UTSA

The University of Texas at San Antonio

University Technology Solutions

Express Bash Scripting Tutorial Part 4

Quickly Learn Bash Scripting in Linux

Brent League

Overview

- File Operations
  - Reading Files
  - Writing Files
  - File Checksums
- Sleep and Process
  - Using the Sleep Command
  - Watching a Process
- Interactive Scripts
  - Getting Input from Users
  - Handling Bad Data

File Operations: Reading Files

- Read a text file into a script using the read command & redirection
- After this exercise you will know how to read a text file into a script and display the contents of the text file
- Let's start off by using your text editor to create a file called names.txt, enter four or five names on separate lines, and then save it. We'll be able to read it with our script later

File Operations: Reading Files

Using your text editor, let's create a script called `reader.sh` with the following information in it:

```
#!/usr/bin/env bash
COUNT=1

while IFS='' read -r LINE
do
  echo "LINE $COUNT: $LINE"
  ((COUNT++))
done < "$1"
```

File Operations: Reading Files

After you save the script from the previous slide, type in `./reader.sh names.txt` – the output we should see is all the names from our text file and their line numbers beginning with Line 1 since we defined a counter (`COUNT=1`) that told our script to begin counting at 1

```
LINE 1: Tammy
LINE 2: Kelly
LINE 3: Tina
LINE 4: Bob
LINE 5: David
```

File Operations: Writing Files

- There are two ways to redirect output of a script to a file
  - Let's use the `reader.sh` script to redirect the list of names to a new file. Type the following:

```
./reader.sh names.txt > output.txt
```

  - When using a single greater than sign, it will create a new file – but if the file name already exists, the existing data will be overwritten.

  - Now that output.txt has been created with data in it, let's explore how to append data to it instead of overwriting it. It's as simple as using the same command as above but use TWO greater than symbols.

```
./reader.sh names.txt >> output.txt
```

File Operations: File Checksums

- A checksum is a value that is used to validate the integrity of a file. Let's explore how we can detect if a file has been tampered with
  - From the terminal window, type the following: `cksum names.txt` (we should see the checksum value and the number of bytes in the file)
  - Let's open the txt file with our text editor and add an extra character to one of the names. Then, run the same command: `cksum names.txt` The checksum should change and so should the number of bytes
  - Finally, let's edit the text file and remove the extra character, and run `cksum names.txt` again. The checksum and the byte count should return to the original values.

Sleep & Process: Using the Sleep Command

Sometimes we need a script or process to only run intermittently, or start later. The sleep command can be used to have our script go to sleep until it is needed, and then execute the remaining commands. Create a new script called `delay.sh`

```
#!/usr/bin/env bash

DELAY=$1
if [[ -z $DELAY ]]
then
  echo "You must supply a delay"
  exit 1
fi

echo "Going to sleep for $DELAY seconds"
sleep $DELAY
echo "We are awake now"
exit 0
```

Watching a Process

We can have a script watch other processes and message when they terminate. Create a script called `proc.sh` and add the following lines of code to it:

```
#!/usr/bin/env bash

STATUS=0

if [[ -z $1 ]]
then
  echo "Please supply a PID"
  exit 1
fi

echo "Watching PID = $1"
while [[ $STATUS -eq 0 ]]
do
  ps $1 > /dev/null
#the command BELOW picks up the status of the LAST command ran - it will tell us
#if the PS command was successful
STATUS=$?
done

echo "Process $1 has terminated"
exit 0
```

Getting Input from Users

We can create an interactive script by using the “read” command to gather input from the user. Create a script called `prompt.sh` and add the following lines to it:

```
#!/usr/bin/env bash

read -p "What is your first name: " NAME
echo "Your name is: $NAME"
exit 0
```

How to Handle Bad Data

When you're asking users to provide input, they might not enter all of the data your script requires to execute properly, or the data may not be in the correct format. Let's explore how to address this.

In this example, we need two parameters from the user; their name (alphabetical characters, and their age (an integer).

Let's create a script and call it `user.sh`

The next slide shows the code that your `user.sh` script should contain.

The slides following the sample code will explain what each line accomplishes.

How to Handle Bad Data

```
#!/usr/bin/env bash
VALID=0

while [[ $VALID -eq 0 ]]
do
  read -p "Please enter your name and age: " NAME AGE
  if [[ ( -z $NAME ) || ( -z $AGE ) ]]
  then
    echo "You did not enter all of the requested information"
    continue
  elif [[ ! $NAME =~ ^[A-Za-z]+$ ]]
  then
    echo "Non alpha characters detected [$NAME]"
    continue
  elif [[ ! $AGE =~ ^[0-9]+$ ]]
  then
   echo "Non numerical character detected [$AGE]"
   continue
  fi
VALID=1
done
echo "Your name is $NAME and your age is $AGE"
exit 0
```

How to Handle Bad Data

Now we will explain what each line of code from the pervious slide performs

- `VALID=0` is a global variable that whether we consider the input valid or not
- Then we will create a “while” loop with by entering `$VALID -eq 0` – this says as long as valid equals zero, we’re going to continue in our “while” loop
- After the `do` statement ,we are going enter `read -p` this tells the script to put the user input on the same line as the prompt, which have entered in quotations. At the end of this line, we are going to enter our input variables: – `NAME and AGE`
- Our `if` statement is going to make sure we received data in both of the parameters we are expecting to have data in – the `-z` you see in the parentheses checks to see if the input value is empty
- In between our two variables, we have to pipes `||` - two pipe symbols are the same as an “or” statement, so if either one of these are true, we didn’t get enough parameters.

(continued on next slide)

How to Handle Bad Data

Explanation of each command (continued from the previous slide)

- It that's the case, then we will echo out something to let the user know that we didn't get what we needed. If that's the case, we will use the `continue` command to go back to the top of the loop
- Otherwise we will move to the `elif` line. The `!` is a "NOT" symbol, so we are reversing the logic of this test – then `$NAME =~` What we are doing on this line is performing a regular expression to make sure any characters in the name are alphabetic. So A through Z upper case or lower case are acceptable input. The `^` symbol you see next, tells the command to start from the beginning of the string. The `+` sign meant that this string must end here as well
- Next, we'll do a `then` in case this test *fails* and we will `echo` a statement to give the user a hint as to what is wrong, and follow it with a `continue` statement and go back to the top of the loop
- The next four lines beginning with the `elif`, are exactly the same, but we are looking for integers for the age value this time

(continued on next slide)

How to Handle Bad Data

Explanation of each command (continued from the previous slide)

- Next, we'll do a `fi` before we exit the loop
- Since we've passed all the validation steps, we're going to enter `VALID=1` and then close out the loop with `done`, and echo back the user's input in the final line
- Now that you have an understanding of each line in our script, let's execute it by typing `./user.sh` and then move to the next slide to see our output

How to Handle Bad Data

After entering `./user.sh`, I am prompted for my name and my age.

The text in black is what was echoed to the screen, and the text in red is what I entered.

```
Please enter your name and age: Brent 21
Your name is Brent and your age is 21
```

Take a few minutes to experiment with this script and enter your age and your name in the wrong order, or enter nothing at all and view the various results you get from the script.

UTSA

The University of Texas at San Antonio

University Technology Solutions

Express Bash Scripting Tutorial Part 4

Quickly Learn Bash Scripting in Linux

Brent League