

# Creating Fun Little Shell Scripts

## IN THIS CHAPTER

Sending a message

Getting inspiration

Sending a text

**T**he primary reason for learning to write bash shell scripts is to be able to create your own Linux system utilities. Understanding how to write useful and practical script utilities is important. However, sometimes it helps to do something fun to learn a concept or skill. The scripts in this chapter are not necessarily practical, but they can be lots of fun! And they help solidify script-writing concepts.

## Sending a Message

---

Messages can be sent in many ways in an office or a home environment — text message, e-mail, and even making a phone call. One method, not commonly used any more, is sending a message directly to a fellow system user's terminal. Because this technique is largely unknown, it can be fun to communicate with someone with this method.

This shell script utility helps you to quickly and easily send a message to someone who is logged onto your Linux system. It is a rather simple script, but it can be loads of fun!

## Understanding the required functions

For this simple script, only a few functions are required. Several of the commands are common and have been covered in the book. However, a few of the commands have only been touched on, and you may not be familiar with the primary command needed. This section looks at the commands needed to put together this simple, but interesting script.

### Determining who is on the system

The first utility needed is the `who` command. The `who` utility allows you to see all the users currently logged into the system:

```
$ who
christine tty2          2015-09-10 11:43
timothy   tty3          2015-09-10 11:46
[...]
$
```

In this partial listing, all the information needed for sending messages is shown. By default, the `who` command gives you the short version of information available. The same information is provided, when `who -s` is issued:

- User name
- User's terminal
- Time the user logged into the system

To send a message, you only need the first two items. Both the user name and the user's current terminal are necessary.

### Allowing messages

Users can disallow anyone to send them messages via the `mesg` utility. Therefore, before you start attempting to send messages, it's a good idea to check whether messages are allowed. For yourself, you can simply enter the `mesg` command as follows:

```
$ mesg
is n
$
```

The `is n` result shows that messaging is turned off. If the result showed `is y`, messages would be allowed.

#### TIP

Some distributions, such as Ubuntu, come with messaging turned off by default. Other distributions, such as CentOS, come with messaging turned on by default. Thus, you need to check your status and other user's message status before attempting to send a message.

To check everyone else's message status, you can use the `who` command again. Keep in mind that this checks the message status only for those who are currently logged into the system. You use the `-T` option to check their message status:

```
$ who -T
christine - tty2          2015-09-10 12:56
```

```
timothy - tty3          2015-09-10 11:46
[...]
$
```

The dash (-) after each user name indicates that messaging is turned off for those users. If it is turned on, you see a plus (+) sign.

To allow messages to be sent to you, if it is turned off, you need to use the message command with the `y` option:

```
$ whoami
christine
$
$ mesg y
$
$ mesg
is y
$
```

Messaging is turned on by the user `christine`, when the command `mesg y` is issued. The user's individual message status is checked, by issuing the `mesg` command. Sure enough, the command shows `is y`, which indicates messages are allowed to this user.

Using the `who` command, other users can see how the user `christine` has changed her message status. The message status is now set to a plus sign, which indicates the user is allowing messages to be sent to her.

```
$ who -T
christine + tty2          2015-09-10 12:56
timothy - tty3           2015-09-10 11:46
[...]
$
```

For two-way communication, you need to allow messaging and one or more users also need to allow messaging. In this example, the user `timothy` has also turned on his messaging:

```
$ who -T
christine + tty2          2015-09-10 12:56
timothy + tty3           2015-09-10 11:46
[...]
$
```

Now that messaging is allowed between you and at least one other user, you can try out the command to send messages. However, the `who` command is also still needed, because it provides the necessary information in order to send a message.

### Sending a message to another user

The primary tool for this script is the `write` command. As long as messaging is allowed, the `write` command allows you to send a message to another logged-in user using his username and current terminal.

### NOTE

The `write` command only allows you to successfully send messages to users logged onto a virtual console terminal (see Chapter 2). A user logged into the graphical environment will not be able to receive messages.

In this example, a message is sent from user `christine` to user `timothy` logged on the `tty3` terminal. From `christine`'s terminal, the session looks as follows:

```
$ who
christine tty2          2015-09-10 13:54
timothy  tty3          2015-09-10 11:46
[...]
$
$ write timothy tty3
Hello Tim!
$
```

After the message is initiated by the `write` command, a blank line is shown for you to begin inputting the message text. It may be as many lines as you desire. When the Enter key is pressed, a new line is available for more message text. After you are finished entering message text, the whole message is sent by pressing the Ctrl+D key combination.

The receiver of the message sees something like the following:

```
Message from christine@server01 on tty2 at 14:11 ...
Hello Tim!
EOF
```

The receiver can see which user on which terminal sent the message. A time stamp is also included. Notice the `EOF` shown at the message's bottom. It indicates End Of File, which lets the message recipient know that the entire message is being displayed.

### TIP

Often, a message recipient needs to press the Enter key in order to get a prompt to show again, after a message is received.

Now you can send messages! The next step is to use these commands to create the script.

## Creating the script

Using a script to send messages helps overcome a few potential problems. First, if you have lots of users on the system, trying to find the one user you want to send a message to can be a pain! You must also determine whether that particular user has messaging turned on.

In addition, a script speeds things up allowing you to quickly send a message to a particular user in one easy step.

