BASH SCRIPTING CRASH COURSE

This document will expose you to a highly functional method of Bash shell scripting while taking you through enough theory to help you see valuable patterns in Bash scripting.

Author: Tay Kratzer tay@cimitra.com
Document Version: 1.0
Published Date: 6/15/19
NOTE: This is a blog entry from cimitra.com

Bash Usage Evolution

BASH 101

1. Issue a command
2. Get a result
3. Analyze the result
4. Act upon the result

BASH 201

1. Issue a command, or a chain of commands
2. Get results
    a. Standard Output
    b. Standard Error
    c. Exit Code
3. Analyze one or more of the results
4. Act upon the results

BASH 301

If you do a sequence of commands more than once, regularly (every couple of weeks) automate it with a **BASH shell scrip**t.

Why?

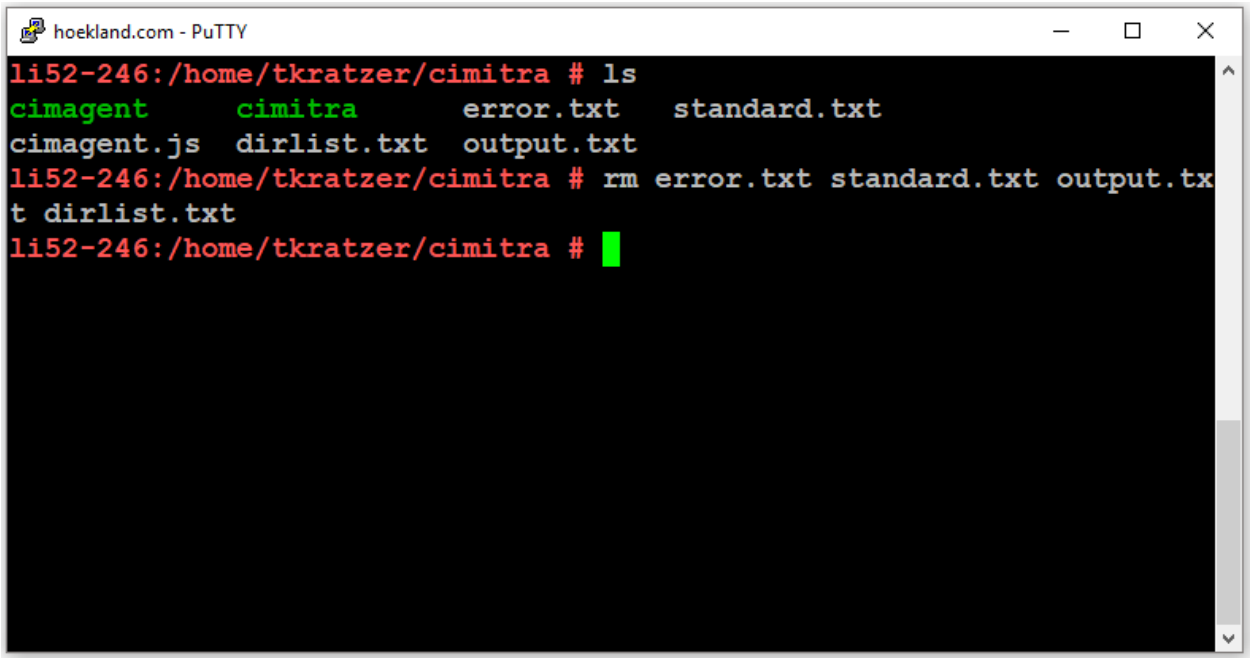
- Eliminates errors
- Saves time
- More fun! Because you are creating which is fundamental to our being

BASH 401

1. Create Functions in BASH shell scripts
2. Create Function Libraries
3. Create Schema Libraries

BASH 101

1. Issue a command: **`ls`**
2. Get a result: **`cimagent cimitra error.txt standard.txt cimagent.js dirlist.txt output.txt`**
3. Analyze the result: ***"I need to remove the .txt files, they aren't needed"***
4. Act upon the result: **`rm error.txt standard.txt output.txt dirlist.txt`**

