# 

# Abstract - Apache Ant and DevOps Practices

We review the problem of creating custom deployable artifacts that vary by intended target environment. This paper proposes using ANT as “glue” to orchestrate artifact creation because:

* ANT provides dependency based target generation, can test conditions sequencing, operations (allowing variable outputs for each target.)
* ANT supports robust error processing, simplifying artifact creation.
* ANT incorporates standard template interpolation, but can be extended with powerful generation capabilities simplifying artifact creation.
* ANT features offers a replacement for Bash with greater reliability (robust execution) and simplified orchestration.

## Introduction

The practice of DevOps frequently requires assembling collections of modified files to deploy to remote servers. Examples are configuration files or payloads for Cloud APIs. There are many tools to handle these “push” operations to remote servers, and to maintain the concordance between the repository of source documents and the transformed representations on remote servers. Configuration Management tools such as Chef and Puppet are classic examples of configuration file management. Infrastructure-as-Code tools like Terraform, Packer, and Ansible are examples of payloads for Cloud APIs. These tools are designed following the philosophy of building one tool that does a specific job well, and loosely coupling many tools together to accomplish a deployment.

Two things are *not* handled well by these tools, using Bash, in most current approaches:

* Orchestrating the tools to create the deployable artifact, and
* The *assembly* and *disassembly* of the deployable files for variant targets (i.e., environments.)

Orchestration is performed by “glue code” that invokes these tools; all too often done with Bash. The deployed files are frequently text files that are very similar across targets. Maintenance of the target-specific variants becomes difficult as more targets are supported. We must deploy the correct variant to each target while keeping the common parts identical. The common portion of each document comprises most of the deployable artifact.

There are “build” tools that can significantly help create the target-specific deployable files (i.e., artifacts.) Software application “build” development is a mature practice with similar requirements and a long history of many tools to “build” software components. This parallels DevOps practice of building deployable components. We will examine using the build tool ANT in a DevOps scenario. ANT is well suited to DevOps target preparation. We will also review an enhanced templating approach that allows creation of complex deployable artifacts without writing “code” to dynamically create these files.

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

## Build Tool History

From the 60’s through the early 90’s, command-line scripting combined with custom programs was used for most build situations. Then MAKE was more widely adapted. Unfortunately, MAKE was optimized to compile C code and did a poor job of dealing with the slow startup of the Javac compiler (see reference 6 below.) In addition, MAKE represented actions to “resolve dependencies” as parameterized native executable invocations, and are thus not OS independent. IDE products sometimes use MAKE internally to orchestrate a build, but they hide that usage from users. Many developers in the early 90’s relied on IDE driven builds that were difficult to automate.

Apache ANT was created for Java builds and used the Javac compiler’s ability to handle incremental builds of large collections 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”, which are similar to MAKE dependencies. Ivy was later added to ANT to provide Java library dependency management. Widespread ANT use resulted in complex build scripts and made poor build engineers have to re-learn each application build as a special case.

There was a movement for ***Convention over Configuration***, first introduced with Rails, to simplify complex software creation (see reference 7.) As part of this movement, Apache Maven was created to take advantage of a strict convention approach. 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 good Java library dependency management, standard project layout and project management (Phases and Goals.) Maven worked well and is widespread, but Maven makes deviation from the “convention” difficult. Complex non-conforming Maven builds are difficult to create and understand.

Many felt Maven went too far in strict convention, 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 a time-consuming parsing is needed to transform source code to object byte code. DevOps tasks have simple dependency resolution compared to complex application building. 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 Archives.