### Checking if user is logged on

The first issue is to let the script know to which user you want to send a message. This is easily done by sending a parameter (Chapter 14) along with the script's execution. For the script to determine whether that particular user is logged on the system, the `who` command is employed as shown in this bit of script code:

```
# Determine if user is logged on:
#
logged_on=$(who | grep -i -m 1 $1 | gawk '{print $1}')
#
```

In the preceding code, the results of the `who` command are piped into the `grep` command (Chapter 4). The `grep` command uses the `-i` option to ignore case, which allows the username to be entered using uppercase or lowercase letters. The `-m 1` option is included on the `grep` command, in case the user is logged into the system multiple times. The `grep` command produces either nothing, if the user is not logged on, or the username's first login information. This output is passed to the `gawk` command (Chapter 19). The `gawk` command returns only the first item, either nothing or the username. This final output from the `gawk` command is stored in the variable `logged_on`.

#### Tip

Some Linux distributions, such as Ubuntu, may not have the `gawk` command installed by default. To install it, type `sudo apt-get install gawk`. Also, you can find more information about installing software packages in Chapter 9.

When the variable, `logged_on`, contains either nothing (if the user is not logged on) or the username, it can be tested and acted upon:

```
#
if [ -z $logged_on ]
then
    echo "$1 is not logged on."
    echo "Exiting script..."
    exit
fi
#
```

Employing the use of an `if` statement and a test command (Chapter 12), the `logged_on` variable is tested to determine if it is a zero-length variable. If it is a zero-length variable, the script user is informed via `echo` commands that the user is not currently logged onto

the system, and the script is exited via the `exit` command. If the user is logged onto the system, the `logged_on` variable contains the user's username, and the script continues.

In the following example, a username, *Charlie*, is passed as a parameter to the shell script. This user is not currently logged onto the system:

```
$ ./mu.sh Charlie
Charlie is not logged on.
Exiting script...
$
```

The code worked perfectly! Now instead of you digging through the `who` command results to determine whether a user is logged onto the system, the message script does that for you.

### Checking if user accepts messages

The next important item is to determine whether a logged on user accepts messages. This script portion operates very closely to the script section for determining whether a user is logged on:

```
# Determine if user allows messaging:
#
allowed=$(who -T | grep -i -m 1 $1 | gawk '{print $2}')
#
if [ $allowed != "+" ]
then
    echo "$1 does not allowing messaging."
    echo "Exiting script..."
    exit
fi
#
```

Notice that this time, the `who -T` command and option are used. This displays a `+` next to the username, if messaging is allowed. Otherwise, it displays a `-` next to the username, if messaging is not allowed. The results from the `who` command are then piped into `grep` and `gawk` to pull out only the messaging indicator. The messaging indicator is stored in the `allowed` variable. Finally, an `if` statement is employed to test for a messaging indicator not set to `+`. If the indicator is not set to `+`, the script user is informed and the script is exited. However, if the messaging indicator shows messaging is allowed, the script continues.

To test out this script's section, a user who is logged into the system with messaging disabled is tested. The user *Samantha* currently has messaging disabled:

```
$ ./mu.sh Samantha
Samantha does not allowing messaging.
Exiting script...
$
```

The test worked as expected. This script portion eliminates any need to manually check for messaging being enabled or disabled.

### Checking if message was included

The message to be sent is also included as a script parameter. Therefore, another needed check is whether a message was included as a parameter to the `mu.sh` shell script. To test for the message parameter, an `if` statement, similar to those used earlier, must be included in the script's code:

```
# Determine if a message was included:
#
if [ -z $2 ]
then
    echo "No message parameter included."
    echo "Exiting script..."
    exit
fi
#
```

To test out this script portion, a message was not included for a user who is both logged into the system and allows messaging:

```
$ ./mu.sh Timothy
No message parameter included.
Exiting script...
$
```

This is exactly what is needed! Now that the script has performed these preliminary checks, the primary task of sending a message can be undertaken.

### Transmitting a simple message

Before a message is sent, the user's current terminal must be identified and stored in a variable. The `who`, `grep`, and `gawk` commands are employed again:

```
# Send message to user:
#
uterminal=$(who | grep -i -m 1 $1 | gawk '{print $2}')
#
```

To transmit the message, both the `echo` and the `write` commands are used:

```
#
echo $2 | write $logged_on $uterminal
#
```

Because `write` is an interactive utility, it must have the message piped into it for the script to work properly. The `echo` command is used to send the message, `$2`, to `STDOUT`,

which in turn is piped into the `write` command. The `logged_on` variable holds the user-name, and the `uterminal` variable holds the user's current terminal.

Now, you can test sending a simple message to a designated user via the script:

```
$ ./mu.sh Timothy test
$
```

The user Timothy receives the following message on his terminal:

```
Message from christine@server01 on tty2 at 10:23 ...
test
EOF
```

Success! You can now send simple one word messages to other users on your system via your script.

### Transmitting a long message

Often, you want to send more than just a single word to another system user. Let's try a longer message using the current script:

```
$ ./mu.sh Timothy Boss is coming. Look busy.
$
```

The user Timothy receives the following message on his terminal:

```
Message from christine@server01 on tty2 at 10:24 ...
Boss
EOF
```

It didn't work. Only the first word of the message, `Boss`, was sent. This is due to the script using parameters (Chapter 14). Recall that the bash shell considers a space to differentiate between parameters. Thus, because there are spaces in the message, each word is treated as a different parameter. The script must be modified to fix this problem.

