Linux Scripting

A Guide for Beginners

Bash Scripting

Copyright Notice: Protection of Intellectual Property

This document, and its contents, is the intellectual property of DigiTalk. It is protected under copyright law and international treaties. Unauthorized use, reproduction, distribution, or resale of this document or any of its content, in whole or in part, is strictly prohibited.

Any infringement of our copyright will result in legal action and may subject the violator to both civil and criminal penalties.

For permissions and inquiries, please contact digitalk.fmw@gmail.com

By accessing or using this document, you agree to abide by these terms and conditions.

Thank you for respecting our intellectual property rights.

DigiTalk

https://digitalksystems.com/

Reach us at digitalk.fmw@gmail.com
DigiTalk Channel: https://www.youtube.com/channel/UCCGTnI9vvF_ETMhGUXGdFWw
Playlists: https://www.youtube.com/@digitalk.middleware/playlists
Weblogic Server Architecture: https://youtu.be/gNqeIfLjUqw

Copyright © [2023] DigiTalk. All rights reserved.

DigiTalk Udemy Courses and

Coupon Code (Embedded in URL)

SOA Suite Administration

https://www.udemy.com/course/mastering-oracle-soa-suite-12c-administration/?couponCode=3748CA8CCCF4A124B4E9

JBoss 8 Administration

https://www.udemy.com/course/mastering-jboss-eap-8-administration-from-intro-to-advanced/?couponCode=C0947AF96757C942530F

OHS Administration

https://www.udemy.com/course/mastering-oracle-ohs-http-12c-web-server-administration/?couponCode=6203B4E94AA374CFA326

Weblogic Server Administration

https://www.udemy.com/course/oracle-weblogic-server-12c-and-14c-administration/?couponCode=D6E8B65B3FACB040D423

You can write us on digitalk.fmw@gmail.com if coupon code expired.

Copyright © [2023] DigiTalk. All rights reserved.

Introduction to Linux Bash Scripting

Bash scripting is a powerful tool that allows users to automate tasks, manage system operations, and create complex workflows with simple text commands. Bash, which stands for "Bourne Again SHell," is the default command interpreter on most Unix-based systems, including Linux. It provides a way to interact with the operating system by executing commands, managing files, and performing various administrative tasks.

Bash scripting enables you to combine multiple commands into a script that can be executed as a program. Whether you're a system administrator, developer, or an enthusiast, learning Bash scripting can significantly enhance your ability to manage systems and perform tasks more efficiently.

Importance of the Shebang (#!/bin/bash)

At the beginning of a Bash script, you'll often see a line that starts with #!/bin/bash. This line is known as a shebang (or hashbang), and it serves a crucial purpose in scripting.

What is a Shebang?

The shebang is the combination of the characters #! followed by the path to an interpreter, which, in this case, is /bin/bash. This line tells the system which interpreter to use when executing the script. By specifying /bin/bash, you are instructing the system to use the Bash shell to interpret and execute the commands within the script.

Why Use the Shebang?

Script Portability:

- The shebang makes your script portable across different environments. By explicitly stating which interpreter to use, you ensure that the script runs consistently, regardless of the user's default shell.

Clarity:

- Including the shebang makes it clear to anyone reading the script which shell or interpreter is intended to execute the script. This is particularly useful in environments where multiple shells (e.g., sh, zsh, bash) are available.

Execution Without Prefix:

If you execute a script without specifying the interpreter explicitly (e.g., ./script.sh), the system uses the shebang to determine which interpreter to invoke. Without the shebang, the script might not execute as expected if the user's default shell is different from Bash.

Example of a Script with Shebang

```
#!/bin/bash
# Backup Script
source_dir="/home/user/data"
backup_dir="/home/user/backup"
# Create a backup
cp -r $source_dir $backup_dir
echo "Backup completed!"
```


Copyright © [2023] DigiTalk. All rights reserved.

In this example, the #!/bin/bash line at the top ensures that the script is executed using the Bash shell, regardless of the environment in which it is run.

Conclusion

Including the shebang (#!/bin/bash) at the beginning of your Bash scripts is a best practice that ensures your script is interpreted correctly, enhances portability, and provides clarity to anyone reading or executing the script.

Where to Begin: Getting Started with Bash

Before diving into scripting, it's essential to familiarize yourself with the basics of the Bash shell. Start by understanding the command line interface (CLI) and how it interacts with the operating system. Here are some foundational topics to begin with:

The Command Line Interface (CLI):

Understanding the command line is crucial. Learn how to navigate the filesystem, execute commands, and use basic utilities like ls, cd, pwd, cp, mv, rm, and cat.

Example:

```
# List the contents of the current directory
ls

# Change directory to /home
cd /home

# Print the current directory
pwd
```

Basic Shell Commands:

Learn the essential shell commands and their options. Explore how to use them in combination to perform more complex tasks.

Example:

```
# Create a new directory
mkdir my_directory

# Move a file into the new directory
mv myfile.txt my_directory/

# Copy a file
cp myfile.txt backup.txt

# Remove a file
rm backup.txt
```


Copyright © [2023] DigiTalk. All rights reserved.