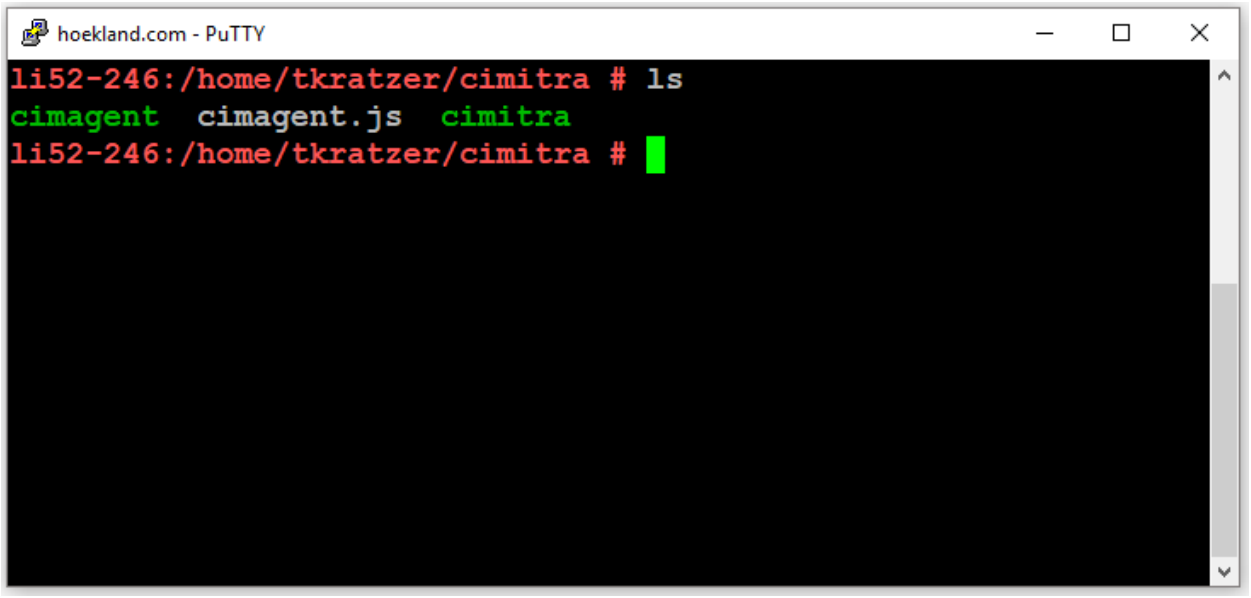


BASH 201

Standard Output

Issue a command, get some output [**Standard Output**], which by default is directed to the **screen** you are in.

Use the directory **lis**ting command: **`ls`**



Issue a command, redirect **Standard Output** (**1>**) to **a file …** instead of the screen

```
ls 1> ./dirlist.txt
```

Now let's see the contents of the file that we redirected **Standard Output** to

```
cat dirlist.txt
```