The `shift` command (Chapter 14) and a `while` loop (Chapter 13) help with this long message issue:

```
# Determine if there is more to the message:
#
shift
#
while [ -n "$1" ]
do
    whole_message=$whole_message ' $1
    shift
done
#
```



Recall that the `shift` command allows you to process the various provided script parameters without knowing the total number of parameters. The `shift` command simply moves the next parameter in line down to parameter `$1`. First, a primary `shift` must be issued before the `while` loop, because the message starts in parameter `$2`, instead of parameter `$1`.

After the `while` loop is initiated, it continues grabbing each message word, tacking the word onto the `whole_message` variable. The loop then shifts to the next parameter. After the final parameter is processed, the `while` loop exits and the `whole_message` variable contains the entire message to send.

One additional script modification is needed to fix this problem. Instead of just sending parameter `$2` to the `write` utility, the script is modified to send the variable, `whole_message`:

```
# Send message to user:
#
uterminal=$(who | grep -i -m 1 $1 | gawk '{print $2}')
#
echo $whole_message | write $logged_on $uterminal
#
```

Now, again try to send that warning message about the boss coming his way to Timothy:

```
$ ./mu.sh Timothy Boss is coming
Usage: grep [OPTION]... PATTERN [FILE]...
Try 'grep --help' for more information.
$
```

Oops! That didn't work either. This is because when `shift` was used in the script, the `$1` parameter contents were removed. Thus, when the script attempts to use `$1` in the `grep` command, it generates an error. To fix this problem a variable, `muser`, is used to capture the `$1` parameter's value:

```
# Save the username parameter
#
muser=$1
#
```

Now `muser` stores the username. The `$1` parameter in the script's various `grep` and `echo` commands can be replaced by the `muser` variable:

```
# Determine if user is logged on:
#
logged_on=$(who | grep -i -m 1 $muser | gawk '{print $1}')
[...]
echo "$muser is not logged on."
[...]
# Determine if user allows messaging:
#
```

```
allowed=$(who -T | grep -i -m 1 $muser | gawk '{print $2}')
[...]
    echo "$muser does not allowing messaging."
[...]
# Send message to user:
#
uterminal=$(who | grep -i -m 1 $muser | gawk '{print $2}')
[...]
```

To test out the script changes, a multi-word message is sent again. In addition, some emphasis is added to the message by tacking on exclamation points:

```
$ ./mu.sh Timothy The boss is coming! Look busy!
$
```

The user Timothy receives the following message on his terminal:

```
Message from christine@server01 on tty2 at 10:30 ...
The boss is coming! Look busy!
EOF
```

It worked! You can now employ the script to quickly send messages to other users on the system. Here's the final message script with all the needed checks and changes:

```
#!/bin/bash
#
#mu.sh - Send a Message to a particular user
#####
#
# Save the username parameter
#
muser=$1
#
# Determine if user is logged on:
#
logged_on=$(who | grep -i -m 1 $muser | gawk '{print $1}')
#
if [ -z $logged_on ]
then
    echo "$muser is not logged on."
    echo "Exiting script..."
    exit
fi
#
# Determine if user allows messaging:
#
allowed=$(who -T | grep -i -m 1 $muser | gawk '{print $2}')
#
```

```

if [ $allowed != "+" ]
then
    echo "$muser does not allowing messaging."
    echo "Exiting script..."
    exit
fi
#
# Determine if a message was included:
#
if [ -z $2 ]
then
    echo "No message parameter included."
    echo "Exiting script..."
    exit
fi
#
# Determine if there is more to the message:
#
shift
#
while [ -n "$1" ]
do
    whole_message=$whole_message' '$1
    shift
done
#
# Send message to user:
#
uterminal=$(who | grep -i -m 1 $muser | gawk '{print $2}')
#
echo $whole_message | write $logged_on $uterminal
#
exit

```

Because you have made it to the last chapter in this book, you should be ready for a script-writing challenge. Here are some suggested improvements for the message script that you can attempt on your own:

- Instead of passing the username and message as parameters, use options (see Chapter 14).
- If a user is logged into multiple terminals, allow a message to be sent to those multiple terminals. (Hint: Use multiple `write` commands.)
- If the message to be sent is for a user who is currently only logged into the GUI, produce a message for the script user and exit the script. (Remember the `write` command can only write to virtual console terminals.)
- Allow a long message stored in a file to be sent to a terminal. (Hint: Use the `cat` command output piped into the `write` utility, instead of the `echo` command.)

Not only does reading through the script help solidify the script-writing concepts you are learning, but so does modifying the script. Come up with your own creative modification. Have a little fun! It helps you learn.

## Obtaining a Quote

---

Inspirational quotes have long been used in the business environment. You may have a few on your office wall right now. This fun little and interesting script helps you obtain a daily inspirational quote to use as you please.

This section takes you through how to create this script. Included is a new rich utility that has not been covered in the book yet. The script also uses some utilities that have been covered, such as `sed` and `gawk`.

