# 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 example detail, just look below for the title “**ANT DevOps Example**”. The files for the example are located in the DemoDev repository (see *DemoDev Repository References* below.) The example is system independent and does not require an IDE. Install Java, ANT, download the example, rename the Jar file, and you are ready to run (or, as Al Bundy would say: “*Let’s Rock!*”)

## 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 parsing is needed to transform source code to object byte code. DevOps tasks have simple dependency resolution compared to complex application building. Primarily, in DevOps we transform source collections of files into target collections of files for deployment. The transformation step usually involves reading text files, modifying those files, and compressing them into an Archieves.

BASH and similar scripting environments were long ago abandoned by build engineers, and for many 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 BASH verbosity, the ANT BASH startup file for Nix systems has 375 lines; the equivalent Windows batch startup has 93 lines. Bash modularization practices are poor, in spite of the fact that, while awkward, BASH can be modularized. 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. ANT includes many capabilities extending the basic MAKE targets-and-dependencies, and many task definitions. ANT is:

* Well suited to for modularization.
* 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 for complex build tasks, but they are also more complex than needed for most DevOps tasks. Both Maven and Gradle involve significant learning curves, and ANT is much simpler to deploy. There are several posted guides discussing DevOps ANT use (Google “DevOps” and “Apache ANT”.) These criteria lead us to consider ANT in a DevOps use case.

## ANT DevOps Example

For our example, we have to generate a collection of processed files for deployment to multiple target environments, adjusting the deliverables based on the target. We have four kinds of file processing needed to create our target:

1. Direct copy of files (immutable.)
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 deployment targets for production verses test or staging.

## Interpolation

Interpolation is the process of replacing string-based expansion requests in a template. The replacement string is supplied by a dictionary listing the keyword and the replacement string. We 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 simple interpolation, as implemented by ANT and may other tools, is a collision between template processing tools. Chef can use interpolation similar to ANT for example, using “***${varname}***”as an interpolation request. Hive uses interpolated variables for parameterization as well. Suppose a template needs to be preprocessed by our ANT tool, and then reprocessed by Hive at runtime. There could be a collision between variable expansion requests; they all look like “***${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 in this example uses an alternate form of interpolation request to avoid collision (“***@@{varname}***”.). Notice that the template also 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. Freemarker generates a view from a template and a data model (think a dynamic versions of JSP). Freemarker is a popular, mature, and well documented Java templating engine we have used to extend ANT capabilities. The references offer material on understanding Freemarker’s capabilities.

## The ANT Script

Our ANT deployment example generates different content in files, and dynamically creates deployable artifacts from templates and properties files. See *DemoDev Repository References* below to obtain the source for this example.

As discussed earlier, ANT provides modularization tools that allow build scripts to be divided into “components” that are reusable. You INCLUDE or IMPORT the component scripts into the main **build.xml** script. Work is accomplished by invoking *targets*. Targets are the unit of work similar to the MAKE utility that performs actions, and targets may be invoked with ANTCALL task.

Let’s review the user-visible structure of our ANT build script (**build.xml**) using the –p option:

**D:\GitHub\DemoDev\dev-topics-generationutils\example\builder>ant -p**

**Buildfile: D:\GitHub\DemoDev\dev-topics-generationutils\example\builder\build.xml**

**[echo] ........**

**[echo] ANT DevOps example, includes Generation**

**[echo] ........**

**Build Deployment Artifacts for multiple targets**

**Main targets:**

**build orchestrate target requests**

**check\_env test if production target**

**clean Remove previously generated files**

**expandTokens Use ANT to replace tokens**

**gen\_init Set default properties for caller**

**generateCommon Generate common files**

**generateFromTemplate Use Freemarker template to generate values**

**generateProdOnly Generate production only files**

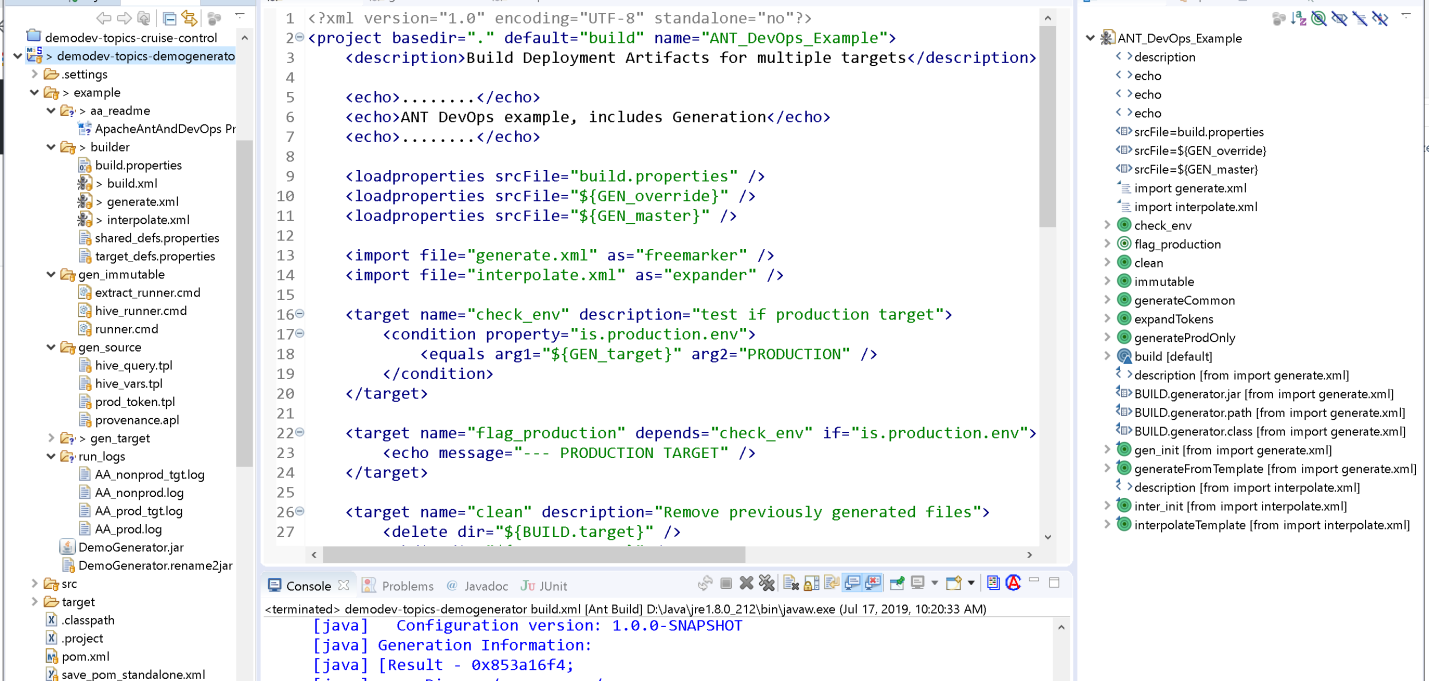
**immutable initialize target directories adding immutable files**

**inter\_init Set default properties for caller**

**interpolateTemplate Use ANT to replace tokens**

**Default target: build**

The targets perform the work outlined in the descriptions given in the listing above. The in the Eclipse IDE looks like this:



Our project structure is displayed on the left, the top center has the XML code for **build.xml**, the right-hand panel displays the target structure, and the bottom center panel shows the console output for an ANT run executed within the IDE.

Reviewing the **build.xml** file and its execution logs reveals these attributes:

1. Targets are orchestrated by mentioning dependencies, ANT works out the execution order.
2. A target’s dependencies are executed before a target is executed (recursively.)
3. A target required my multiple targets is only executed ONCE.
4. Targets are conditionally executed using the “if=” attribute.
5. Targets are multiply executed with parameters using ANTCALL.

We have created separate “modules” for the interpolation and generation tasks that are imported into the main build script (**build.xml**). The project structure shown in the left panel has our build scripts in folder ***builder***, our source template files in ***gen\_source***, our immutable files in ***gen\_immutable***, and the resulting processed files are placed in ***gen\_target***. We use two kinds of template files: ANT with extension *“.apl*” and Freemarker (generation) with extension *“.tpl*”.

The build script is parameterized by Java properties files (dictionaries with key-value pairs used in interpolation.) The property definition hierarchy for a build execution is:

1. build.properties
2. target\_defs.properties
3. shared\_defs.properties

Shared properties are the most common, and they are overridden by the target properties. Target properties are usually deployment environment specific. Finally, build properties focus on build-specific issues and override all other properties.

## 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 ANT DevOps interview questions: <http://www.scmgalaxy.com/tutorials/apache-ant-interview-questions/>.
4. Eclipse Ant integration example: <https://community.synopsys.com/s/article/Setting-up-ant-build-for-Java-Workspace-in-Eclipse>.
5. ANT tutorial: <https://www.vogella.com/tutorials/ApacheAnt/article.html>.

*ANT and Freemarker Templating Engine Support*

1. Ant Book: Ant in Action (see <http://testa.roberta.free.fr/My%20Books/Computer%20programming/Java/Manning%20-%20Ant%20in%20Action%202nd%20Edition%20(2007).pdf>.)
2. Freemarker site: <https://freemarker.apache.org/>.
3. Freemarker manual: <https://freemarker.apache.org/docs/index.html>.
4. Template Authors Guide: <https://freemarker.apache.org/docs/dgui.html>.

*Write your own Java Generator*

1. Freemarker Java development tutorial-1: <http://zetcode.com/java/freemarker/>.
2. Freemarker tutorial-2: <https://www.vogella.com/tutorials/FreeMarker/article.html>.
3. Collected Freemarker tutorials: <https://www.concretepage.com/freemarker/>.

*DemoDev Repository References*

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

## 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