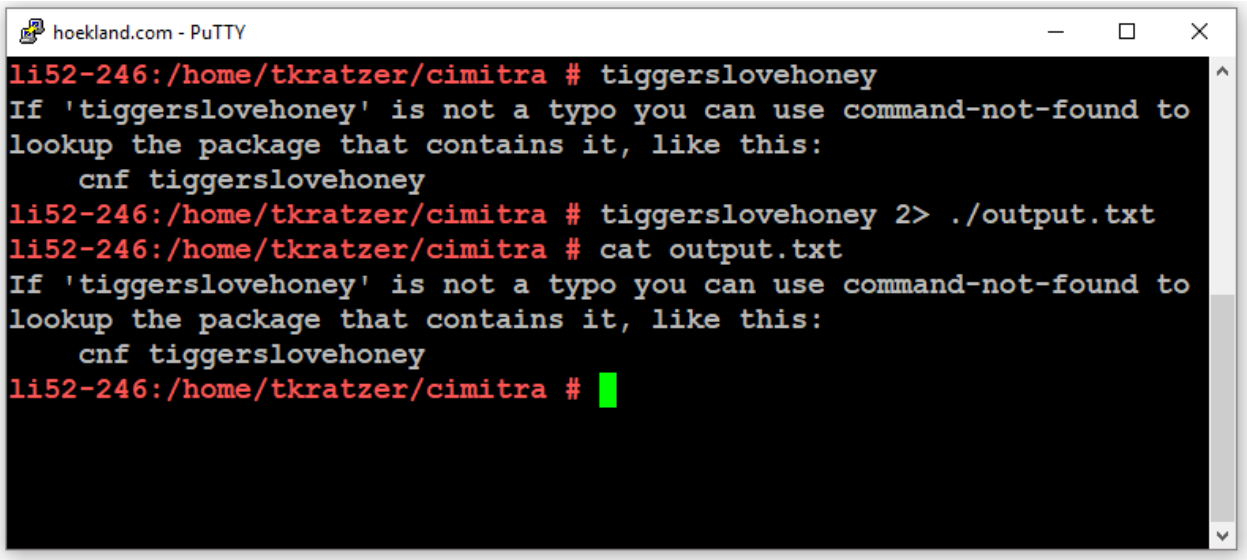
Standard Error

Issue a **bogus** command, get some output [**Standard Error**], which by default is directed to the **screen** you are in.

```
tiggerslovehoney
```

Issue a command, redirect **Standard Error** (**2>**) to **a file …** instead of the screen

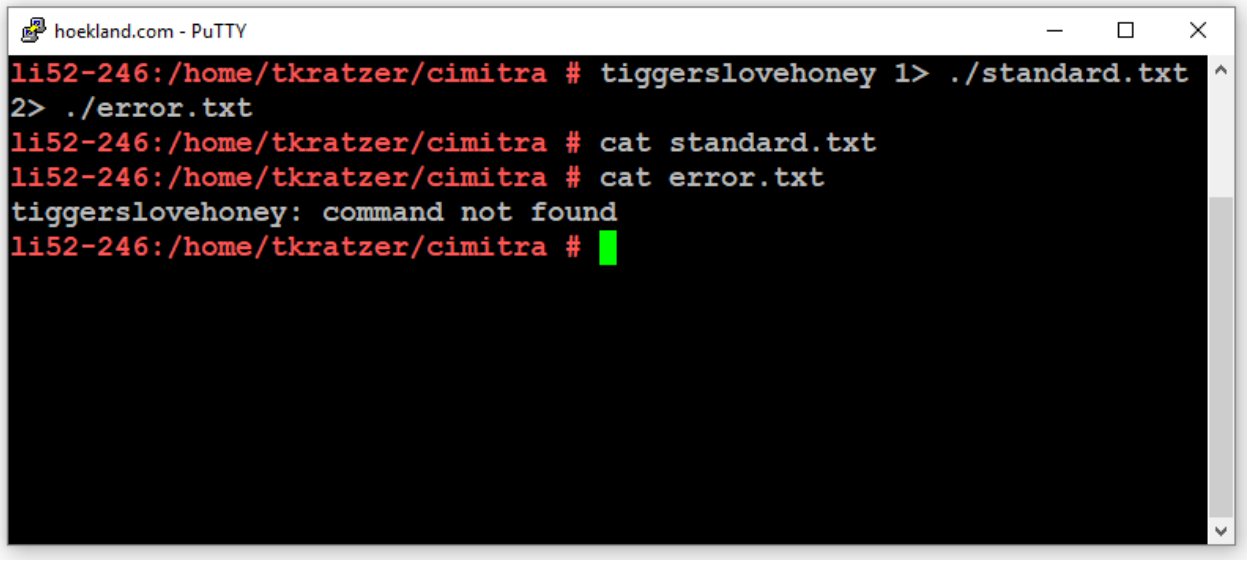
```
tiggerslovehoney 2> ./output.txt
```