## Understanding the required functions

Several great websites allow you to obtain daily inspiration quotes. Just open your favorite search engine, and you can find many sites. After you find a site for your daily quote, you need a utility to download that quote. For this script, the `wget` utility is just what's needed.

### Learning about the `wget` utility

The `wget` utility is a flexible tool that allows web pages to be downloaded to your local Linux system. From these pages, you can glean your daily inspirational quote.

### NOTE

The `wget` command is an extremely rich utility. In this chapter, only a small portion of its power is used. Find out more about `wget` via the man pages.

To download a web page via `wget`, you just need the `wget` command and the website's address:

```
$ wget www.quotationspage.com/qotd.html
--2015-09-23 09:14:28-- http://www.quotationspage.com/qotd.html
Resolving www.quotationspage.com... 67.228.101.64
Connecting to www.quotationspage.com|67.228.101.64|:80. connected
HTTP request sent, awaiting response... 200 OK
Length: unspecified [text/html]
```

```

Saving to: "gotd.html"

[ <=> ] 13,806 --.-K/s in 0.1s

2015-09-23 09:14:28 (118 KB/s) - "gotd.html" saved [13806]

$

```

The website's information is stored in a file named after the web page. In this case, it's `gotd.html`. And as you might have guessed by now, the file is full of HTML code:

```

$ cat gotd.html

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">

<html xmlns:fb="http://ogp.me/ns/fb#">
<head>
    <title>Quotes of the Day - The Quotations Page</title>
[...]
```

Only a partial HTML code listing is shown here. For the script, the `sed` and `gawk` utilities help strip out the desired inspirational quote. But before tackling the script, you need a little more control over the `wget` utility's input and output.

You can use a variable to hold the web address (URL). Simply pass the variable to `wget` as a parameter. Just don't forget to use the `$` along with the variable name:

```

$ url=www.quotationspage.com/gotd.html
$
$ wget $url
--2015-09-23 09:24:21-- http://www.quotationspage.com/gotd.html
Resolving www.quotationspage.com... 67.228.101.64
Connecting to www.quotationspage.com|67.228.101.64|:80 connected.
HTTP request sent, awaiting response... 200 OK
Length: unspecified [text/html]
Saving to: "gotd.html.3"

[ <=> ] 13,806 --.-K/s in 0.1s

2015-09-23 09:24:21 (98.6 KB/s) - "gotd.html.3" saved [13806]

$

```

The daily inspiration quote script is eventually to be run daily via `cron` (Chapter 16) or some other script automation utility. Thus, having the `wget` command's session output display to `STDOUT` is undesirable. To store the session output to a log file, use the option `-o`. This allows session output to be viewed at a later time:

```

$ url=www.quotationspage.com/gotd.html
$

```

```
$ wget -o quote.log $url
$
$ cat quote.log
--2015-09-23 09:41:46-- http://www.quotationspage.com/qotd.html
Resolving www.quotationspage.com... 67.228.101.64
Connecting to www.quotationspage.com|67.228.101.64|:80 connected.
HTTP request sent, awaiting response... 200 OK
Length: unspecified [text/html]
Saving to: "qotd.html.1"

      OK ..... 81.7K=0.2s

2015-09-23 09:41:46 (81.7 KB/s) - "qotd.html.1" saved [13806]

$
```

The `wget` utility now stores its session output into the log file as it retrieves web page information. If desired, you can view the logged session output by using the `cat` command, as shown in the preceding code.

### NOTE

For various reasons, you may decide that you do not want `wget` to produce a log file or display session output. In this case, just use the `-q` option, and the `wget` command quietly performs its directed duties.

To control where the web page information is stored, use the `-O` option on the `wget` command. Thus, instead of having the web address as the storage file name, you can use the filename of your choice:

```
$ url=www.quotationspage.com/qotd.html
$
$ wget -o quote.log -O Daily_Quote.html $url
$
$ cat Daily_Quote.html

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">

<html xmlns:fb="http://ogp.me/ns/fb#">
<head>
[...]
```

Using the `-O` option allows the web page data to be stored in the designated file, `Daily_Quote.html`. Now that the `wget` utility's output is controlled, the next required function, checking the web address's validity, can be explored.

### Testing a web address

Web addresses change. Sometimes, it seems this happens daily. Therefore, it is important to test the address validity within the script. The `wget` utility gives the ability to conduct such a test with the `--spider` option:

```
$ url=www.quotationspage.com/qotd.html
$
$ wget --spider $url
Spider mode enabled. Check if remote file exists.
--2015-09-23 12:45:41-- http://www.quotationspage.com/qotd.html
Resolving www.quotationspage.com... 67.228.101.64
Connecting to www.quotationspage.com|67.228.101.64|:80 connected.
HTTP request sent, awaiting response... 200 OK
Length: unspecified [text/html]
Remote file exists and could contain further links,
but recursion is disabled -- not retrieving.

$
```

This output indicates that the URL is valid, but it's too much to read through. You can cut down on the output by adding the `-nv` option, which stands for non-verbose:

```
$ wget -nv --spider $url
2015-09-23 12:49:13
URL: http://www.quotationspage.com/qotd.html 200 OK
$
```

The `-nv` option allows just the web address's status to be displayed, making the output much easier to read. Contrary to what you may think, the `OK` at the non-verbose line's end does not indicate that the web address is valid. The indication is that the web address came back as it was sent. This concept is a little unclear, until you see an invalid web address.

