Shell Scripting Standards

Imran Quadri Syed, 2020-02-11

Abstract:

The purpose of this document is to detail standards for coding shell scripts. Following these standards will help create shell scripts that are easy to read, understand, and maintain.

The intent is that the database admins & developers across organizations would follow these standards to create a consistent and high standard of coding practices.

Keywords: Shell Script, Standards

Introduction to Shell Script Standards

A *shell* is a program that interfaces between the user and the Unix operating System; it takes the commands entered by the user and calls the Unix operating system to run those commands.

There are many types of shells. Some of the well-known ones are

1. Bourne Shell
2. C Shell
3. Korn Shell
4. Bash Shell (Bourne again Shell)

Unix commands placed in a file to be executed by a shell is called *shell script*.

Shell scripting is very powerful and fast. It is used in for various task. Some of the examples are as follows:

Some of the places where Shell Scripts are used are:

1. Starting and Stopping a Server
2. Taking backups of databases and various other administrative tasks.
3. Utility scripts for performing various transformation on files in a data warehousing environment where traditional ETL tools (DataStage/Informatica) might not be efficient.
4. For one time use only requirements, for example, Update or Delete records for fixing data quality issues.

Shebang Line

The first line of a shell script should always identify which shell is being used. This line is called the Shebang line.

Shebang Examples for Various Shells

Korn Shell:

```
#!/bin/ksh
```

Bash Shell:

```
#!/bin/bash
```

Awk Shells:

```
#!/bin/awk -f
```

Perl:

```
#!/usr/bin/perl
```

Header & Revision History

The header of the script should provide following details:

1. Name of the Shell Script/Program
2. Short Description about the script
3. Detailed Description about the script with the details of required input parameters
4. Name of the Developer and Date created.
5. Revision History with any updates made to the script.

In organizations today, there are multiple developers who work on the same script at different times as part of various projects/tasks. It is necessary to track these changes in the revision history.

Sample Header

```
#!/bin/ksh
#|------------------------------------------------------------------
#| Program Name: InsertDQ_db.ksh
#|------------------------------------------------------------------
#| Description: This script inserts data into control tables for Data
#|------------------------------------------------------------------
#| Description: This script inserts data into control tables for Data
#|          1) This script checks if the database is available
#|          2) Executes Inserts statements against DQ_Monitoring Tables
#|          3) Does an Audit count check to verify all records are load
#|          4) Logs entry into process complete table for traceability
#|
#| Note:
#|          a) This script takes config file as a parameter and inserts
#|
#|------------------------------------------------------------------
#| Author: Imran Syed
#| Date: 2017/01/01
#|------------------------------------------------------------------
#|Revision History:
#|Date | Developer Name: 2019/01/01 | Imran Syed
```

```
#|Description of Change Made: Added additional tables for Inserts.
#|-----------------------------------------------------------------
```

Indentation

Indentation helps identify sections of code much easier increasing the readability and understandability of the scripts. If the scripts are not indented properly, it would be very challenging for a person to understand someone else's code and make modifications.

Code blocks should be indented. Whether the indentation is two spaces, three spaces, four spaces or a tab, it should be decided on as a group and then adhered to.

Annotations

The code should be well commented. The annotations should be very clear on why a specific piece of code was written, and what it is trying to achieve. The code should be self-documenting. Each section of code shall have appropriate comments, so that the reviewer gets a good understanding of what the code is trying to achieve than guessing at author’s intent.

Examples of Indentation and Annotation

```
#-------------------- Check for Config ------------------------------
if [[ ! -s ${CFGFILE} ]]; then
            echo “Error: Could not find Config file: ${CFGFILE}”
            echo “Exiting with Status Code of 10”
            exit 10
fi
```

Configuration Files

Configuration Files are used to pass parameters to shell scripts. It is highly discouraged to hard code any of the parameters in shell scripts. Hard coding parameters may result in code change if any of the parameters change.

Config files can be sourced in the main shell script to provide input parameters.

Config files can be located in a directory where all config files are located for that department/organization on that server. Config files are used to pass details like source file directory, target file directory, database name, log file directory, and many others.