Issue a command, redirect the **Standard Output** (**1>**) to a file and the **Standard Error** (**2>**) to a different file `...` instead of the screen

```
tiggerslovehoney 1> ./standard.txt 2> ./error.txt
```

```
cat standard.txt
cat error.txt
```



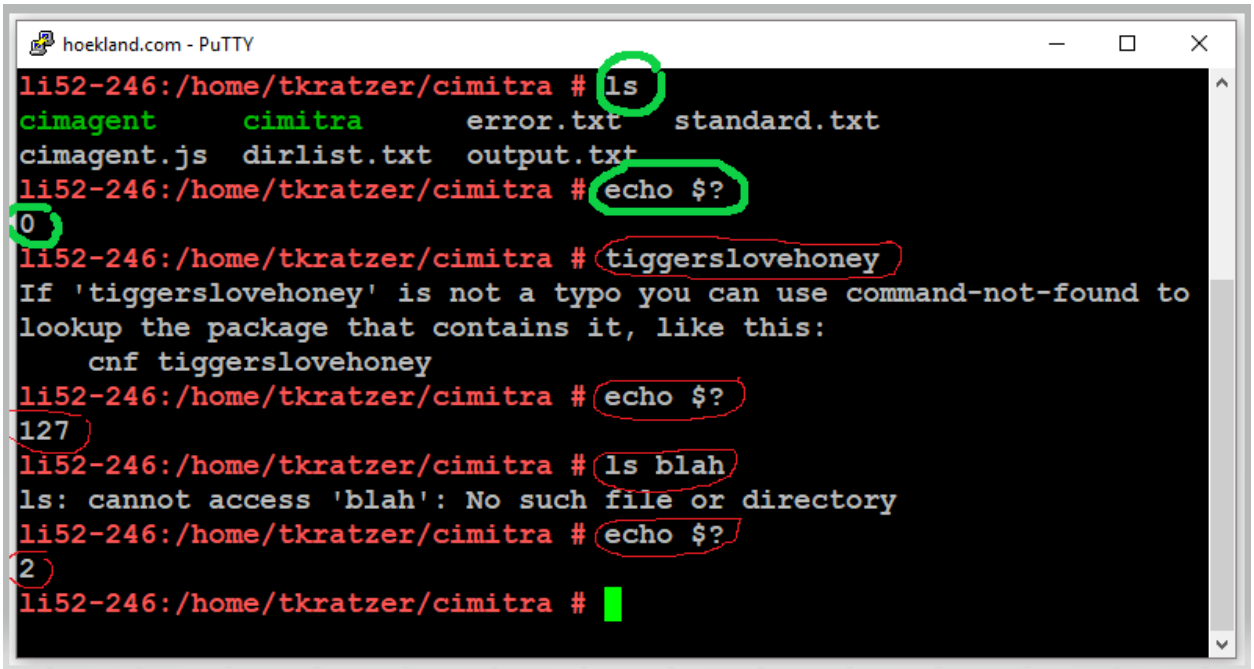
Exit Codes

Every command you call in Linux has an "exit" code

0 = Generally means no error	**1 = Generally means error**	**Some other number generally means error or warning**

You can get the exit code from a command in this manner:

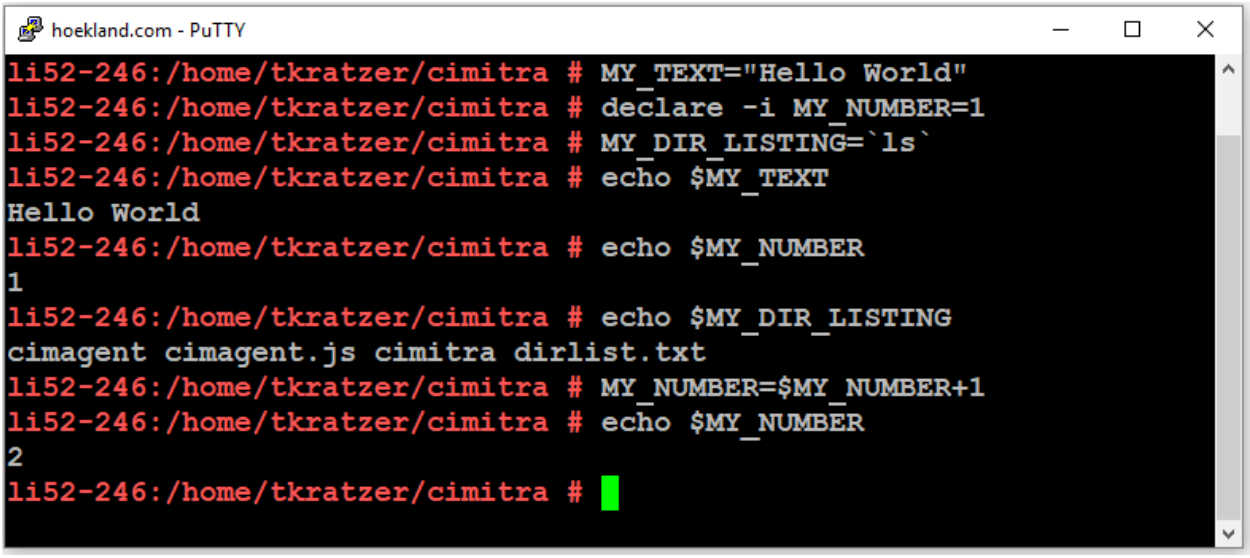
```
<Issue the Command>
echo $?
```



BASH 301

Making Variables

```
MY_TEXT="HELLO WORLD"

declare -i MY_NUMBER="1"

MY_DIR_LISTING=`ls`

echo $MY_TEXT

echo $MY_NUMBER

echo $MY_DIR_LISTING

MY_NUMBER=MY_NUMBER+1

echo $MY_NUMBER
```



Self Documenting Code

YES	NOPE!
`declare -i WEB_SERVER_RUNNING_STATE=1`	`declare -i VAR1`
`function checkWebServerStatus()`	`function CWS()`

Style & Convention

Establish your style . . .	. . . stick with it!
Lower Camel Case	Underscores
webServerRunningState	**WEB_SERVER_RUNNING_STATE**
Upper Camel Case	Underscores with Lowercase
WebServerRunningState	**web_server_running_state**

You don't have to use one convention for everything though. For example, functions might use a certain kind of convention different from variables. And variables that are global might use a different convention than local variables.

Example Conventions

Example Function Convention	Example Variable Conventions
Looks different from variables, explains what the function does **checkWebServerStatus**	Global Variable - Available in the entire script **WEB_SERVER_RUNNING_STATE**
	Local Variable - Available only in a function **web_server_running_state**

Code Simplicity

Short is good, in the right place. Your goal is to have the least amount of lines of code. However, you should never scrimp on the names of variables and functions. The names of variables and functions should always be very descriptive and self-documenting.

Let's make a BASH shell script:

1. The first line should specify the interpreter (BASH): **#!/bin/bash**
2. Now list the BASH commands you want to accomplish in the script
3. Use comments generously, use the pound symbol (**#**) before comments

```
#!/bin/bash
# Check Web Server Version 1
# Determine if the local web server is running
# curl is a command line web browser
# Use curl to test the default HTTP port (80)
# The web server is local to this Linux box

# Create a variable for restarting the web server
RESTART_WEB_SERVER="rcapache2 restart"

# Issue curl command
curl localhost
# Analyze the exit code of the curl command
declare -i web_server_running_state=`echo $?`

# If the exit code is not zero (0) take action
if [ $web_server_running_state -ne 0 ]
then
${RESTART_WEB_SERVER}
echo "The web server was restarted"
else
echo "The web server is running"
fi
```

Save the contents of the script to a Linux box, and make the file executable. On Windows, I use WinSCP as my editor typically.