To see an invalid web address indicator, the `URL` variable is changed to an incorrect web address. The `wget` command is reissued using this bad address:

```
$ url=www.quotationspage.com/BAD_URL.html
$
$ wget -nv --spider $url
2015-09-23 12:54:33
URL: http://www.quotationspage.com/error404.html 200 OK
$
```

Notice that the output still has an `OK` at its end. However, the web address ends in `error404.html`. This indicates the web address is invalid.

With the necessary `wget` command to grab the inspirational quote's web page information, and the ability to test the web page's address, it is time to start building the script. Your daily inspirational quote awaits retrieval.

### Creating the script

To test the script as it is built, a parameter containing the website's URL is passed to the script. Within the script, the variable `quote_url` contains the passed parameter's value:

```
#
quote_url=$1
#
```

### Checking the passed URL

It is always a good idea to have checks in place within your script. The first check is to ensure that the daily inspirational quote script website's URL is still valid.

As you would expect, the script checks the web address validity with `wget` and the `--spider` option. However, the resulting indicator must be saved so the indicator can be checked later with an `if` statement test. Thus, the resulting indicator must be saved to a variable. This is a little tricky with the `wget` command.

To save the indicator output, the standard `$()` syntax is used around the command. But in addition, `STDERR` and `STDOUT` redirection is needed. This is accomplished by tacking on `2>&1` to the end of the `wget` command:

```
#
check_url=$(wget -nv --spider $quote_url 2>&1)
#
```

Now the indicator status message is saved within the `check_url` variable. To carve out the error indicator, `error404`, from the `check_url` string, parameter expansion and the `echo` command can be used:

```
#
bad_url=$(echo ${check_url/*error404*/error404})
#
```

In this example, *string parameter expansion* allows the string stored in `check_url` to be searched. Think of string parameter expansion as a quick and easy `sed` alternative. Using wildcards around the search word, `*error404*` allows the entire string to be searched. If the search is successful, the `echo` command sends the string `error404` to be stored into the `bad_url` variable. If the search is not successful, the `bad_url` variable contains the `check_url` variable's contents.



Now an `if` statement (Chapter 12) is employed to check the `bad_url` variable's string. If the string `error404` is found, a message is displayed and the script exits:

```
#
if [ "$bad_url" = "error404" ]
then
    echo "Bad web address"
    echo "$quote_url invalid"
    echo "Exiting script..."
    exit
fi
#
```

An easier and shorter method can be used. This method removes the need for string parameter expansion and the `bad_url` variable altogether. A double bracket `if` statement allows a search to be conducted of the `check_url` variable:

```
if [[ $check_url == *error404* ]]
then
    echo "Bad web address"
    echo "$quote_url invalid"
    echo "Exiting script..."
    exit
fi
```

The test statement within the `if` structure searches the `check_url` variable's string. If the string `error404` is found anywhere within the variable string, a message is displayed and the script exits. If the indicator string does not contain the error message, the script continues. This statement saves time and effort. No need for any string parameter expansion or even the `bad_url` variable.

Now that the check is in place, the script can be tested with an invalid web address. The `url` variable is set to an incorrect URL and passed to the `get_quote.sh` script:

```
$ url=www.quotationspage.com/BAD_URL.html
$
$ ./get_quote.sh $url
Bad web address
www.quotationspage.com/BAD_URL.html invalid
Exiting script...
$
```

That works great. Just to make sure that all is well, now a valid web address is tested:

```
$ url=www.quotationspage.com/qotd.html
$
```

```
$ ./get_quote.sh $url
$
```

No error message received. The script works perfectly so far! This is the only check needed, so the next item to be added to the script is obtaining the web page's data.

### Obtaining web page information

Grabbing the inspiration daily quote's web page data is simple. The `wget` command shown earlier in the chapter is used in the script. The only needed change is to store the log file and the HTML file, which contains the web page information, in the `/tmp` directory:

```
#
wget -o /tmp/quote.log -O /tmp/quote.html $quote_url
#
```

Before moving on to the rest of the script, this code section should be tested using a valid web address:

```
$ url=www.quotationspage.com/qotd.html
$
$ ./get_quote.sh $url
$
$ ls /tmp/quote.*
/tmp/quote.log  /tmp/quote.html
$
$ cat /tmp/quote.html

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">

<html xmlns:fb="http://ogp.me/ns/fb#">
<head>
[... ]
</body>
</html>
$
```

The script still works well! The log file, `/tmp/quote.log`, and the html file, `/tmp/quote.html`, were properly created.

#### TIP

If you do not want cookies to be involved when obtaining website information, you can add the `--no-cookies` option to the `wget` command. By default, storing cookies is turned off.

The next task is to dig the daily inspirational quote out of the HTML code within the downloaded web page HTML file. This task requires both the `sed` and the `gawk` utilities.

**Parsing out the desired information**

In order to pull out the actual inspirational quote, some processing must take place. This part of the script uses `sed` and `gawk` to parse out the desired information.

**NOTE**

This section is where the most variety is introduced when you modify this script for your own use. The `sed` and `gawk` utilities are used to search for keywords that are specific for this particular quote website's data. You may need to use different keywords as well as different `sed` and `gawk` commands to extract the data you desire.

