Lab 3: Shell Scripting, Git Basics & Application Concepts

Lab Overview

This lab introduces you to Bash shell scripting for task automation, Git version control for code management, and fundamental application architecture concepts. You will learn to write scripts that automate repetitive tasks, use Git for tracking code changes locally, and understand how modern applications are structured with frontend and backend components.

Topics Covered:

- Bash scripting fundamentals
- Variables, conditionals, and loops in shell scripts
- Git installation and configuration
- Local Git workflows (init, add, commit, log)
- Introduction to frontend/backend architecture
- Understanding client-server model

Learning Objectives

By the end of this lab, you will be able to:

1. Write Bash scripts to automate common system administration tasks
2. Use variables, conditionals, and loops in shell scripts
3. Initialize Git repositories and manage version history
4. Perform basic Git operations (add, commit, status, log)
5. Explain the difference between frontend and backend components
6. Understand how applications communicate in a client-server model

Part 1: Introduction to Shell Scripting

1.1 What is Shell Scripting?

A shell script is a text file containing a sequence of commands that the shell can execute. Shell scripts allow you to automate repetitive tasks, combine multiple commands, and create custom tools for system administration.

Benefits of Shell Scripting:

- Automates repetitive tasks
- Reduces human error
- Saves time on routine operations
- Creates reusable utilities
- Improves system administration efficiency

1.2 Creating Your First Script

Shell scripts typically start with a shebang line that tells the system which interpreter to use.

Basic Script Structure:

bash

```
#!/bin/bash
# This is a comment
echo "Hello, DevOps!"
```

Creating and Running a Script:

bash

```
# Create a new script file
nano hello.sh

# Make it executable
chmod +x hello.sh

# Run the script
./hello.sh
```

The `#!/bin/bash` line is called a shebang. It tells the system to use the Bash interpreter to execute the script.

1.3 Variables in Bash

Variables store data that can be used throughout your script.

Defining Variables:

bash

```
#!/bin/bash

# Variable assignment (no spaces around =)
name="DevOps Student"
age=20
course="SE202"
```

```
# Using variables (use $ to access value)
echo "Name: $name"
echo "Age: $age"
echo "Course: $course"
```

Important Rules:

- No spaces around the = sign
- Variable names are case-sensitive
- Use $ to access variable values
- Quote strings with spaces

Reading User Input:

bash

```
#!/bin/bash

echo "Enter your name:"
read username
echo "Hello, $username!"
```

Command Substitution:

Store command output in variables:

bash

```
#!/bin/bash

current_date=$(date)
current_user=$(whoami)
current_dir=$(pwd)

echo "Date: $current_date"
echo "User: $current_user"
echo "Directory: $current_dir"
```

1.4 Conditional Statements

Conditionals allow scripts to make decisions.

If Statement Syntax:

bash

```bash
#!/bin/bash

if [ condition ]; then
   # commands
fi
```

If-Else Example:

bash

```bash
#!/bin/bash

echo "Enter a number:"
read number

if [ $number -gt 10 ]; then
   echo "Number is greater than 10"
else
   echo "Number is 10 or less"
fi
```

Comparison Operators:

For numbers:

- `-eq` : equal to
- `-ne` : not equal to
- `-gt` : greater than
- `-lt` : less than
- `-ge` : greater than or equal to
- `-le` : less than or equal to

For strings:

- `=` : equal to
- `!=` : not equal to
- `-z` : string is empty
- `-n` : string is not empty

Multiple Conditions:

bash

```bash
#!/bin/bash

echo "Enter your age:"
read age

if [ $age -lt 18 ]; then
   echo "You are a minor"
elif [ $age -ge 18 ] && [ $age -lt 65 ]; then
   echo "You are an adult"
else
   echo "You are a senior citizen"
fi
```