To make a script executable in a console session (Generally **putty** on Windows) type the following command:

```
chmod +x /home/scripts/webserver.sh
```

To run a BASH script type something similar to this:

```
./webserver.sh
```

or

```
/home/tkratzer/scripts/webserver.sh
```

LEARNING NOTE

The construct used in this script is called an **if then else fi** statement. You can test all kinds of things with this construct. Another common construct is called an **if then fi** statement. For example:

```
if [ $web_server_running_state -ne 0 ]
then
${RESTART_WEB_SERVER}
echo "The web server was restarted"
fi
```

The `if then fi` construct is extremely popular since often you only want to take an action only if a condition exists, and if the condition does not exist you do not want to take any kind of action.

What's with the **fi** portion of these constructs? Well, **"fi"** is the ending of the **if** statement so it's kind of like the opposite of **if** . . . similar to opening and closing tags **<** and **>**. It's also somewhat humorous in that it isn't a proper word, but these weirdisms are common in the Unix/Linux/BASH world which makes for comic relief and endearment towards the platform and the BASH language.

The `-ne` is a testing method for numbers. It means **n**ot **e**qual to. There are others such as **-gt** (greater than) **-lt** (less than) **-eq** (equal to).

The line `[ $web_server_running_state -ne 0 ]` is called a test statement. It means: <TEST BEGIN **[** ><THE TEST STATEMENT(**$web_server_running is not equal to 0**><TEST END **]**>

There are a couple of things that I want to improve on in this script. First off the output from the script is too chaotic. We want to **subdue the output** from the curl command. It's really just the **exit code** we want to get without all of the other output.

Here is a command for running curl and subduing the output:

```
curl localhost 1> /dev/null 2> /dev/null
```

What this command is doing is this:

curl *check* **localhost** *and send the standard output* (**1>**) *and standard error* (**2>**) *into a black hole* (**/dev/null**) *because I don't need the output.*

The next thing I want to do is combine the **curl** command and **echo** command into one statement associated with a variable. We use the semicolon command (**;**) which basically means, after you do this command, then run another command.

```
declare -i web_server_running_state=`curl localhost 2>
/dev/null 1> /dev/null ; echo $?`
```

Below is the second version of the BASH script. It has the following improvements:

1. The script is a very quiet script where all the noise from the curl command is filtered out. This increases the ease of use of the script. Before, we didn't know if the output was coming from the script or the curl command.

2. The curl command, and the variable to hold the state of the web server, are now combined into one element.

```
#!/bin/bash
# Check Web Server Version 2
# Determine if the local web server is running
# curl is a command line web browser
# Use curl to test the default HTTP port (80)
# The web server is local to this Linux box
```

```
# Create a variable for restarting the web server
RESTART_WEB_SERVER="rcapache2 restart"

# Issue curl command, supress output, get exit code
declare -i web_server_running_state=`curl localhost 2>
/dev/null 1> /dev/null ; echo $?`


# If the exit code is not zero (0) take action
if [ $web_server_running_state -ne 0 ]
then
${RESTART_WEB_SERVER}
echo "The web server was restarted"
else
echo "The web server is running"
fi
```

BASH 401

Functions
Function Libraries
Schemas

Functions

The command to restart the web server is currently a global variable called:

```
$RESTART_WEB_SERVER
```

However, it is better and more scalable over time to call a **function** to restart the web server. This way if we wanted to do more things than restarting the web server we could. Here is how a function is created:

```
function nameOfFunction()
{
<Commands to run in function>
}
```