The script first needs to remove all the HTML tags from the downloaded web page's information stored in the `/tmp/quote.html` file. The `sed` utility can provide this capability:

```
#
sed 's/<[^>]*//g' /tmp/quote.html
#
```

The preceding code should look very familiar. It was covered in Chapter 21 in the “Removing HTML tags” section.

After the HTML tags are removed, the output looks like the following:

```
$ url=www.quotationspage.com/qotd.html
$
$ ./get_quote.sh $url
[...]
    >Quotes of the Day - The Quotations Page>
>
[...]
>>Selected from Michael Moncur's Collection of Quotations
  - September 23, 2015>>
>>>Horse sense is the thing a horse has which keeps
[...]
>
$
```

This snippet listing shows that there is still too much unnecessary data in this file. Therefore, some additional parsing must be done. Fortunately, the quote text needed is situated right next to the current date. Therefore, the script can use the current date as a search term!

The `grep` command, the `$()` format, and the `date` command can help here. The output from the `sed` command is piped into the `grep` command. The `grep` command uses the



Now we're getting somewhere! However, we can still remove a little more clutter from the quotation.

You may have noticed that two quotations are listed in the output instead of one. This happens occasionally with this particular website. Some days, it may be one quote, and other days, it may be two. Therefore, the script needs a way to pull out only the first quote.

The `sed` utility can help again with this problem. Using the `sed` utility's `next` and `delete` commands (Chapter 21), the string `&nbsp;` is located. After it's found, `sed` moves to the next line of the data and deletes it:

```
#
sed 's/<[^>]*//g' /tmp/quote.html |
grep "$(date +%B '%-d,' '%Y)' -A2 |
sed 's/>//g' |
sed '/&nbsp;/{n ; d}'
#
```

Now the script can be tested to see if the new `sed` addition fixes the multiple quotation problem:

```
$ ./get_quote.sh $url
Selected from Michael Moncur's Collection of Quotations
- September 23, 2015
Horse sense is the thing a horse has which keeps it from
betting on people. W. C. Fields (1880 - 1946) &nbsp;
$
```

The extra quotation is removed! One item remains for the quotation cleanup. At the quotation's end, the string `&nbsp;` is still hanging around. The script could use another `sed` command to remove this pesky item, but just for variety, the `gawk` command is used:

```
#
sed 's/<[^>]*//g' /tmp/quote.html |
grep "$(date +%B '%-d,' '%Y)' -A2 |
sed 's/>//g' |
sed '/&nbsp;/{n ; d}' |
gawk 'BEGIN{FS="&nbsp;"} {print $1}'
#
```

In the preceding code, the input field separator variable, `FS`, is used with the `gawk` command (Chapter 22). The string `&nbsp;` is set as a field separator, which causes `gawk` to drop it from the output:

```
$ ./get_quote.sh $url
Selected from Michael Moncur's Collection of Quotations
- September 23, 2015
Horse sense is the thing a horse has which keeps it from
betting on people. W. C. Fields (1880 - 1946)
$
```

One last needed script action is to save this quotation text to a file. Here the `tee` command (Chapter 15) helps. Now the entire quote extraction process looks as follows:

```
#
sed 's/<[^>]*//g' /tmp/quote.html |
grep "$(date +%B ' %-d, ' %Y)" -A2 |
sed 's/>//g' |
sed '/&nbsp;/{\n ; d}' |
gawk 'BEGIN{FS="&nbsp;"} {print $1}' |
tee /tmp/daily_quote.txt > /dev/null
#
```

The extracted quote is saved to `/tmp/daily_quote.txt`, and any output produced by the `gawk` command is redirected to `/dev/null` (see Chapter 15). To make the script a little more self-directed, the URL is hard-coded into the script:

```
#
quote_url=www.quotationspage.com/qotd.html
#
```

Now these two new changes to the daily inspirational quote script can be tested:

```
$ ./get_quote.sh
$
$ cat /tmp/daily_quote.txt
Selected from Michael Moncur's Collection of Quotations
- September 23, 2015
Horse sense is the thing a horse has which keeps it from
betting on people. W. C. Fields (1880 - 1946)
$
```

That works perfectly! The daily inspiration quote was extracted from the website's data and stored in a text file. You may have noticed by now that this quotation is less a traditional inspirational quote and more a humorous quote. Just know that some people find humor inspirational!

For your review, here's the final daily inspirational quote script with all the needed checks and changes:

```
#!/bin/bash
#
# Get a Daily Inspirational Quote
#####
#
# Script Variables ####
#
quote_url=www.quotationspage.com/qotd.html
#
# Check url validity ###
#
```

```

check_url=$(wget -nv --spider $quote_url 2>&1)
#
if [[ $check_url == *error404* ]]
then
    echo "Bad web address"
    echo "$quote_url invalid"
    echo "Exiting script..."
    exit
fi
#
# Download Web Site's Information
#
wget -o /tmp/quote.log -O /tmp/quote.html $quote_url
#
# Extract the Desired Data
#
sed 's/<[^>]*//g' /tmp/quote.html |
grep "$(date +%B' '%-d, ' '%Y)" -A2 |
sed 's/>//g' |
sed '/&nbsp;/{n ; d}' |
gawk 'BEGIN{FS="&nbsp;"} {print $1}' |
tee /tmp/daily_quote.txt > /dev/null
#
exit

```

This script is an excellent opportunity to try out some of your newly learned script writing and command line skills. The following are a few suggested changes for the daily inspirational quote script that you can attempt on your own:

- Change the website to your favorite quotation or sayings website, and make the necessary changes to the quote extraction commands.
- Try different `sed` and `gawk` commands for extracting the daily quotation.
- Set up the script to run daily on an automated basis via `cron` (see Chapter 16).
- Add a command to display the quote text file at certain times, such as when you first log in for the day.