File Permissions and Ownership:

Understanding file permissions is vital for security and proper system management. Learn how to check, modify, and understand permissions using chmod, chown, and chgrp.

Example:

 # Change file permissions to read, write, and execute for the owner

```
chmod 700 myfile.txt
```

 # Change the owner of a file

```
chown user:group myfile.txt
```

What to Learn First: Essential Bash Concepts

Once you are comfortable with the basic commands, the next step is to dive into Bash scripting fundamentals. Here are the core concepts to focus on:

Variables and Environment:

Variables in Bash are typically assigned using the = operator, without any spaces around the =. Here's how to assign different types of variables:

String (Character) Variables

String variables hold text data, which can include letters, numbers, and special characters.

Example:

```
 Assign a string to a variable
greeting="Hello, World!"
name="Alice"
# Access the variable
echo $greeting       # Output: Hello, World!
echo "Hello, $name"  # Output: Hello, Alice
```

Important Notes:

- No spaces should be around the = sign.
- Strings can be enclosed in double quotes ("), single quotes ('), or no quotes at all if the string has no spaces.
- Double quotes allow variable interpolation (i.e., the value of the variable is inserted), while single quotes do not.

Numeric Variables

Numeric variables store integer or floating-point numbers. Bash treats numbers as strings by default, so to perform arithmetic, you need to use (( )) or expr.

Example:

Copyright © [2023] DigiTalk. All rights reserved.

```
# Assign numbers to variables
num1=10
num2=20
# Perform arithmetic
sum=$((num1 + num2))
product=$((num1 * num2))
echo "Sum: $sum"          # Output: Sum: 30
echo "Product: $product"  # Output: Product: 200
```

Important Notes:

- When assigning a numeric value, no quotes are necessary.
- Arithmetic operations require special syntax like (( )) or expr.

Arithmetic and Logical Operations in Bash Scripting

In Bash scripting, arithmetic and logical operations are fundamental for making decisions, performing calculations, and controlling the flow of your script. Let's break down how to perform these operations with examples.

Arithmetic Operations

Bash supports basic arithmetic operations, such as addition, subtraction, multiplication, division, and modulus. These operations are typically performed using the (( )) syntax or the expr command.

1. Using (( )) for Arithmetic

The (( )) syntax is the preferred way to perform arithmetic operations in Bash because it is concise and supports various operators.

Addition (+)

```
#!/bin/bash
num1=5
num2=3
result=$((num1 + num2))
echo "Addition: $result"   # Output: Addition: 8
```

Subtraction (-)

```
#!/bin/bash
num1=5
num2=3
```


Copyright © [2023] DigiTalk. All rights reserved.

```
result=$((num1 - num2))
echo "Subtraction: $result"   # Output: Subtraction: 2
```

Multiplication (*)

```
#!/bin/bash
num1=5
num2=3
result=$((num1 * num2))
echo "Multiplication: $result"   # Output: Multiplication: 15
```

Division (/)

```
#!/bin/bash
num1=6
num2=3
result=$((num1 / num2))
echo "Division: $result"   # Output: Division: 2
```

Modulus (%): Remainder of the division

```
#!/bin/bash
num1=7
num2=3
result=$((num1 % num2))
echo "Modulus: $result"   # Output: Modulus: 1
```

2. Using expr for Arithmetic

expr is another way to perform arithmetic in Bash. It is slightly older and less commonly used than (( )), but it can still useful.

Addition

```
#!/bin/bash
num1=5
num2=3
result=$(expr $num1 + $num2)
echo "Addition using expr: $result"   # Output: Addition using expr: 8
```


Copyright © [2023] DigiTalk. All rights reserved.

Subtraction

```
#!/bin/bash

num1=5

num2=3

result=$(expr $num1 - $num2)

echo "Subtraction using expr: $result"   # Output: Subtraction using expr: 2
```

Multiplication

```
#!/bin/bash

num1=5

num2=3

result=$(expr $num1 \* $num2)

echo "Multiplication using expr: $result"   # Output: Multiplication using expr: 15
```

Division

```
#!/bin/bash

num1=6

num2=3

result=$(expr $num1 / $num2)

echo "Division using expr: $result"   # Output: Division using expr: 2
```

Logical Operations

Logical operations are used for decision-making in scripts. They help in evaluating conditions and determining the flow of execution. The most common logical operators in Bash are &&, ||, and !.

1. AND (&&)

The && operator is used to check if both conditions are true.

Example:

```
#!/bin/bash

age=25

if [[ $age -gt 18 && $age -lt 30 ]]; then

   echo "Age is between 18 and 30"

else

   echo "Age is not between 18 and 30"
```


Copyright © [2023] DigiTalk. All rights reserved.

```
fi
```

Explanation:

The script checks if $age is greater than 18 and less than 30.

If both conditions are true, it prints "Age is between 18 and 30"; otherwise, it prints "Age is not between 18 and 30".