File Test Operators:

bash

```bash
#!/bin/bash

file="test.txt"

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

Common file tests:

- `-f` : file exists and is a regular file
- `-d` : directory exists
- `-r` : file is readable
- `-w` : file is writable
- `-x` : file is executable

1.5 Loops in Bash

Loops allow you to repeat commands multiple times.

For Loop:

bash

```bash
#!/bin/bash

# Loop through a list
for name in Alice Bob Charlie
do
   echo "Hello, $name"
done

# Loop through numbers
for i in {1..5}
do
   echo "Number: $i"
done

# Loop through files
for file in *.txt
do
   echo "Found file: $file"
done
```

While Loop:

```bash
#!/bin/bash

counter=1

while [ $counter -le 5 ]
do
   echo "Counter: $counter"
   counter=$((counter + 1))
done
```

Reading Files Line by Line:

```bash
#!/bin/bash
```

```
while read line
do
    echo "Line: $line"
done < input.txt
```

1.6 Functions in Bash

Functions help organize code into reusable blocks.

bash

```
#!/bin/bash

# Define a function
greet() {
    echo "Hello, $1!"
}

# Call the function
greet "DevOps Student"
greet "World"

# Function with return value
add_numbers() {
    local result=$(($1 + $2))
    echo $result
}

sum=$(add_numbers 5 3)
echo "Sum: $sum"
```

1.7 Script Arguments

Scripts can accept command-line arguments.

bash

```
#!/bin/bash

echo "Script name: $0"
echo "First argument: $1"
```

```
echo "Second argument: $2"
echo "All arguments: $@"
echo "Number of arguments: $#"
```

Run with: `./script.sh arg1 arg2`

Part 2: Git Basics

2.1 What is Git?

Git is a distributed version control system that tracks changes in source code during software development. It allows multiple developers to work on the same project, maintains a complete history of changes, and helps manage different versions of code.

Why Use Git?

- Track changes over time
- Collaborate with team members
- Revert to previous versions
- Create branches for features
- Industry standard tool

2.2 Installing and Configuring Git

Installation:

bash

```
# Ubuntu/Debian
sudo apt update
sudo apt install git

# Check installation
git --version
```

Initial Configuration:

bash

```
# Set your name
git config --global user.name "Your Name"

# Set your email
git config --global user.email "your.email@example.com"
```

```
# Set default branch name
git config --global init.defaultBranch main

# View configuration
git config --list
```

2.3 Creating a Git Repository

Method 1: Initialize a New Repository

bash

```
# Create a project directory
mkdir my-first-repo
cd my-first-repo

# Initialize Git repository
git init

# Check status
git status
```

Understanding the .git Directory:

When you run `git init`, Git creates a hidden `.git` directory that stores all version control information. Never delete this directory manually.

2.4 Basic Git Workflow

The basic Git workflow consists of three main areas:

1. **Working Directory**: Where you modify files
2. **Staging Area (Index)**: Where you prepare changes for commit
3. **Repository**: Where Git permanently stores committed changes

The Workflow:

bash

```
# 1. Create or modify files
echo "# My Project" > README.md
```

```
# 2. Check status
git status

# 3. Add files to staging area
git add README.md

# Or add all changes
git add .

# 4. Check status again
git status

# 5. Commit changes
git commit -m "Initial commit with README"

# 6. View commit history
git log
```

2.5 Git Status and Tracking

Understanding File States:

Files in a Git repository can be in one of four states:

1. **Untracked**: Git doesn't know about the file
2. **Unmodified**: File is tracked but hasn't changed
3. **Modified**: File has been changed but not staged
4. **Staged**: File is ready to be committed

bash

```
# Check detailed status
git status

# Short status format
git status -s
```

2.6 Staging Changes

The staging area allows you to selectively choose which changes to commit.

bash

```
# Stage a specific file
git add filename.txt

# Stage multiple files
git add file1.txt file2.txt

# Stage all changes in current directory
git add .

# Stage all txt files
git add *.txt

# Remove file from staging area (unstage)
git reset filename.txt
```

2.7 Making Commits

Commits are snapshots of your project at a specific point in time.

Good Commit Messages:

- Present tense: "Add feature" not "Added feature"
- Concise but descriptive
- Explain what and why, not how

bash