Reading your daily quotes can inspire you. They may just inspire you to get out of that next business meeting. The next chapter section helps you write a script that does just that.

## Generating an Excuse

You've been there. That endless staff meeting that is full of unimportant information. You would really rather be working on that fascinating bash shell script project back at your desk. Here's a little fun script you can use to get out of the next staff meeting.

Short Message Service (SMS) allows text messages to be sent between cell phones. However, you can also use SMS to send text messages directly from e-mail or the command line. The script in this section allows you to construct a text message to be sent at a specified time directly to your phone. Receiving a “critical” message from your Linux system is the perfect excuse for leaving a staff meeting early.

### Understanding the required functions

You can send an SMS message from the command line in several ways. One way is via your system’s e-mail using your phone carrier’s SMS service. Another way is using the `curl` utility.

#### Learning about `curl`

Similar to `wget`, the `curl` utility allows you to transfer data from a particular web server. Unlike `wget`, it also allows you to transfer data to a web server. Transferring data to a particular web server is exactly what is needed here.

#### TIP

Some Linux distributions, such as Ubuntu, may not have the `curl` command installed by default. To install it, type `sudo apt-get install curl`. Also, you can find more information about installing software packages in Chapter 9.

Besides the `curl` utility, you need a website that provides free SMS message transfer. The one used here for this script is `http://textbelt.com/text`. This website allows you to send up to 75 text messages per day for free. You need it only for one text message, so it should be no problem.

#### TIP

If your company already uses an SMS provider, such as `http://sendhub.com` or `http://eztexting.com`, you can use those sites in your script instead. Be aware that the syntax needs to change depending upon those SMS provider’s requirements.

To use `curl` and `http://textbelt.com/text` to send yourself a text message, you need to use the following syntax:

```
$ curl http://textbelt.com/text \  
-d number=YourPhoneNumber \  
-d "message=Your Text Message"
```

The `-d` option tells `curl` to send specified data to the website. In this case, the website needs particular data sent in order to send a text message. This data includes



*YourPhoneNumber*, which is your cell phone number starting with the area code. And it also includes *Your Text Message*, which is the text message you desire to send.

## NOTE

The `curl` utility can handle much more than simply transferring data to and from a web server. It can handle many other network protocols, such as FTP, without any human intervention as well. Look at the man pages for `curl` to discover its rich power.

When the message is sent, the website provides a success message, `"success": true`, if no problems occurred:

```
$ curl http://textbelt.com/text \
> -d number=3173334444 \
> -d "message=Test from curl"
{
  "success": true
}$
$
```

Or it provides a fail message, `"success": false`, if data, such as the phone number, is incorrect:

```
$ curl http://textbelt.com/text \
-d number=317AAABBBB \
-d "message=Test from curl"
{
  "success": false,
  "message": "Invalid phone number."
}$
$
```

## NOTE

If your cell phone carrier is not in the United States of America, it is likely that `http://textbelt.com/text` will not work for you. You can try `http://textbelt.com/Canada` if your cell phone carrier is in Canada. If your cell phone carrier is located elsewhere, try `http://textbell.com/intl` instead. For additional help, see `http://textbelt.com`.

The success/fail messages are very helpful, but they are unwanted for the script. To remove these messages, simply redirect `STDOUT` to `/dev/null` (see Chapter 15). Unfortunately, now `curl` supplies undesired output:

```
$ curl http://textbelt.com/text \
> -d number=3173334444 \
> -d "message=Test from curl" > /dev/null
```

```
% Total      % Received % Xferd  Average Speed...
              Dload  Upload...
0      21      0      21      0      45      27      58 ...
$
```

The preceding snipped listing shows various statistics, which may be helpful when debugging your `curl` command. However, for the script, this information must be suppressed. Fortunately, the `curl` command has a `-s` option, which makes it silent:

```
$ curl -s http://textbelt.com/text \
> -d number=3173334444 \
> -d "message=Test from curl" > /dev/null
```

That is much better. The `curl` command is ready to be put into a script. However, before looking at the script, one more topic needs to be addressed: sending text messages via e-mail.

### Choosing to use e-mail

If you choose not to use the text message relay service provided by `http://textbelt.com/text` or if for some reason it doesn't work for you, you can always substitute sending a text message via e-mail. This section briefly covers how to accomplish this substitution.

#### CAUTION

If your cell phone carrier is not in the United States of America, it is likely that this web service will not work for you. Also, your cell phone carrier may block SMS messages from this site. In this case, you must attempt to use e-mail instead.

