Writing a Shell Script From Scratch

Writing shell scripts can be rather daunting, primarily because the shell isn't the most friendly of languages to use. However, I hope to show you in this tutorial that shell scripting is actually not as tough or as scary as you might expect.

For this tutorial, we'll write a script that makes the process of using the Jasmine test framework [1] a little bit easier. Actually, I wouldn't use this script today; I would use Grunt.js [2] or something similar. However, I wrote this script before Grunt was around, and I found that writing it proved to be an excellent way to get more comfortable with shell scripting, so that's why we're using it.

> *One note: this tutorial is loosely associated with my upcoming Tuts+ Premium [3] course, "Advanced Command Line Techniques." To learn more about pretty much anything in this tutorial, stay tuned for that course's release. Hereafter in this tutorial, it will be referred to as "the course."*

So, our script, which I call `jazz`, will have four main features:

- It will download Jasmine from the web, unzip it, and delete the sample code.
- It will create JavaScript files and their associated spec files, and pre-fill them with a bit of template code.
- It will open the tests in the browser.
- It will display the help text, which outlines the above.

Let's begin with the script file.

Step 1 - Creating the File

Writing a shell script is only useful if you can use it from the terminal; to be able to use your custom scripts on the terminal, you need to put them in a folder that is in your terminal's `PATH` variable (you can see your `PATH` variable by running `echo $PATH`). I've created a `~/bin` folder (where ~ is the home directory) on my computer, and that's where I like to keep custom scripts (if you do the same, you'll have to

add it to your path). So, just create a file, called `jazz`, and put it in your folder.

Of course, we'll also have to make that file executable; otherwise, we won't be able to run it. We can do this by running the following command:

```
chmod +x jazz
```

Now that we can actually execute the script, let's add a very important part. All shell scripts should begin with a shebang [4]). As Wikipedia says, this should be the first line of the script; it states what interpreter, or shell, this script should be run with. We're just going to use a basic, standard shell:

```
#!/bin/sh
```

All right, with all that set up, we're ready to start writing the actual code.

Step 2 – Outlining the Script Flow

Earlier, I pointed out what the different features of our shell script should be. But how will the script know which feature to run? We'll use a combination of a shell parameter and a case statement. When running the script from the command line, we'll use a sub-command, like this:

```
jazz init
```

```
jazz create SomeFile
```

```
jazz run
```

```
jazz help
```

This should look familiar, especially if you've used Git:

```
git init
```

```
git status
```

```
git commit
```

Based on that first parameter (`init`, `create`, `run`, `help`), our case statement will decide what to run. However, we do need a default case: what happens if no first parameter is given, or we get an unrecognized first parameter? In those cases, we'll show the help text. So, let's get started!

Step 3 – Writing the Help Text

We begin with the `if` statement that checks for our first parameter:

```
if [ $1 ]

then

# do stuff

else

# show help

fi
```

You might be a little confused at first, because a shell `if` statement is pretty different from a "regular" programming language's `if` statement. To get a better understanding of it, watch the screencast on conditional statements in the course. This code checks for the presence of a first parameter (`$1`); if it's there, we'll execute the `then` code; `else`, we'll show the help text.

It's a good idea to wrap the printing of the help text in a function, because we need to call it more than once. We do need to define the function before it is called, so we'll put it at the top. I like this, because now, as soon as I open the file, I see the documentation for the script, which can be a helpful reminder when returning to code you haven't seen in a while. Without further ado, here's the `help` function:

```
function help () {
```

```
echo "jazz - A simple script that makes using the Jasmine testing framework in
a standalone project a little simpler."

echo "

echo "    jazz init                  - include jasmine in the project";

echo "    jazz create FunctionName   - creates ./src/FunctionName.js
./spec/FunctionNameSpec.js";

echo "    jazz run                   - runs tests in browser";

}
```

Now, just replace that `# show help` function with a call to the `help` function.

```
else

help

fi
```

Step 4 – Writing the Case Statement

If there is a first parameter, we need to figure out which it is. For this, we use a `case` statement:

```
case "$1" in

init)

;;

create)

;;
```

```
run)

;;

*)

help

;;

esac
```

We pass the first parameter to the `case` statement; then, it should match one of four things: "init", "create", "run", or our wildcard, default case. Notice that we don't have an explicit "help" case: that's just our default case. This works, because anything other than "init", "create", and "run" aren't commands we recognize, so it should get the help text.

