10 Tips for Writing Efficient Bash Scripts

Bash is the default command line interface for many Linux distributions and a powerful scripting language. Here are some suggestions that will keep your Bash scripts efficient and lean. Feel free to comment with your own suggestions.

1. **Avoid Full Paths to Bash Builtins**

   Bash has many builtins that can be used instead of calling external commands. You should leverage the builtin commands whenever possible since it avoids calling a subcommand from the system.

   Since Bash has builtins for some commands found in /bin and /usr/bin (such as echo), avoid using the full path for these commands and the builtin will be used.

   ```
   # avoid this
   /bin/echo "hello"
   ```

   Use the Bash builtin instead:

   ```
   echo "hello"
   ```

   Other bash builtins include: test, read, declare, eval, let pushd and popd. See the Bash man page for a full listing of builtins.

2. **Avoid External Commands for Integer Math**

   Bash also provides builtins that can be used for integer arithmetic. Only use /usr/bin/bc if you need to do floating point arithmetic. Integer calculations can be made with these Bash builtins:

   ```
   four=$(( 2 + 2 ))
   four=$[ 2 + 2 ]
   let four="2 + 2"
   ```

3. **Avoid using Cat**

   Tools like Grep, Awk and Sed will take files as arguments. There is rarely a need to use /bin/cat. For instance, the following is unnecessary:

   ```
   # avoid this
   cat /etc/hosts | grep localhost
   ```

   Instead, use Grep's native ability to read files:

   ```
   grep localhost /etc/hosts
   ```

4. **Avoid Piping Grep to Awk**

   If using Awk, you can often eliminate the need for grep. Try not to pipe Grep to Awk:

   ```
   # avoid this
   grep error /var/log/messages | awk '{ print $4 }'
   ```

   Use Awk's native ability to parse text and save yourself a command.

   ```
   awk '/error/ { print $4 }' /var/log/messages
   ```

5. **Avoid Piping Sed to Sed**

   Sed can take more than one command in a single execution. Avoid piping sed to sed.

   ```
   # avoid this
   sed 's/hello/goodbye/g' filename | sed 's/monday/friday/g'
   ```

   Instead, use sed -e or delimit the sed expressions with a semicolon (;)

   ```
   sed -e 's/hello/goodbye/g' -e 's/monday/friday/g' filename
   sed -e 's/hello/goodbye/g; s/monday/friday/g' filename
   ```

6. **Use Double Brackets for Compound and Regex Tests**

   The [or test builtins can be used to test expressions, but the [[builtin operator additionally provides compound commands and regular expression matching.

   ```
   if [[ expression1 || expression2 ]]; then do_something; fi
   if [[ string =~ regex ]]; then do_something; fi
   ```

7. **Use Functions for Repetitive Tasks**

Break your script up into pieces and use functions to conduct repetitive tasks. Functions can be declared like so:

```
function_name() {
  do_something
  return $?
}
```

Make your functions usable by more than one shell script by sourcing a functions file from the various scripts. You can source another file in Bash using the . builtin.

```
#!/bin/bash
. /path/to/shared_functions
```

See the Bash man page, or my article on Bash functions for more information.

8. **Use Arrays Instead of Multiple Variables**

   Bash arrays are very powerful. Avoid using unnecessary variables:

```
# avoid this
color1='Blue'
color2='Red'
echo $color1
echo $color2
```

   Instead, use Bash arrays.

```
colors=('Blue' 'Red')
echo ${colors[0]}
echo ${colors[1]}
```

   Check out my article on Bash arrays for more details.

9. **Use /bin/mktemp to Create Temp Files**

   Need a temporary file? Use /bin/mktemp to create temporary files or folders.

```
tempfile=$(/bin/mktemp)
tempdir=$(/bin/mktemp -d)
```

10. **Use /bin/egrep or /bin/sed for Regex Pattern Matching**

    Think you need Perl? Check out Sed or Egrep (grep -e) for regex pattern matching.

```
# grep for localhost or 127.0.0.1 in /etc/hosts
egrep 'localhost|127\.0\.0\.1' /etc/hosts

# print pattern localhost.* in /etc/hosts
sed -n 's/localhost.*/&/p' /etc/hosts
```