Bash and similar scripting approaches were abandoned by build engineers because of the complexity and unreliability of the interdependent build actions. 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. Google “Bashing Bash” and you will find several blogs covering Bash coding problems. 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

In our DevOps example, the Automotive Sales department has asked us to design a Sales Query development pipeline to speed up their insights into vehicle sales. The process to make queries available to the Sales department in their UI involves generating a collection of processed files for deployment to multiple target environments. In this example, we will deploy to production, staging, and test environments. We need to adjust the deployable artifacts based on the deployment target. We have four kinds of file processing needed to create our target:

1. Direct copy of files (immutable.)
2. Simple 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, combined with property files, 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\_service=Sales Query**  **GEN\_source=Thor**  **GEN\_target=No\_Target**  **GEN\_memory=128M**  **GEN\_tasks=5**  **; Hive properties**  **GEN\_env\_prefix=None**  **GEN\_src\_tbl=VehicleSales**  **GEN\_tgt\_tbl=SalesSummary** |
| + |  |
| **Template** | **Generated System Provenance for ${GEN\_service}**  **Source : ${GEN\_source}**  **Target : ${GEN\_target}**  **Memory : ${GEN\_memory}**  **Tasks : ${GEN\_tasks}**  **Generation Details for ${GEN\_service}**  **Master Properties : ${GEN\_master}**  **Override Properties: ${GEN\_override}** |
| * Generator = |  |
| **Interpolated** | **Generated System Provenance for Sales Query**  **Source : PROD\_REPO\_URL**  **Target : PRODUCTION**  **Memory : 512M**  **Tasks : 12**  **Generation Details for Sales Query**  **Master Properties : shared\_defs.properties**  **Override Properties: target\_defs.properties**  **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. For example, Chef can use interpolation requests similar to ANT (i.e., “***${varname}***”.) Hive uses interpolated variables for runtime parameterization as well. Table names, database names, and where clause constants are command entries.

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}***”. Our generator solves that problem. Here is a simple example of this collision avoidance for Hive in action:

|  |  |
| --- | --- |
| **Collaborators** | **Content** |
| **Dictionary** | **; Hive properties**  **GEN\_env\_prefix=None**  **GEN\_src\_tbl=VehicleSales**  **GEN\_tgt\_tbl=SalesSummary**  **. . .**  **; Hive overrides**  **GEN\_env\_prefix=PRODUCTION** |
| + |  |
| **Template** | **-- hive\_vars.hql**  **--**  **-- Defines Hive variables used by the Hive query. An overview of Hive variables 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 dbname=<#if GEN\_target == "PRODUCTION">RAW\_SALES<#else>RAW\_SALES\_SAMPLE</#if>;**  **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\_source\_table\_name};**  **set prod\_brands="CHEVY", "FORD", "TOYOTA", "LEXUS", "HONDA","BMW", "VW";**  **set nonprod\_brands="CHEVY", "TOYOTA";**  **set brand\_list=<#if GEN\_target == "PRODUCTION">${prod\_brands}<#else>${nonprod\_brands}</#if>;** |
| Generator = |  |
| **Interpolated** | **-- hive\_vars.hql**  **--**  **-- Defines Hive variables used by the Hive query. An overview of Hive variables is found at**  **-- https://cwiki.apache.org/confluence/display/Hive/LanguageManual+VariableSubstitution.**  **--**  **--Generation date: Run date: Jul 18, 2019 9:42:47 PM**  **set tbl\_prefix=PRODUCTION;**  **set dbname=RAW\_SALES;**  **set base\_source\_table\_name=VehicleSales;**  **set source\_table=${tbl\_prefix}\_${base\_source\_table\_name};**  **set base\_target\_table\_name=SalesSummary;**  **set target\_table=${tbl\_prefix}\_${base\_source\_table\_name};**  **set prod\_brands="CHEVY", "FORD", "TOYOTA", "LEXUS", "HONDA", "BMW", "VW";**  **set nonprod\_brands="CHEVY", "TOYOTA";**  **set brand\_list=${prod\_brands};** |

The generator in this example uses an alternate form of interpolation request to avoid collision (i.e., “***@@{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 Build Script

Our ANT deployment example dynamically creates deployable artifacts from templates and properties files. See *DemoDev Repository References* below to obtain the files to run this example.

As discussed earlier, ANT provides modularization tools that allow build scripts to be divided into “components” that are reusable. You then INCLUDE or IMPORT the component scripts into the main **build.xml** script to use them. ANT *targets* are the unit of work similar to the MAKE utility dependencies, and targets may be invoked with ANTCALL task when parameterization is necessary.