Now we're ready to write the functional code, and we'll start with `jazz init`.

Step 5 – Preparing Jasmine with `jazz init`

All the code we write here will go within our `init)` case, from the case statement above. The first step is to actually download the standalone version of Jasmine, which comes in a zip file:

```
echo "Downloading Jasmine ..."

curl -sO $JASMINE_LINK
```

We first echo a little message, and then we use `curl` to download the zip. The `s` flag makes it silent (no output) and the `O` flag saves the contents of the zip to a file (otherwise, it would pipe it to standard out). But what's with that `$JASMINE_LINK` variable? Well, you could put the actual link to the zip file there, but I prefer to put it in a variable for two reasons: first, it keeps us from repeating part of the path, as you'll see in a minute. Second, with that variable near the top of the file, it makes it easy to change the version of Jasmine we're

using: just change that one variable. Here's that variable declaration (I put it outside the `if` statement, near the top):

```
JASMIME_LINK="http://cloud.github.com/downloads/pivotal/jasmine/jasmine-
standalone-1.3.1.zip [5]"
```

Remember, no spaces around the equals-sign in that line.

Now that we have our zip file, we can unzip it and prepare the contents:

```
unzip -q <code>basename $JASMINE_LINK</code>

rm -rf <code>basename $JASMINE_LINK</code> src/*.js spec/*.js
```

In two of these lines, we're using `basename $JASMINE_LINK`; the `basename` command just reduces a path to the base name: so `path/to/file.zip` becomes just `file.zip`. This allows us to use that `$JASMINE_LINK` variable to reference our local zip file.

After we unzip, we'll delete that zip file, as well as all the JavaScript files in the `src` and `spec` directories. These are the sample files that Jasmine comes with, and we don't need them.

Next, we have a Mac-only issue to deal with. By default, when you download something from the internet on a Mac, when you try to run it for the first time, you'll be asked to confirm that you want to run it. This is because of the extended attribute `com.apple.quarantine` that Apple puts on the file. We need to remove this attribute.

```
if which xattr > /dev/null && [ "<code>xattr SpecRunner.html</code>" =
"com.apple.quarantine" ]

then

xattr -d com.apple.quarantine SpecRunner.html

fi
```

We begin by checking for the presence of the `xattr` command, because it doesn't

exist on some Unix systems (I'm not sure, but it may be a Mac-only program). If you watched the course screencast on conditionals, you'll know that we can pass any command to `if`; if it has an exit status of anything but `0`, it's false. If `which` finds the `xattr` command, it will exit with `0`; otherwise, it will exit with `1`. In either case, `which` will display some output; we can keep that from showing by redirecting it to `/dev/null` (this is a special file [6] that discards all data written to it).

That double ampersand is an boolean AND; it's there for the second condition we want to check. That is, does the `SpecRunner.html` have that attribute? We can simply run the `xattr` command on the file, and compare it's output to the string we're expecting. (We can't just expect the file to have the attribute, because you can actually turn this feature off in Mac OS X, and we'll get an error when trying to remove it if the file doesn't have the attribute).

So, if `xattr` is found and the file has the attribute, we'll remove it, with the `d` (for delete) flag. Pretty straightforward, right?

