# Apache Ant and DevOps Practices

The practice of DevOps frequently requires assembling collections of modified files to be deployed to remote servers. There are many tools to handle the “push” operation to remote servers, and to maintain the concordance between the repository of source documents and the transformed representations on remote servers. Chef and Puppet are popular examples.

What is *not* handled well by these tools is the *actual assembly* and *disassembly* of the deployable files at each end of the deployment. Software development is a mature practice with similar requirements and a long history of tools to “build” software components. This parallels DevOps practice of building deployable components. We are going to examine using one such tool, ANT, in a DevOps scenario.

We first begin with a little background on build tools and why ANT is a good choice. If you wish to skip ahead to the code, look below for the title “**ANT Example**”. The files for the example are located in the DemoDev repository (see references #1 and #3 below.)

## Build Tool History

In the beginning, there was command-line scripting combined with custom programs created for unique build situations (I was there and I remember.) Then MAKE came along. Unfortunately, MAKE was optimized to compile C code and did a poor job of dealing with the slow Javac compiler (see <https://stackoverflow.com/questions/2209827/why-is-no-one-using-make-for-java>.) In addition, MAKE represented actions to resolve dependencies as parameterized native executable invocations, and are thus not OS independent. IDE products sometimes use MAKE, but they hide that use from users. Many developers relied on IDE driven builds that were difficult to automate.

Apache ANT was created for Java builds and used the Javac compiler to handle incremental builds of Java files. ANT is basically an OS independent scripting language expressed using XML. ANT runs on all systems with Java, and *it runs the same way!* ANT is extensible using “tasks”. Ivy was later added to ANT to provide dependency management. Widespread ANT use resulted in complex build scripts and made poor build engineers having to re-learn each build as a special case.

There was a movement for ***Convention over Configuration***, first introduced with Rails, to simplify complex software creation (see <https://en.wikipedia.org/wiki/Convention_over_configuration>.) As part of this movement, Apache Maven was created to take advantage of strict convention. Everything in Maven is a convention. Directory layouts, the build cycle, and nested project structure are all proscribed. In summary, Maven extends ANT capabilities by providing dependency management, standard project layout and project management (Phases and Goals.) Maven worked well and its use is widespread, but Maven makes deviation from the “convention” difficult. Complex non-conforming Maven builds are difficult to understand.

Many felt Maven went too far, and so Gradle was created to allow “convention over configuration” but make it easier to express complex builds. Gradle expresses build steps as “tasks” using a DSL (Domain Specific Language) based on the JVM language Groovy, and recently added Kotlin. Plugins provide the functionality for tasks. Like Maven, there are standard project layouts, but these can easily be altered as needed. Gradle offers greater flexibility over other build tools when resolving version conflicts and managing transitive dependencies. In addition, Gradle accelerates builds through sophisticated incremental compilation and build caching.

## Why ANT

Most DevOps tasks do not require a compilation step, where the target file (object code) is older than the source file (Java) and time-consuming processing is needed to transform source to object. DevOps tasks have little dependency resolution. Primarily, in DevOps we transform source collections of files into target collections of files. The transformation step usually involves reading text files, modifying the files, and compressing them into an Achieves.

BASH and similar scripts were long ago abandoned by build engineers, and for good reasons. There are no standard versions of any shell scripting language . . . not even BASH. BASH scripts have very limited IDE support, offer no runtime error protection, make modularization difficult, and typically invoke native executables that vary of OS. Sadly, they offer no built-in target-dependency specification to ease construction specification.

As a concrete example of verbosity, the ANT BASH startup file for Nix systems has 375 lines; the Windows batch startup has 93 lines. Bash modularization practices are poor. Even with the “source” command, BASH coders tend to write monoliths. To restate some bumper-sticker wisdom: friends don’t let friends use BASH!

MAKE can perform many of these DevOps tasks, but MAKE is not system independent. There are many versions of MAKE, and the MAKE dependency actions are specific to the operating system running MAKE and the version of MAKE. For example, a Linux MAKE invoking a C compiler will not work on a Windows system with a different C compiler.

ANT offers all of the features of MAKE, but is system-independent, and includes special capabilities extending the basic MAKE targets-and-dependencies, and many task definitions. ANT is:

* Easily extensible by multiple mechanisms.
* Well documented, well supported in the community.
* Has excellent IDE support.

Finally, ANT is easily installed in many environments, has a small footprint, and only requires a version of Java to be accessible. ANT is so *historic* that there is a version of ANT for *any* version of Java.

Maven and Gradle are more capable than ANT, but they are also more complex than needed for most DevOps tasks. Both Maven and Gradle involve significant learning curves, and ANT is simpler to deploy. There are several references to DevOps using ANT (Google “DevOps” and “Apache ANT”.) These criteria lead us to consider ANT in a DevOps use case.

## ANT Example

We have to generate a collection of processed files for multiple target environments, adjusting the deliverables based on the target. We have three kinds of file processing:

1. Direct copy of files.
2. Interpolation of embedded expansions.
3. Generation based on embedded directives.
4. Condition some of the steps (1-3) depending on the target environment

Direct file copy will take files from an “immutable” source and copy them to the target directory. *Interpolation* will use template files, with embedded directives, and process them with ANT components, placing the result into the target directory. Generation is a more complex use case explained in detail below. We will create different targets for production then test or staging.

