#90DaysOfDevOps Challenge - Day 4 -Basic Linux Shell Scripting for DevOps Engineers

Welcome to Day 4 of the #90DaysOfDevOps challenge. In today's session, we will explore the basics of Linux shell scripting and its relevance for DevOps engineers. Shell scripting is a powerful skill that allows you to automate tasks, manage system configurations, and streamline DevOps workflows. Let's dive in and understand the fundamentals of shell scripting.

What is Shell Scripting for DevOps?

In simple terms, shell scripting involves writing a series of commands in a script file to be executed by the shell interpreter. The shell acts as an interface between the user and the operating system, interpreting and executing the commands in the script.

For DevOps engineers, shell scripting plays a crucial role in automating repetitive tasks, performing system configurations, managing infrastructure, and orchestrating various tools and processes. It allows DevOps professionals to create custom workflows, automate deployments, monitor system health, and much more.

Shell scripting enables you to:

- **Automate tasks:** By writing scripts, you can automate repetitive tasks, saving time and reducing the risk of errors.
- **Customize workflows:** Shell scripts provide the flexibility to create custom workflows tailored to your specific requirements.
- **Perform system configurations:** Shell scripts allow you to configure and manage system settings, install software packages, and set up environments.
- **Integrate tools and processes:** You can use shell scripts to integrate different tools and processes into cohesive pipelines, enabling seamless collaboration and automation.

What is `#!/bin/bash`? Can we write `#!/bin/sh` as well?

The `#!/bin/bash` (shebang) is called an interpreter directive and specifies the interpreter to be used for executing the script. In this case, it indicates that the script should be interpreted using the Bash shell. Bash (Bourne Again SHell) is a widely used shell and the default shell on most Linux distributions.

Yes, you can write `#!/bin/sh` instead of `#!/bin/bash`. The `sh` interpreter is a more generic reference to the system's default shell. It may or may not be the same as Bash, depending on the system. Using `#!/bin/sh` allows the script to be executed by the system's default shell, which could be Bash or another compatible shell.

It's important to note that Bash offers more features and capabilities compared to the basic POSIX shell (`/bin/sh`). If your script requires specific Bash features, it's recommended to use `#!/bin/bash` to ensure compatibility.

Tasks:

1 - Writing a Shell Script to Print a Message

Let's start with a simple example of a shell script that prints a message. Create a new file, such as `task1.sh`, and add the following code:

```
bashCopy code#!/bin/bash
echo "I will complete the #90DaysOfDevOps challenge"
```

In this script, the shebang `#!/bin/bash` specifies the interpreter as Bash. The `echo` command is used to display the message "I will complete the #90DaysOfDevOps challenge" on the terminal when the script is executed.

Save the file, make it executable (`chmod 700 task1.sh`), and run it (`./task1.sh`). You should see the message displayed on the terminal.

2 - Taking User Input and Printing Variables

Shell scripting allows you to interact with users and handle input dynamically. Here's an example of a script that takes user input, stores it in a variable, and prints the variables.

```
bashCopy code#!/bin/bash

# variable
var="I'm a variable"

# Read user input and store it in the variable 'name'
echo "Enter your name: "
read name

# Print the variable 'name'
echo "My name is $name"

# Print the predefined variable 'var'
echo "I'm printing the variable 'var': $var"
```

In this script, the `read` command is used to prompt the user to enter their name, and the input is stored in the `name` variable. The variable `var` is predefined with a value. The script then prints the user's name and the predefined variable value.

Executing this script will prompt the user to enter their name. After entering the name, the script will display the entered name and the predefined variable value.

3 - Using If-Else Statements in Shell Scripting

Conditional statements are essential for decision-making in shell scripts. Let's look at an example that uses an if-else statement to compare two numbers.

```
bashCopy code#!/bin/bash

# Read user input and store it in the variable 'number1'
echo "Enter the first number: "
read number1

# Read user input and store it in the variable 'number2'
echo "Enter the second number: "
read number2

# Compare both numbers and specify if they are equal or not
if [ $number1 -eq $number2 ]; then
    echo "Both numbers are the same"
else
    echo "Both numbers are different"
fi
```

In this script, the user is prompted to enter two numbers. The script then uses the if-else statement to compare the numbers. If the numbers are equal, it displays the message "Both numbers are the same". Otherwise, it displays "Both numbers are different".

When you execute this script, it will ask you to enter two numbers. Based on the comparison, the script will print the corresponding message on the terminal.

Shell scripting provides powerful constructs like if-else statements, loops, and functions that allow you to build complex automation logic and decision-making within your scripts.

You've now learned the basics of Linux shell scripting for DevOps engineers. Shell scripting opens up a world of possibilities for automation, customization, and efficient management of your systems and processes.

Stay tuned for Day 5, where we will learn about advanced Linux Shell scripting for DevOps engineers with user management