The final step is to edit `SpecRunner.html`. Currently, it contains scripts tags for the sample JavaScript files that we deleted; we should also delete those scripts tags. I happen to know that those script tags span lines 12 to 18 in the files. So, we can use the stream editor `sed` to delete those lines:

```
sed -i "" '12,18d' SpecRunner.html

echo "Jasmine initialized!"
```

The `i` flag tells `sed` to edit the file in place, or to save the output from the command to the same file we passed in; the empty string after the flag means that we don't want `sed` to back up the file for us; if you wanted that, you could just put a file extension in that string (like `.bak`, to get `SpecRunner.html.bak`).

Finally, we'll let the user know that Jasmine has been initialized. And that's it for our `jazz init` command.

Step 6 – Creating Files with `jazz create`

Next up, we're going to let our users create JavaScript files and their associated spec files. This portion of the code will go in the “create" section of the `case` statement we wrote earlier.

```
if [ $2 ]

then

# create files

else

echo "please include a name for the file"

fi
```

When using `jazz create`, we need to include a name for the file as the second parameter: `jazz create View`, for example. We'll use this to create `src/View.js` and `spec/ViewSpec.js`. So, if there isn't a second parameter, we'll remind the user to add one.

If there is a file name, we'll start by creating those two files (inside the `then` part above):

```
echo "function $2 () {\n\n}" > src/$2.js

echo "describe('$2', function () {\n\n});" > spec/$2Spec.js
```

Of course, you can put whatever you want into your `src` file. I'm doing something basic here; so `jazz create View` will create `src/View.js` with this:

```
function View () {

}
```

You could replace that first `echo` line with this:

```
echo "var $2 = (function () {\n\tvar $2Prototype = {\n\n\t};\n\n\treturn
```

```
{\n\t\tcreate : function (attrs) {\n\t\t\tvar o =
Object.create($2Prototype);\n\t\t\textend(o, attrs);\n\t\t\treturn o;\n\t\t}\n
\t};\n}());" > src/$2.js
```

And then `jazz create View` will result in this:

```
var View = (function () {

var ViewPrototype = {

};

return {

create : function (attrs) {

var o = Object.create(ViewPrototype);

extend(o, attrs);

return o;

}

};

}());
```

So, really, your imagination is the limit. Of course, you'll want the spec file to be the standard Jasmine spec code, which is what I have above; but you can tweak that however you like as well.

The next step is to add the script tags for these files to `SpecRunner.html`. At first, this might seem tricky: how can we add lines to the middle of a file programmatically? Once again, it's `sed` that does the job.

```
sed -i "" "11a\\
```

```
  <script src='src/$2.js'></script>\\

  <script src='spec/$2Spec.js'></script>

  " SpecRunner.html
```

We start just like we did before: in-place edit without a backup. Then our command: at line 11, we want to append the two following lines. It's important to escape the two new-lines, so that they'll appear in the text. As you can see, this just inserts those two scripts tags, exactly what we need for this step.

We can end with some output:

```
  echo "Created:"

  echo "\t- src/$2.js"

  echo "\t- spec/$2Spec.js"

  echo "Edited:"

  echo "\t- SpecRunner.html"
```

And that's `jazz create`!

Step 7 – Running the Specs with `jazz run`

The last step is to actually run the tests. This means opening the `SpecRunner.html` file in a browser. There's a bit of a caveat here. On Mac OS X, we can use the `open` command to open a file in it's default program; this won't work on any other OS, but that's the way I do it here. Unfortunately, there's no real cross-platform way to do this, that I know of. If you're using this script under Cygwin on Windows, you can use `cygstart` in place of `open`; otherwise, try googling "[your OS] shell script open browser" and see what you come up with. Unfortunately, some versions of Linux (at least Ubuntu, in my experience) have an `open` command that's for something completely different. All this to say, your mileage with the following may vary.

```
if [ "`which open`" = '/usr/bin/open' ]

then

open SpecRunner.html

else

echo "Please open SpecRunner.html in your browser"

fi
```

By now, you know exactly what this does: if we have `open`, we'll open `SpecRunner.html`, otherwise, we'll just print a message telling the user to open the file in the browser.

