Bash Scripting in DevOps

Written by **Zayan Ahmed** | 4 mins read

Introduction

Bash (Bourne Again SHell) is the default shell on most Linux distributions and macOS, and it's widely used in DevOps for automation, configuration management, and orchestration of tasks. Mastering bash scripting is essential for any DevOps engineer to automate workflows, manage systems, and handle repetitive tasks efficiently.

This document will guide you through the fundamentals of Bash scripting for DevOps, including best practices, examples, and use cases

1. Why Bash Scripting is Important in DevOps

Bash scripting is widely used in DevOps for the following reasons:

- **Automation**: Automate repetitive tasks such as system updates, backups, and service deployments.
- **Infrastructure Management**: Manage infrastructure as code (IaC) by integrating bash with tools like Terraform, Ansible, and Kubernetes.
- **CI/CD**: Facilitate Continuous Integration/Continuous Deployment pipelines by automating steps like code builds, tests, and deployments.
- **Configuration Management**: Modify and manage configuration files dynamically.
- **Monitoring**: Create custom monitoring scripts to gather system metrics or check the health of services.

---

2. Bash Script Basics

2.1 Comments

Comments make scripts more understandable. Any line starting with # is treated as a comment.

```
#!/bin/bash
# This is a single-line comment
```

2.2 Variables

Variables store values and can be used throughout the script.

Declaring Variables

```
#!/bin/bash
name="DevOps Engineer"
echo "Hello, $name"
```

Reading User Input

```
#!/bin/bash
read -p "Enter your name: " user_name
echo "Hello, $user_name"
```

2.3 Input/Output

Bash provides ways to read from and write to files, terminals, and command outputs.

Standard Input (stdin), Output (stdout), and Error (stderr)

- **> redirects output to a file.**
- **>> appends to a file.**
- **2> redirects errors.**

```
#!/bin/bash
echo "This is output" > output.txt
echo "This is error" 2> error.txt
```

Piping (|)

```
#!/bin/bash
ls -l | grep "log"
```

2.4 Conditions

If-Else Statements

```
#!/bin/bash
if [ -d "/var/www" ]; then
 echo "Directory exists"
else
 echo "Directory does not exist"
fi
```

Case Statements

```
#!/bin/bash
read -p "Enter a number: " num
case $num in
 1) echo "You chose one.";;
 2) echo "You chose two.";;
 *) echo "Invalid choice";;
esac
```

2.5 Loops

For Loop

```
#!/bin/bash
for i in 1 2 3 4 5; do
 echo "Number: $i"
done
```

While Loop

```
#!/bin/bash
count=1
while [ $count -le 5 ]; do
 echo "Count: $count"
 ((count++))
done
```

2.6 Functions

Functions allow you to reuse code blocks within your script.

Defining and Calling Functions

```
#!/bin/bash
function greet() {
 echo "Hello, $1"
}

greet "DevOps Engineer"
```

3. Environment Variables and Configuration Files

Bash scripts often interact with environment variables, such as `$HOME`, `$PATH`, or custom environment variables. These variables can be exported and accessed by other scripts or subshells.

```
#!/bin/bash
export DEV_ENV="production"
echo "Running in $DEV_ENV mode"
```

Additionally, bash scripts can source configuration files using the `source` or `.` command.

```
#!/bin/bash
source /etc/myapp/config
```

4. Automation and Task Scheduling

4.1 Cron Jobs

Cron is a Linux utility for scheduling tasks. You can use cron jobs to automate script execution.

Creating a Cron Job

```
crontab -e
```

Add a cron job to run a script every day at midnight:

```
0 0 * * * /path/to/script.sh
```

4.2 Automating Deployment

In DevOps, automation is key to efficient CI/CD pipelines. Bash scripts can help deploy services and manage environments.

Example: Automated Deployment

```
#!/bin/bash
# Pull the latest code from Git
git pull origin main

# Build the application
docker build -t myapp .

# Deploy to Kubernetes
kubectl apply -f deployment.yaml
```

5. Best Practices

- **Use `#!/bin/bash` shebang**: Always include the shebang at the start of the script.
- **Make scripts executable**: Use `chmod +x script.sh`.
- **Use functions to organize code**: Functions make your scripts modular and easier to maintain.
- **Error handling**: Use `set -e` to stop the script when a command fails.
- **Log outputs**: Keep logs of your script's activities by redirecting output and errors.
- **Document your code**: Use comments to explain non-obvious logic.

6. Common DevOps Use Cases

1. **System Monitoring and Alerts**: Create a script to check disk space and send alerts.

```
#!/bin/bash
THRESHOLD=80
df -h | grep 'sda1' | awk '{ print $5 }' | sed 's/%//' | while read
output;
do
 if [ $output -ge $THRESHOLD ]; then
   echo "Disk usage exceeded threshold" | mail -s "Alert: Disk Space"
admin@domain.com
 fi
done
```

2. **Service Restarts on Failure**: Automate the restart of a service if it fails.

```
#!/bin/bash
service nginx status
```

```
if [ $? -ne 0 ]; then
 echo "Restarting Nginx"
 service nginx restart
fi
```

3. **Backup Automation**: Backup files to S3 using bash and AWS CLI.

```
#!/bin/bash
DATE=$(date +%Y-%m-%d)
tar -czf /backup/$DATE-backup.tar.gz /var/www
aws s3 cp /backup/$DATE-backup.tar.gz s3://my-bucket/backups/
```

7. Conclusion

Bash scripting is a crucial skill for DevOps engineers, allowing for automation, configuration management, and orchestration across systems. Mastering the basics covered in this document will allow you to handle many essential DevOps tasks, and with practice, you'll be able to build more complex solutions.

Follow me on LinkedIn 😊