Let’s review the reported documentation 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**

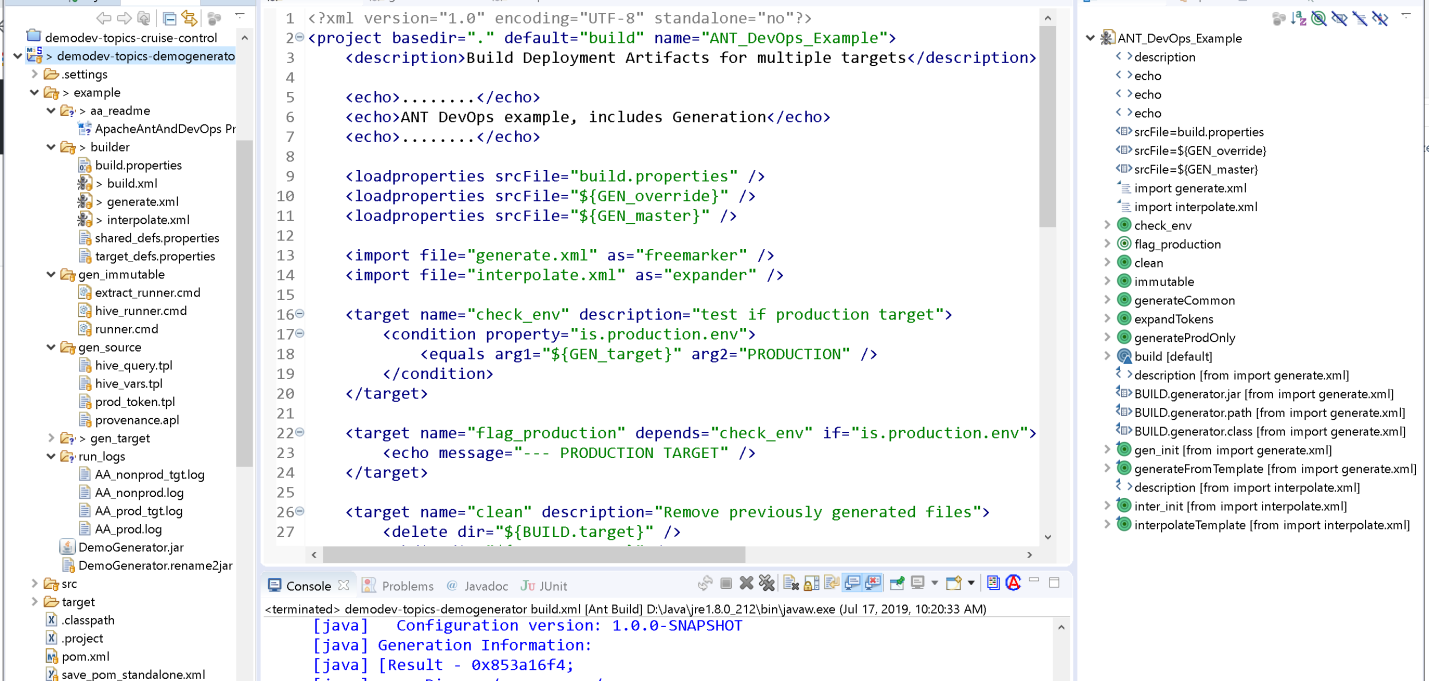
**immutable initialize target directories adding immutable files**

**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. Our **build.xml** file in the Eclipse IDE looks like this:



Our Eclipse project structure is displayed in the left panel, the top center has the XML code for **build.xml**, the right-hand panel displays the ANT 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 parameterized “modules” for the interpolation and generation tasks that are imported into the main build script (**build.xml**). The project Eclipse 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 interpolation with extension *“.apl*” and Freemarker generation with extension *“.tpl*”.