Credentials can be stored in vaults. When connecting to Teradata Database, Teradata vault could be invoked in the shell script to get the credentials.

Sample Config File

```
DB_Name= ‘DQ_DB_DEV’
ControlDir =/apps/dev/customer/controlThan
LogDir=/apps/dev/customer/log
```

Validate All Parameters Passed Through Config Files:

Validate all the parameters that are passed to shell script by sourcing configuration file. For example, if the directory path is passed, check if it is a valid directory location. If the source file name is passed, check to see if the file exists.

All parameters sourced from configuration file need to be validated if possible.

Examples

To check if the config file exists:

```
if [[ ! -s ${CFGFILE} ]]; then
            echo “ERROR: Could not find config file: ${CFGFILE}”
            echo “Exiting with Status Code of 10”
            exit 10
fi
```

To check if the log directory exists:

```
if [[ -d ${LOGDIR} ]];then
            echo “Log Directory exists”
else
```

```
            echo “Invalid Log Directory: ${LOG_DIR}”
            echo “Exiting with Status code of 10”
            exit 10
fi
```

Usage Function

It is a good practice to write scripts that are largely self-explanatory. Developers should be able to look at the script and figure out what is being accomplished. The usage function shall be invoked when the script is called with the option of “-h”, i.e. “scriptname -h”. The usage function shall clearly explain which parameters are required and which parameters are optional. Use the getopts function to implement Usage functionality.

USAGE Function Example

```
#------------------------------Help Function ----------------------
function help
{
            USAGE=”USAGE:${PROG} -h[Help] -c Config File “
            Print $USAGE
            cat <<-!EOF
                        -h see this help
                        -c Config File   <fullpath>      (required)
            !EOF
            exit 0
}
```

Parsing Command Line Parameters

Use the getopts function to parse the command line parameters.

```
while getopts hc:p: OPT 2>/dev/null; do
            case $OPT in
                        h)
                        help
                        ;;c)
```

```
c)
                        CFGFILE=$OPTARG
                        echo “CFGFILE : “${CFGFILE}
                        ;;
                        *)
                        help
            ;;
            esac
done
```

Shell script Logs

Every shell script creates a log file in the corresponding project log directory with all the details of execution. The name and location of the log file could be read from configuration file.

Many times, shell scripts are called using ETL tools (DataStage/Informatica). In those cases it would be ideal to log the entries both in the ETL tool logs and also a specific log file generated by a shell script. This could be achieved by using a small custom function.

Custom Function Example

```
#-------------------------------Function to Echo and Log to Log File-
echot() {
            echo “$*” | tee -a ${LOGFILE}
}
```

Instead of calling the echo function, if you call echot function this way, you can send the log entries the ETL tools logs as well create a separate log file for the shell script.

Variable Names

The Korn Shell and POSIX standards (also referred to as the Open Systems Standards) have many more reserved names that the original Bourne shell. All these reserved variable names are spelled in upper case like PWD, SECONDS, etc.

Avoid using user defined variable names in all upper case. User defined variable names preferably should be prefixed with “v_” to distinguish easily reserved variables and user defined variables.

Always enclose variable names with in flower (also known as curly or squiggly) brackets {Reserved_Variable_Name} or {v_UserDefined_Variable_Name} for any variable name longer than one character unless it's a simple variable name.

Variable Examples

Not Recommended:

```
echo $USER_ID
```

Recommended:

```
echo ${v_USER_iD}
```

Names of local, non-environment, non-constant variables should be lowercase. Names of variables local to the current script which are not exported to the environment should be lowercase, while variable names which are exported to the environment should be uppercase.

Do not use variable names which are reserved keywords in C/C++/JAVA or the POSIX standards in order to avoid confusion and/or future changes/updates to the shell language.

Sourcing Other Shell Scripts

Use the keyword “source” instead of ‘.’ (dot) to source other shell scripts in the current shell script. Using the keyword “source” is easy to read when compared with the tiny dot, and a failure can be caught easily within the script.