So for example:

```
function restartWebServer()
{
rcapache2 restart
echo "The web server was restarted"
}
```

Tips for writing functions:

1. Functions should do one thing, and do it well

2. Functions must exist above the line in the script that is calling the function

In our web server script, I would like to add another function for **logging**. Here are the requirements of the log function:

1. It will take input
2. It will write to a log file
3. It will keep the log file trimmed to 100 lines
4. It will proceed each logged line with the date and time

We will make a second function that will perform the actual **trimming** of the log. That way this function can be reused. This will also help to keep the log function smaller and more concise to the function's actual purpose.

We will also contain the web server restart and the checking of the web server into their own two functions called:

`checkWebServer()` and **`restartWebServer()`**

The **checkWebServer()** function will be the only function name actually in the body of the script. There are 3 other functions that are called in this script, but they are called from other functions. The **function names** are in **blue**. The **calls to the functions** are in **yellow**.

```
#!/bin/bash
# Check Web Server Version 3
# Determine if the local web server is running
# curl is a command line web browser
# Use curl to test the default HTTP port (80)
# The web server is local to this Linux box

function trimTextFile()
{
# Assign first passed variable to: text_file_to_trim
text_file_to_trim=$1

# Assign second passed variable to: max_file_length
max_file_length=$2

# See if the file exists
declare -i text_file_exists=`test -f $text_file_to_trim
; echo $?`

# The file does not exist, nothing to do!
if [ $text_file_exists -ne 0 ]
then
return
fi

declare -i current_file_length=`wc -l <
$text_file_to_trim`

# The file is not beyond the max, nothing to do!
```

```
if [ $current_file_length -lt $max_file_length ]
then
return
fi

# Make a temporary file
# Note: $RANDOM is a built-in variable in Linux

temp_file="/tmp/${RANDOM}.tmp"

declare -i can_create_temp_file=`touch ${temp_file} ;
echo $?`

# If cannot create temp file, then get out of here
if [ $can_create_temp_file -ne 0 ]
then
return
fi

tail -${max_file_length} ${text_file_to_trim} 1>
$temp_file

mv ${temp_file} ${text_file_to_trim}
}

function log()
{
# Assign first passed variable to: input_text_to_log
input_text_to_log=$1

# Define the location and name for the log file
log_file="/tmp/webserverlog.txt"
```

```
# Integer variable for the max number of log lines
declare -i max_log_length="100"

date_string=`date`

echo "${date_string} : ${input_text_to_log}" 1>>
${log_file}

trimTextFile ${log_file} ${max_log_length}

}


function restartWebServer()
{
rcapache2 restart
echo "The web server was restarted"
}

function checkWebServer()
{
# Issue curl command, supress output, get exit code
declare -i web_server_running_state=`curl localhost 2>
/dev/null 1> /dev/null ; echo $?`

# If the exit code is not zero (0) take action

if [ $web_server_running_state -ne 0 ]
then
restartWebServer
log "The web server was restarted"
else
echo "The web server is running"
```

```
fi
}

checkWebServer
```

Function Libraries

The **`trimTextFile()`** function is a pretty handy function. This function could have a lot more life and usability in other scripts if somehow we could break it away from the **`webserver.sh`** script. We could always copy the function to another script. However, the problem with that is that if we wanted to improve the function, we would then have to copy the updated function to other scripts that use that function. Where's the joy in that!

So here is how we can do this.

1. Create a file called "**`functions.sh`**" that contains the **`trimTextFile()`** function.
    a. The **`functions.sh`** script can contain other reusable functions, and so we will call it a **function library**.

2. Then read the **`functions.sh`** script at the top of the **`webserver.sh`** script. This puts the **`functions.sh`** script contents in memory so that the contents of the **`functions.sh`** script can be called from within anywhere within the **`webserver.sh`** script. Here is how you read in an external file:

```
<period symbol><space><path to the external file>
```

So for example:

```
. /home/tkratzer/scripts/function.sh
```

3. Calls to the **`trimTextFile()`** function are simply calls to the function that have already loaded in memory from the **`functions.sh`** script.