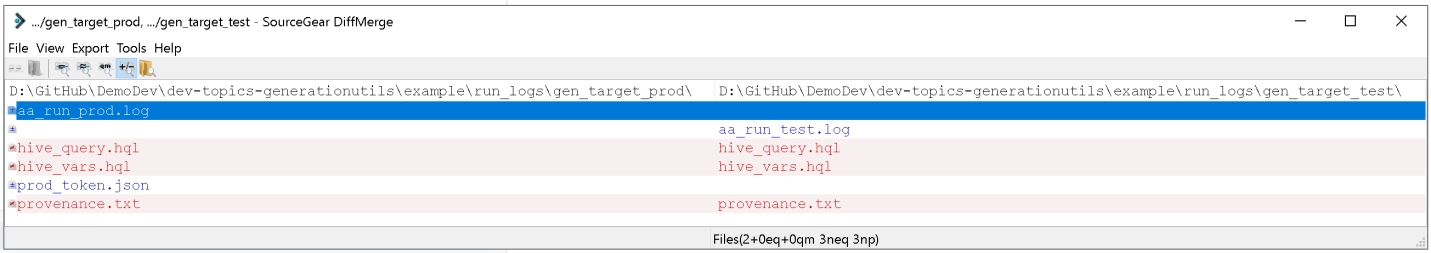
The build script is parameterized by three Java properties files (dictionaries with key-value pairs.) The property definition hierarchy for a build execution is:

1. Define *build.properties*,
2. Define *target\_defs.properties*
3. Define *shared\_defs.properties*

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

## Target Environment Differences

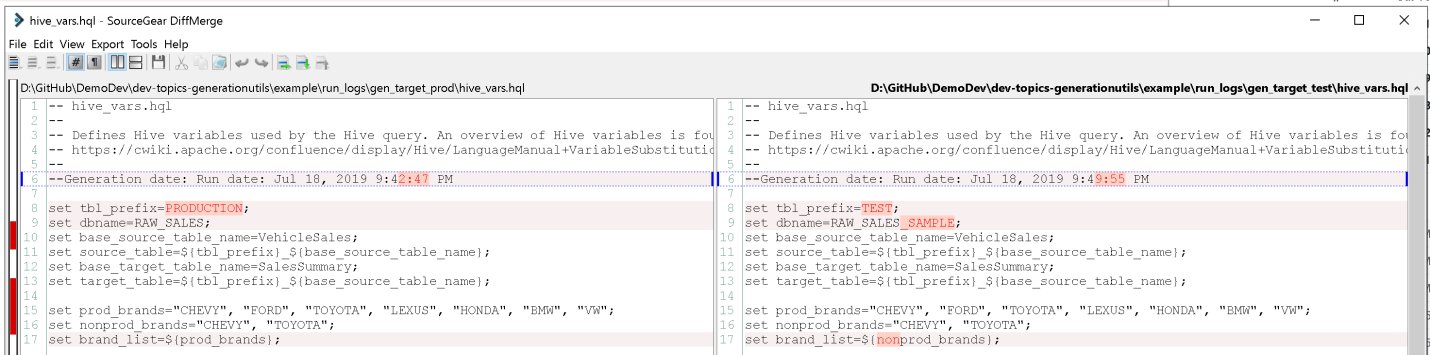
The file differences between target environments is shown in this file comparison application:



The files in red have different content. Note that the file **prod\_token.json** is only generated for the production target.

