right to have their own opinions :)

This is what we use for our big projects. It's a deployment system.

Now, a deployment system is kind of a new concept -this is research technology after all- but its interesting. A tool focused on deployment.

Declarative Distributed Deployment

1. Defaults
2. Custom tasks
3. Big projects
4. Libraries
5. Deployment



- Templates configure components of a system
- Cross-binding across templates, and hence machines
- Manages deploy, undeploy and liveness

For anyone who has used <presetdef> in ant, you have a hint of what is done here.

Every template you declare is just a set of name-value pairs, nested children or cross-references to other templates

The template you actually instantiate must describe the application you are deploying.

Every instantiated template is bound to a "component" class instantiated on a local or remote host

These components configure and deploy the real systems,

Monitor their health (throw exceptions on failure, handling policy is the choice of the parent)

Undeploy the app when terminated.

Deploy from the build

1. Defaults
2. Custom tasks
3. Big projects
4. Libraries
5. Deployment

build.xml

```
<sf:deploy classpathref="run.classpath"
  host="${deploy.host}" >
 <codebase url="${codebase}"/>
 <application name="alpine">
sfCodeBase PROPERTY org.sma
#include "/alpine.sf"
  </application>
</sf:deploy>
```

alpine.sf

```
server extends AlpineServerOnJetty {
echo extends EchoEndpoint {
 path "/echo/";
 servlet LAZY servlets:alpineServlet;
 }
ping extends LivenessPage {
 enabled PARENT:liveness;
 host LAZY PARENT:servlets:ipaddr;
 port PARENT:port;
 page LAZY endpoint:absolutePath;
}
liveness true;
}
```


This is a deployment of

1. Jetty web server
2. A servlet "alpineServlet" on jetty that supports SOAP endpoints
3. An endpoint that echos posted soap request. It is bound to the alpineServlet -at deploy time it tells the servlet to map request to its path to its class
4. A liveness page that checks that a GET of the echo endpoint returns without an error

It does so much in so few lines, because most of the stuff is inherited from other deployment descriptors.

Java1.5 support

- New source="1.5", target="1.5" switches for <javac> with compiler workarounds (sigh)
- Fixed <rmic> defaults to work again
- <apt> to process annotations
- <isreachable> condition to probe hosts
  ex: dynamic use of proxy on a laptop

```
<fail message="host missing" status="-3">
  <condition property="proxy.enabled">
    <isreachable host="${host}"/>
  </condition>
</fail>
```

1.7

Page 21

•Some fixes for the inevitable changes of the JDK, so your builds work as before. We don't know why Sun keep changing the defaults, but we have switched them back so that build files that work on older systems behave consistently on java1.5

•isreachable is a network test that isnt guaranteed to work through firewalls; use it for local probing, not checking for long-distance connectivity.

•Note the use of a nested condition in the failure and an exit code. These are (ant1.6.2+) features.

Apt: annotation processor tool

- Compile Java1.5 annotations in source, 1.7
- Invoke annotation factories (slow)
- Can generate new source or compile to .class

```
<apt srcdir="${src}"
  destdir="${classes2.dir}"
  compile="true"
  fork="true"
  factory="DistributedAnnotationFactory"
  preprocessdir="${preprocess.dir}">
  <factorypath path="${classes.dir}" />
  <option name="build.dir" value="${build.dir}" />
</apt>
```


- I say could. I don't use it myself, yet.
- If/When jikes adds 1.5 support, life will be better.

What will an Ant1.7 build file look like?

1. Common build files across a big project
2. <import> into dependent files
3. Antlib declarations
4. use of <presetdef>/<macrodef>/scripts
5. (maven2 dependency management)
6. (smartfrog deployment)

So here are all the features to deal with scale. What will a build system that uses this stuff look like?

When will Ant1.7 ship?

Ask Stefan, Jan, Stephane...

Prerequisites:

- ResourceCollection retrofitting
- Locals in <macrodef>
- More M2 integration
- Documentation rework (automation)
- Move to SVN
- (Get more of my book done)

...later this year. Probably.

Big question: when will ant1.7 ship? Answer: after we have full integrated some major changes.

One interesting complexity is that there is now pressure from the IDEs to align with their schedule. This isnt explicit "you must ship", but more "if you have it done by oct 1 we can ship with it in our next release". I think this is a bit dangerous to ship unstable there. We are too core to sacrifice stability for short deadlines.

Moving to Ant1.7

- Get the Ant 1.7 Beta -*when available!*
- Use with Ant1.6.x-compatible builds
  - <import> <presetdef>
  - Maven2 <artifact:dependencies>
  - Explore SmartFrog deployment
- Java1.5 users: adopt Ant1.7 as soon as you can
- Write more tests!

If Ant1.7 is coming, how do you get ready for it?

1. Look at the leading edge stuff in Ant1.6: import, presetdef, macrodef, scriptdef, and use where appropriate
2. Pull down the maven2 tasks to work with their library management. Ant1.7 supports these slightly better.
3. Explore SmartFrog for deployment. Again, Ant1.7 works better here, but ant1.6.3 still functions.
4. Write more tests. Always.

Questions?