Originally, that `if` condition looked like this:

```
if which open > /dev/null
```

As we did with `xattr`, it just checked for existence of `open`; however, since I found out that there's a different `open` command on Linux (even on my Ubuntu server, which can't even open a browser!), I figured it might be better to compare the path of the `open` program, since the Linux one is at `/bin/open` (again, at least on Ubuntu server).

All this extra wordiness about `open` might sound like an excuse for my lack of a good solution, it actually points out something important about the command line. Don't mistake understanding the terminal with understanding a computer's configuration. This tutorial, and the associated course, have taught you a little bit more about the Bash shell (and the Z shell), but that doesn't mean each computer you use will be configured the same; there are many ways to install new commands (or different versions of commands), as well as remove commands. *Caveat developer*.

Well, that's the entire script! Here it is again, all together:

```
#! /bin/sh
```

```
function help () {

echo "jazz - A simple script that makes using the Jasmine testing framework in
a standalone project a little simpler."

echo ""

echo "    jazz init                  - include jasmine in the project";

echo "    jazz create FunctionName   - creates ./src/FunctionName.js
./spec/FunctionNameSpec.js";

echo "    jazz run                   - runs tests in browser";

}

JASMIME_LINK="http://cloud.github.com/downloads/pivotal/jasmine/jasmine-
standalone-1.3.1.zip [7]"

if [ $1 ]

then

case "$1" in

init)

echo "Downloading Jasmine . . ."

curl -sO $JASMIME_LINK

unzip -q `basename $JASMIME_LINK`

rm `basename $JASMIME_LINK` src/*.js spec/*.js

if which xattr > /dev/null && [ "`xattr SpecRunner.html`" =
"com.apple.quarantine" ]

then
```

```
xattr -d com.apple.quarantine SpecRunner.html

fi

sed -i "" "12,18d" SpecRunner.html

echo "Jasmine initialized!"

;;

create)

if [ $2 ]

then

echo "function $2 () {\n\n}" > ./src/$2.js

echo "describe('$2', function () {\nit('runs');\n});" > ./spec/$2Spec.js

sed -i "" "11a\\

<script src='src/$2.js'></script>\\

<script src='spec/$2Spec.js'></script>

" SpecRunner.html

echo "Created:"

echo "\t- src/$2.js"

echo "\t- spec/$2Spec.js"

echo "Edited:"

echo "\t- SpecRunner.html"

else
```

```
  echo 'please add a name for the file'
  fi
  ;;
  "run")
  if [ "`which open`" = '/usr/bin/open' ]
  then
  open ./SpecRunner.html
  else
  echo "Please open SpecRunner.html in your browser"
  fi
  ;;
  *)
  help;
  ;;
  esac
  else
  help;
  fi
```

Well, go on, give it a try!

```
  mkdir project
```

```
cd project

jazz init

jazz create Dog

# edit src/Dog.js and spec/DogSpec.js

jazz run
```

By the way, if you want to have some more fun with this project, you can find it on Github [8].

Conclusion

So there you have it! We've just written an intermediate level shell script; that wasn't so bad, now, was it? Don’t forget to stay tuned for my upcoming Tuts+ Premium [9] course; you'll learn a lot more about many of the techniques used in this article, as well as countless others. Have fun on the terminal!

1. http://pivotal.github.com/jasmine/
2. http://gruntjs.com/
3. http://tutsplus.com/
4. https://en.wikipedia.org/wiki/Shebang_(Unix
5. http://cloud.github.com/downloads/pivotal/jasmine/jasmine-standalone-1.3.1.zip
6. https://en.wikipedia.org/wiki//dev/null
7. http://cloud.github.com/downloads/pivotal/jasmine/jasmine-standalone-1.3.1.zip
8. https://github.com/andrew8088/jazz
9. http://tutsplus.com/