Whether or not e-mail works as a substitute depends upon your cell phone carrier. If your cell phone carrier has an SMS gateway, you are in luck. Contact your cell phone carrier and find out the name of the gateway. Often, it is something similar to `txt.att.net` or `vtext.com`.

#### TIP

You can often find out your cell phone carrier's SMS gateway on your own via the Internet. One great site listing various SMS gateways, along with usage tips, is `http://martinfitzpatrick.name/list-of-email-to-sms-gateways/`. If you cannot find your carrier there, use your favorite search engine to locate it.

The basic syntax for sending a text message via e-mail is as follows:

```
mail -s "your text message" your_phone_number@your_sms_gateway
```

**NOTE**

If the `mail` command does not work on your Linux system, you need to install the `mailutils` package. See Chapter 9 for a review of installing software packages.

Unfortunately, after you enter the syntax, you must type your message and press Ctrl+D to send the text message. This is similar to sending a regular e-mail (see Chapter 24). Using this method doesn't work well in a script. Instead, you can store your e-mail message in a file and use it to send a text message. The basic idea for this method is as follows:

```
$ echo "This is a test" > message.txt
$ mail -s "Test from email" \
3173334444@vtext.com < message.txt
```

Now the e-mail syntax is more compatible with a script. However, be aware that many problems may exist with this approach. First, you must have a mail server running on your system (see Chapter 24). Secondly, your phone service provider may block SMS messages coming from your system via e-mail. This is often true, if you are attempting this method from your home.

**TIP**

If your phone service provider blocks SMS messages coming from your system, you can use a cloud-based e-mail provider as an SMS relay. Use your favorite Internet browser and search for the words `SMS relay your_favorite_cloud_email` and see what sites come up.

Although sending a text message via e-mail is a potential alternative, it can be fraught with problems. If you can, it is much easier to use a free SMS relay website and the `curl` utility. The script in the next section uses `curl` to send a text message to the phone of your choice.

## Creating the script

After you have the required functions, creating the script to send a text message is fairly simple. You just need a few variables and the `curl` command.

You need three variables for the script. Setting up these particular data items as variables makes it easier if any of this information changes. The variables are shown here:

```
#
phone="3173334444"
SMSrelay_url=http://textbelt.com/text
```

```
text_message="System Code Red"
#
```

The only other needed item is the `curl` utility. Thus, here is the entire send a text message script:

```
#!/bin/bash
#
# Send a Text Message
#####
#
# Script Variables ####
#
phone="3173334444"
SMSrelay_url=http://textbelt.com/text
text_message="System Code Red"
#
# Send text #####
#
curl -s $SMSrelay_url -d \
number=$phone \
-d "message=$text_message" > /dev/null
#
exit
```

If you see this script as simple and easy, you are right! Even more important, that means you have learned a great deal about shell script writing. Even easy scripts need to be tested, so be sure to test this script using your cell phone number in the `phone` variable before continuing.

### TIP

While you are testing your script, be aware that this website, <http://textbelt.com/text>, does not allow you to send more than three text messages to the same phone number in less than three minutes.

To have a text message sent to you at a desired time, you must employ the `at` command. If you need a reminder, the `at` command was covered in Chapter 16.

First, you can test the use of the `at` command with your new script. Have the `at` utility execute the script by using the `-f` option along with the script's file name, `send_text.sh`, in this case. Have the script run immediately using the `Now` option:

```
$ at -f send_text.sh Now
job 22 at 2015-09-24 10:22
$
```

The script runs instantly. However, it may be a minute or two before you receive the text message on your phone.

To have the script run at another time, you simply use other `at` command options (see Chapter 16). In the following example, the script is run 25 minutes from the current time.

```
$ at -f send_text.sh Now + 25 minutes
job 23 at 2015-09-24 10:48
$
```

Note in the example, the `at` command provides an informational message when the script is submitted. The date and time listed in that message is when the script will execute.

What fun! Now you have a script utility that will be of assistance when you need an excuse to get out of that staff meeting. Better yet, you could modify the script to send yourself truly serious system messages that need to be addressed.

## Summary

This chapter showed how to put some of the shell-scripting information presented in the book to use for fun little shell scripts. Each script reinforced material covered in the chapters along with a few new commands and ideas.

The chapter demonstrated how to send a message to another user on the Linux system. The script checked to see whether the user was logged on to the system and whether the user allowed messaging. After those checks were made, the passed message was sent using the `write` command. Included were some suggestions for modifying this script, which improve your shell-scripting abilities.

The next section walked you through obtaining website information using the `wget` utility. The created script pulled a quote from the web. After retrieval, the script used several utilities to pull out the actual quote text. These now familiar commands included `sed`, `grep`, `gawk`, and the `tee` command. For this script, suggestions were made for how the script could be modified. These are well worth pursuing to solidify and improve your new skills.

The chapter ended with a very fun and simple script for sending yourself a text message. We explored the `curl` utility, along with SMS concepts. Although this is a fun script, it can be modified and used for more serious purposes.

Thanks for joining us on this journey through the Linux command line and shell scripting. We hope you've enjoyed the journey and have learned how to get around on the command line and how to create shell scripts to save time. But don't stop your command line education here. There's always something new being developed in the open source world, whether it's a new command line utility or a full-blown shell. Stay in touch with the Linux community and follow along with the new advances and features.