## Sample Hive Generation Differences

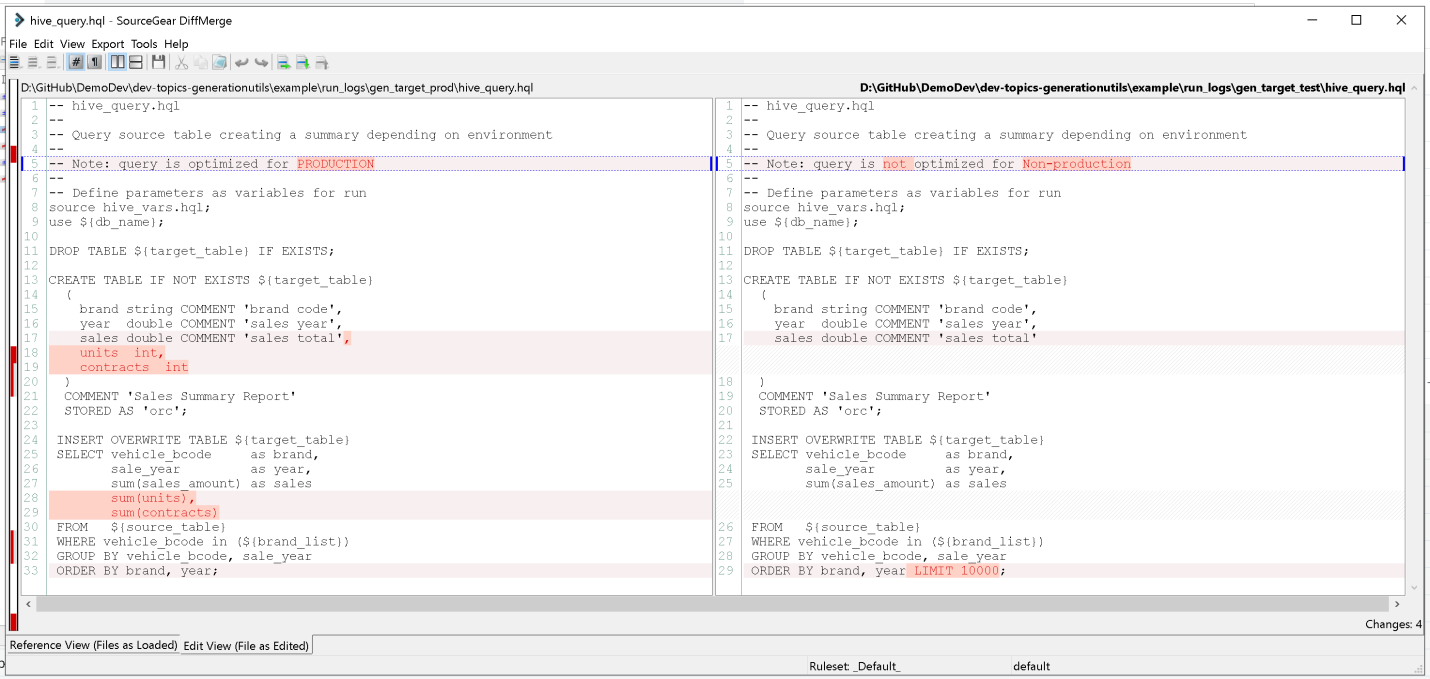
Hive uses *hive variables* supporting interpolation in the input template. We are able to use our generator, with a unique interpolation request format, to process the template interpolation requests.



As shown above, we can also inject values like the current date and time, and we can perform selective interpolation based on conditions.

## Complex Template Generation

Freemarker allows us to generate complex output simply. Here is the generated hive query file **hive\_query.hql** (note the import of **hive\_vars.hql**.):



We see that the production environment includes additional columns in the query, and the non-production environment has a limit on retrieved rows, as well as a different documentation line.

## Summary

We have seen these ANT capabilities to generate artifacts:

1. Provides dependency orchestration between “targets” for producing variable artifacts.
2. Tests arbitrary conditions and makes the test result available for sequencing, so that
3. ANT has conditional “target” execution (allowing variable outputs for each deployment scenario.)
4. Error processing is incorporated into the ANT execution (e.g, no status code checking required.)
5. Offers standard template interpolation, but
6. Is extensible (e.g., offering Templates with programmatic generation using Freemarker.)

The Freemarker generator used to extend ANT offers powerful templating capabilities that are mature, well documented, and easier to use than custom Bash code to achieve dynamic file generation.

## References

*Supporting materials 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>.
6. MAKE shortcoming reference: <https://stackoverflow.com/questions/2209827/why-is-no-one-using-make-for-java>.

*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>.
5. Template tutorial: <https://www.concretepage.com/freemarker/java-freemarker-templates-ftl-tutorial-with-html-example>.
6. Freemarker templates as used by RedHat: <https://access.redhat.com/documentation/en-us/jboss_enterprise_soa_platform/5/html/smooks_user_guide/chap-templates>.
7. Convention over Configuration: <https://en.wikipedia.org/wiki/Convention_over_configuration>.

*Write your own Java Generator*

1. Freemarker Java development tutorial-1: <http://zetcode.com/java/freemarker/>.
2. Freemarker Java usage tutorial: <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