Now we have a tidier script file that is short and more concise. Unlike basketball, in the coding world, shorter is better. Fewer lines of code mean less debugging. See version 4 of our script

below. The line that loads **the call to the function library is highlighted in green (because you are now using reusable the code ;)))))**).

```
#!/bin/bash
# Check Web Server Version 4
# Determine if the local web server is running
# curl is a command line web browser
# Use curl to test the default HTTP port (80)
# The web server is local to this Linux box

. /home/tkratzer/scripts/functions.sh

function log()
{
# Assign first passed variable to: input_text_to_log
input_text_to_log=$1

# Define the location and name for the log file
log_file="/tmp/webserverlog.txt"

# Integer variable for the max number of log lines
declare -i max_log_length="100"

date_string=`date`

echo "${date_string} : ${input_text_to_log}" 1>>
${log_file}

trimTextFile ${log_file} ${max_log_length}
}

function restartWebServer()
{
```

```
rcapache2 restart
echo "The web server was restarted"
}

function checkWebServer()
{
# Issue curl command, suppress output, get exit code
declare -i web_server_running_state=`curl localhost 2>
/dev/null 1> /dev/null ; echo $?`


# If the exit code is not zero (0) take action

if [ $web_server_running_state -ne 0 ]
then
restartWebServer
log "The web server was restarted"
else
echo "The web server is running"
fi
}

checkWebServer
```

Schemas

A schema might sound kind of scary, but it's really a simple idea. Schemas are really just a **static (unchanging) variable library**. So instead of a file with a bunch of functions in it like a function library, a schema is a file with a bunch of pre-established variables. For example:

```
TEMP_PATH="/tmp"
WEB_SERVER_RESTART_COMMAND="rcapache2 restart"
CIMITRA_RESTART_COMMAND="cimitra restart"
SERVER_INFO=`cat /etc/issue`
```

The name you give to a schema file doesn't really matter. It can be called something like **`schema.lib`** for example. Or just **`schema`**. The actual name is your choice. You read in your schema file in the exact same manner as a function library. So…

`<period symbol><space><path to the external file>`

Example:

```
. /home/tkratzer/scripts/schema.lib
```

Your function library and your script would generally load the same schema file. This way they can use common variables. You don't need to put the #!/bin/bash at the top of the schema library, but you can if you would like.

Variable Configuration Files

You can store variables in a file. I like to name the file with a .cfg extension. Here is an example of the contents of a variable configuration file.

```
MAX_LOG_LENGTH="100"
MAX_AGENT_RETRIES="10"
ADMIN_EMAIL_ADDRESS="tay@cimitra.com"
CIMITRA_APP_SERVER="155.100.111.141"
```

You read in your variable configuration files in the exact same manner as a function library. So…

`<period symbol><space><path to the external file>`

Example:

```
. /home/tkratzer/scripts/variables.cfg
```

Once the variables are read in they are available by using the following syntax:

${<VARIABLE NAME>}

For example:

`${MAX_LOG_LENGTH}`

The primary purpose behind variable configuration files is so that you don't have to store variables that you would want to change in the script that you have created. The script will stay static, and the configuration file will change as needed. **DO NOT use the #!/bin/bash** line in a variable configuration file, because this is not code, it is a configuration file.

Conclusion

Bash is a never-ending journey. You can create very elaborate scripts that total in the thousands of lines. However, your goal with Bash scripting should always be:

- Self-documenting code through descriptive variables and functions
- Abstraction by using function libraries, schemas and configuration files
- Simplicity - Brings less confusion and more joy
- Finding shorter and shorter forms of getting results from Bash so that your code minimizes as you gain more familiarity with the language