## Interpolation

Interpolation is the process of replacing string-based expansion requests in a template by a supplied dictionary listing the keyword and the string used to replace the request. We can illustrate this process with an example:

|  |  |
| --- | --- |
| **Collaborators** | **Content** |
| **Dictionary** | **; shared\_defs.properties**  **GEN\_source=Thor**  **GEN\_target=No\_Target**  **GEN\_memory=256M**  **GEN\_tasks=10** |
| + |  |
| **Template** | **Generated System Provenance**  **Source : ${GEN\_source}**  **Target : ${GEN\_target}**  **Memory : ${GEN\_memory}**  **Tasks : ${GEN\_tasks}**  **Generation Details**  **Master Properties : ${GEN\_master}**  **Override Properties: ${GEN\_override}** |
| Generator = |  |
| **Interpolated** | **Generated System Provenance**  **Source : Thor**  **Target : No\_Target**  **Memory : 256M**  **Tasks : 10**  **Generation Details**  **Master Properties : shared\_defs.properties**  **Override Properties: target\_defs.properties** |

## Generation

The first problem with Interpolation is a collision between processing tools. Chef can use interpolation similar to ANT for example. Hive uses interpolated variables for parameterization as well. Suppose a template needs to be preprocessed by our ANT tool, and then reprocessed by Chef. There could be a collision between variable expansion requests; they all look like ***“${varname}***”. Here is a simple example of this collision avoidance for Hive in action:

|  |  |
| --- | --- |
| **Collaborators** | **Content** |
| **Dictionary** | **; Hive overrides**  **GEN\_env\_prefix=tst**  **GEN\_src\_tbl=Test\_Source\_Table** |
| + |  |
| **Template** | **; hive\_vars.hql**  **;**  **; Defines Hive variables used by the Hive query. An overview is found at**  **; https://cwiki.apache.org/confluence/display/Hive/LanguageManual+VariableSubstitution.**  **;**  **<#-- Hive variable definition generated by Freemarker -->**  **; Generation date: Run date: @@{.now}**  **set tbl\_prefix=@@{GEN\_env\_prefix};**  **set base\_source\_table\_name=@@{GEN\_src\_tbl};**  **set source\_table=${tbl\_prefix}\_${base\_source\_table\_name};**  **set base\_target\_table\_name=@@{GEN\_tgt\_tbl};**  **set target\_table=${tbl\_prefix}\_${base\_target\_table\_name};** |
| Generator = |  |
| **Interpolated** | **; hive\_vars.hql**  **;**  **; Defines Hive variables used by the Hive query. A overview is found at**  **; https://cwiki.apache.org/confluence/display/Hive/LanguageManual+VariableSubstitution.**  **;**  **; Generation date: Run date: Jul 15, 2019 4:02:17 PM**  **set tbl\_prefix=tst;**  **set base\_source\_table\_name=Test\_Source\_Table;**  **set source\_table=${tbl\_prefix}\_${base\_source\_table\_name};**  **set base\_target\_table\_name=Test\_Target\_Table;**  **set target\_table=${tbl\_prefix}\_${base\_target\_table\_name};** |

The generator uses an alternate form of interpolation request to avoid collision (“***@@{varname}***”.). Notice that the template injects the run date into the generated output. Sometimes our Generation use case is more complicated than simple string substitution in the template.

We use a Freemarker based templating engine from the J2EE applications world that generates a view (dynamic versions of JSP.) Freemarker is a popular, mature, and well documented Java templating engine we have used to extend ANT.

## References

Supporting material to understand the ANT example:

1. Ant Project: <https://ant.apache.org/>.
2. Ant On-line user manual: <https://ant.apache.org/manual/>.
3. Sample Tutorial: <https://www.vogella.com/tutorials/ApacheAnt/article.html>.
4. Sample ANT DevOps interview questions: <http://www.scmgalaxy.com/tutorials/apache-ant-interview-questions/>.
5. Eclipse Ant integration example: <https://community.synopsys.com/s/article/Setting-up-ant-build-for-Java-Workspace-in-Eclipse>.

*Freemarker Templating Engine Support*

1. Freemarker site: <https://freemarker.apache.org/>.
2. Freemarker Java development tutorial-1: <http://zetcode.com/java/freemarker/>.
3. Freemarker tutorial-2: <https://www.vogella.com/tutorials/FreeMarker/article.html>.
4. Freemarker manual: <https://freemarker.apache.org/docs/index.html>.

*DemoDev Repository References*

1. The DemoDev repository: <https://github.com/DonaldET/DemoDev>.
2. The template generation utility (*DemoGenerator*): <https://github.com/DonaldET/DemoDev/tree/master/dev-topics-generationutils>.
3. This example: <https://github.com/DonaldET/DemoDev/tree/master/dev-topics-generationutils/example>.

## Appendix A – TextSourceGeneratorRunner command line interface

-defaultContext VAL : A file with properties definitions to use as the

[REQUIRED] primary context

-overrideContextList VAL : A comma separated list file names of properties

definitions to augment the primary context

-srcDir VAL : Defines base directory for all text templates

(source) files

-templateList VAL : A comma separated list source file names, found

[REQUIRED] under srcDir, of templates to process

-dstDir VAL : Defines base target directory for all generated

[REQUIRED] text files

-generatedFileList VAL : A comma separated list file names, found under

[REQUIRED] dstDir, of generated files from processing

templates and contexts

Usage: -defaultContext VAL -dstDir VAL -generatedFileList VAL -overrideContextList VAL \

-srcDir VAL -templateList VAL

Ω