Source Example

Not Recommended:

```
            .${CFGFILE}
```

Recommended:

```
            source ${CFGFILE}
```

Use Proper Exit Code

Exit codes must be explicitly defined in a shell script. If no exit codes are defined, the shell script will show the return code of the last command executed which can be misleading for the debugging of shell.

A shell script should exit with a return code of 0 when successfully completed. Exit codes between 1 and 4 can be used to exit with warnings. Fatal errors shall have an exit code greater than 4 and shall abort the script.

It would be preferable to have different sections of the code exit with different error codes, so, that it’s easy to pin point the failure cause.

Exit Code Examples

1. When help function is invoked and completed successfully, it exits with return code 0

```
# Help Function
help () {
            print $USAGE
            cat <<-!EOF
                        -h see this help
                        -c Config File   <fullpath>       (required)
!EOF
            exit 0
}
```

When the configuration file is not found, the shell exits with exit code of 10

```
#-----------------------Check for Config----------------------------
if [[ ! -s ${CFGFILE} ]];then
            echot “Error: Could not find Config file: ${CFGFILE}”
            echot “Exiting with Status Code of 10”
            exit 10
fi
```

Avoid Using System Exit Codes

Avoid using system exit codes during coding. System exit codes are reserved exit codes used by Unix system and they should not be used for any other purpose. The system exit codes can be found at /usr/include/sysexits.h

Below is the List of System Exit Codes

```
# define EX_OK                      0          /* successful termina
```

Functions

Functions are used in order to break the code into readable and reusable sections. Do not use function names which are reserved keywords or function names in C/C++/JAVA or the POSIX shell standard in order to avoid confusion and/or future changes/updates to the shell language.

Function Examples

ksh-style function

```
function help {….}
```

Bourne -style functions

```
foo() {….}
```

Use Local Variables Instead of Global Variables

Local Variables are variables that are only available in the script in which they are defined. These variables are not available to any child process that may be started from the main shell script.
Global/Environmental variables are variables that are available in the main script and to any child process that are started from the main script.
There are times were environmental variables are required but unnecessary use of environmental variables in shell scripts should be avoided.
Most commonly environmental variables are used in .profile

Local Variable Example

```
# Defining local Variable
i=5;
# echo the value in variable.
echo $5
```

Global Variable Example

```
# Defining global Variable
export PATH=/dev/in:/test/in
```

Test Command

Use the new test command "[[expr]]" instead of the old test command "[expr]"

With the use of new test command [[…]], additional operators are available. Wild cards can be used in string-matching tests, and many of the errors from the old test command have been eliminated.

Test Command Example

```
if [[ $i = [Yy]* ]]
```

Here the contents of variable “i” are tested to see if the matches anything starting with Y or y.

Let Command

Use the let command “((….))” for numeric expressions. Use the let command “((….))” instead of “[expr]” or “[[expr]]” for numeric expressions

Let Command Example

Avoid using

```
i=5
if [ $i -gt 5 ]; then
```

Use

```
i=5
if (( $i > 5 )); then
```

Compare Exit Codes Using Arithmetic Expressions

Use the POSIX arithmetic expressions to test for exit/return codes of commands and functions.

Exit Code Example

```
if (( $? > 0)); then
```

Instead of

```
If [ $? -gt 0 ]; then
```

True Command

Use built-in commands in conditions for while endless loops. Make sure that your shell uses “true” built-in (ksh93) when executing endless loops.

True Command Example

```
while true ; do
            sleep 10 ;
done
```

Temporary Files

Avoid creation of temporary files and store the values in variables where possible

Temporary File Examples

Not Recommended:

```
ls -1 > filename
for i in $(cat filename); do
   sleep 10;
done
```

Recommended:

```
x=”$(ls -1)”
for i in ${x} ; do
            sleep 10;
done
```

Command Name and Arguments Before Redirections.

Put the command name and arguments before redirections.

Command Name and Argument Example

Use

```
date > file
```

Instead of

```
            > file date
```

Conclusion

By Implementing the above shell scripting standards, developers could create shell scripts that are consistent, easy to read, understand and maintain.