2. OR (||)

The || operator is used to check if at least one of the conditions is true.

Example:

```
#!/bin/bash

num=5

if [[ $num -lt 3 || $num -gt 4 ]]; then
   echo "Number is either less than 3 or greater than 4"
else
   echo "Number is between 3 and 4"
fi
```

Explanation:

The script checks if $num is less than 3 or greater than 4.

If either condition is true, it prints "Number is either less than 3 or greater than 4"; otherwise, it prints "Number is between 3 and 4".

3. NOT (!)

The ! operator is used to negate a condition, i.e., it checks if a condition is not true.

Example:

```
!/bin/bash

file="myfile.txt"

if [[ ! -f $file ]]; then
   echo "File does not exist"
else
   echo "File exists"
fi
```


Copyright © [2023] DigiTalk. All rights reserved.

Explanation:

The script checks if myfile.txt does not exist using the ! operator.

If the file doesn't exist, it prints "File does not exist"; otherwise, it prints "File exists".

4. Combining Logical Operations

You can combine multiple logical operations to create complex conditions.

Example:

```
!/bin/bash

num=10

if [[ $num -gt 0 && ($num -lt 5 || $num -gt 8) ]]; then
    echo "Number is greater than 0 and either less than 5 or greater than 8"
else
    echo "Condition not met"
fi
```

Explanation:

The script checks if $num is greater than 0 and either less than 5 or greater than 8.

If the condition is met, it prints "Number is greater than 0 and either less than 5 or greater than 8"; otherwise, it prints "Condition not met".

Control Structures:

Control structures are crucial for decision-making and looping. Begin with if-else statements and loops (for, while, until).

Example:

```
# If-else statement
if [ -f "myfile.txt" ]; then
    echo "File exists"
else
    echo "File does not exist"
fi

# For loop
for i in 1 2 3 4 5; do
    echo "Number $i"
```


Copyright © [2023] DigiTalk. All rights reserved.

```
done
# While loop
count=1
while [ $count -le 5 ]; do
    echo "Count is $count"
    count=$((count + 1))
done
```

Functions:

Functions allow you to create reusable blocks of code. They help organize and simplify complex scripts.

Example:

```
# Define a function
greet() {
    echo "Hello, $1"
}
# Call the function
greet "Alice"
```

Script Execution and Debugging:

Learn how to execute scripts and pass arguments to them. Explore basic debugging techniques using echo statements and the set -x command.

Example:

```
# Execute a script with arguments
./myscript.sh arg1 arg2
# Enable debugging
set -x
echo "This is a debug message"
set +x
```

Examples to Illustrate Concepts

Now that you have an understanding of the basics, let's explore some practical examples to solidify these concepts:

Copyright © [2023] DigiTalk. All rights reserved.

Backup Script:

A simple script to back up a directory.

```
#!/bin/bash
# Backup Script
source_dir="/home/user/data"
backup_dir="/home/user/backup"
# Create a backup
cp -r $source_dir $backup_dir
echo "Backup completed!"
```

User Management Script:

A script to automate user creation.

```
#!/bin/bash
# User Management Script
if [ "$#" -ne 2 ]; then
    echo "Usage: $0 username password"
    exit 1
fi
user=$1
pass=$2
# Create a new user
useradd $user
# Set the password
echo $user:$pass | chpasswd
echo "User $user created successfully!"
```

Log File Monitoring:

A script to monitor a log file for a specific keyword and alert the user.

```
#!/bin/bash
# Log Monitoring Script
logfile="/var/log/system.log"
```


Copyright © [2023] DigiTalk. All rights reserved.

```
keyword="ERROR"
f grep -q $keyword $logfile; then
    echo "An error was found in the log file!"
else
    echo "No errors found."
fi
```


Copyright © [2023] DigiTalk. All rights reserved.

DISCLAIMER AND CONSENT

This document is being provided by DigiTalk as part of its effort to assist users in understanding and working with Linux. While every effort has been made to ensure the accuracy and reliability of the information presented in this document, there is a possibility of typographical errors or inaccuracies. DigiTalk does not guarantee the correctness or completeness of the content provided in this document.

Users of this document are encouraged to cross-reference the information presented here with official documentation available on their website or other authoritative sources. Any discrepancies or inaccuracies found in this document should be reported to us at digitalk.fmw@gmail.com.

By using this document, you acknowledge and consent to the following:

This document is not officially endorsed or verified by any other third party organization..

The Company makes no claims or guarantees about the accuracy or suitability of the information contained in this document.

Users are responsible for verifying and validating any information presented here for their specific use case.

DigiTalk disclaims any liability for any errors, omissions, or damages that may result from the use of this document.

If you discover any inaccuracies or errors in this document, please report them to digitalk.fmw@gmail.com, and the Company will endeavor to correct them as necessary.

This consent statement is provided to ensure transparency and understanding of the limitations of the information contained in this document. By using this document, you agree to abide by the terms and conditions outlined herein.

Copyright © [2023] DigiTalk. All rights reserved.