```
# Commit staged changes
git commit -m "Add user authentication feature"

# Commit with detailed message
git commit -m "Add login functionality" -m "Users can now log in with email and password"

# Stage and commit in one step (tracked files only)
git commit -am "Update documentation"
```

2.8 Viewing History

bash

```
# View commit history
git log
```

```bash
# Compact log (one line per commit)
git log --oneline

# Show last n commits
git log -n 5

# Show commits with changes
git log -p

# Show commits by author
git log --author="Your Name"

# Graphical representation
git log --graph --oneline
```

2.9 Viewing Changes

```bash
# Show unstaged changes
git diff

# Show staged changes
git diff --staged

# Show changes in a specific file
git diff filename.txt

# Compare two commits
git diff commit1 commit2
```

2.10 Ignoring Files

Create a `.gitignore` file to tell Git which files to ignore:

```bash
# Create .gitignore
nano .gitignore
```

```
# Example .gitignore contents:
# Ignore log files
*.log

# Ignore temporary files
*.tmp
*~

# Ignore node_modules directory
node_modules/

# Ignore environment files
.env

# But don't ignore .gitignore itself
!.gitignore
```

Then add and commit it:

```
bash
git add .gitignore
git commit -m "Add gitignore file"
```

Part 3: Application Concepts

3.1 Understanding Applications

An application (or app) is a software program designed to perform specific tasks for users. In DevOps, we typically work with web applications, which are accessed through web browsers.

Types of Applications:

- **Desktop Applications**: Run locally on a computer (e.g., Microsoft Word)
- **Web Applications**: Accessed through browsers (e.g., Gmail, Facebook)
- **Mobile Applications**: Run on smartphones and tablets
- **Command-Line Applications**: Text-based interface (like the tools we've been using)

3.2 Client-Server Architecture

Most modern web applications follow a client-server model:

Client (Frontend):

- Runs in the user's web browser
- Handles user interface and user interactions
- Displays information to users
- Technologies: HTML, CSS, JavaScript

Server (Backend):

- Runs on a remote computer (server)
- Processes requests from clients
- Manages data and business logic
- Stores and retrieves data from databases
- Technologies: Python, Node.js, Java, Go

How They Communicate:

1. User interacts with frontend in browser
2. Frontend sends HTTP request to backend server
3. Backend processes the request
4. Backend queries database if needed
5. Backend sends HTTP response back to frontend
6. Frontend displays the result to user

3.3 Frontend vs Backend

Frontend (Client-Side):

- What users see and interact with
- Presentation and user experience
- Runs in the user's browser
- Examples: buttons, forms, menus, animations

Backend (Server-Side):

- What users don't see
- Business logic and data processing
- Runs on the server
- Examples: authentication, database operations, calculations

Example Scenario - Login Process:

Frontend (Browser):

1. User enters username and password
2. User clicks "Login" button
3. JavaScript sends credentials to server

4. Displays "Logged in successfully" or error message

Backend (Server):

1. Receives username and password
2. Checks credentials against database
3. Creates session if credentials are valid
4. Sends success or failure response

3.4 HTTP Basics

HTTP (Hypertext Transfer Protocol) is the protocol used for communication between frontend and backend.

HTTP Request Methods:

- **GET**: Retrieve data (e.g., load a webpage)
- **POST**: Send data to server (e.g., submit a form)
- **PUT**: Update existing data
- **DELETE**: Delete data

HTTP Status Codes:

- **200 OK**: Request succeeded
- **201 Created**: Resource created successfully
- **400 Bad Request**: Invalid request
- **401 Unauthorized**: Authentication required
- **404 Not Found**: Resource doesn't exist
- **500 Internal Server Error**: Server error

3.5 Introduction to APIs

API (Application Programming Interface) is a way for different software components to communicate.

REST API Example:

```
GET /api/users          - Get all users
GET /api/users/123      - Get user with ID 123
POST /api/users         - Create new user
PUT /api/users/123      - Update user 123
DELETE /api/users/123   - Delete user 123
```

Part 4: Practical Exercises

Exercise 1: Basic Shell Scripts

Task 1: Create a greeting script

Create `greet.sh`:

bash

```
#!/bin/bash

echo "Enter your name:"
read name
echo "Welcome to DevOps, $name!"
echo "Today's date is: $(date +%Y-%m-%d)"
```

Make it executable and run:

bash

```
chmod +x greet.sh
./greet.sh
```

Task 2: System information script

Create `sysinfo.sh`:

bash

```
#!/bin/bash

echo "=== System Information ==="
echo "Hostname: $(hostname)"
echo "Current User: $(whoami)"
echo "Current Directory: $(pwd)"
echo "Disk Usage:"
df -h | grep "^/dev"
echo ""
echo "Memory Usage:"
free -h
```

Task 3: File backup script

Create `backup.sh`:

bash

```bash
#!/bin/bash

# Check if argument is provided
if [ $# -eq 0 ]; then
    echo "Usage: $0 <directory_to_backup>"
    exit 1
fi

source_dir=$1
backup_name="backup_$(date +%Y%m%d_%H%M%S).tar.gz"

echo "Creating backup of $source_dir..."
tar -czf $backup_name $source_dir

if [ $? -eq 0 ]; then
    echo "Backup created successfully: $backup_name"
else
    echo "Backup failed!"
    exit 1
fi
```

Exercise 2: Conditional Logic

Task 1: File checker script

Create `filecheck.sh`:

bash

```bash
#!/bin/bash

echo "Enter filename:"
read filename

if [ -f "$filename" ]; then
    echo "File exists!"
    echo "Size: $(du -h "$filename" | cut -f1)"
    echo "Permissions: $(ls -l "$filename" | cut -d' ' -f1)"
```

```
elif [ -d "$filename" ]; then
    echo "This is a directory, not a file"
else
    echo "File does not exist"
fi
```

Task 2: Grade calculator

Create `grade.sh`:

bash

```
#!/bin/bash

echo "Enter your score (0-100):"
read score

if [ $score -ge 85 ]; then
    echo "Grade: A"
elif [ $score -ge 70 ]; then
    echo "Grade: B"
elif [ $score -ge 60 ]; then
    echo "Grade: C"
elif [ $score -ge 50 ]; then
    echo "Grade: D"
else
    echo "Grade: F"
fi
```

Exercise 3: Loops

Task 1: File processor

Create `process_files.sh`:

bash

```
#!/bin/bash

echo "Creating test files..."
for i in {1..5}
```

```
do
    echo "This is file number $i" > file$i.txt
    echo "Created file$i.txt"
done

echo ""
echo "Listing created files:"
ls -l file*.txt
```

Task 2: Directory size report

Create `dirsize.sh`:

bash

```
#!/bin/bash

echo "=== Directory Size Report ==="
for dir in */
do
    size=$(du -sh "$dir" 2>/dev/null | cut -f1)
    echo "$dir: $size"
done
```

Exercise 4: Git Practice

Task 1: Create a project with Git

bash

```
# Create project directory
mkdir git-practice
cd git-practice

# Initialize Git
git init

# Create README file
echo "# Git Practice Project" > README.md
echo "This project is for learning Git basics." >> README.md
```

```
# Check status
git status

# Add and commit
git add README.md
git commit -m "Initial commit: Add README"

# View history
git log
```

Task 2: Make multiple commits

bash

```
# Create a new file
echo "## Features" >> README.md
echo "- Feature 1" >> README.md

# Check what changed
git diff

# Commit the change
git add README.md
git commit -m "Add features section to README"

# Create a new file
echo "print('Hello, DevOps!')" > hello.py

# Add and commit
git add hello.py
git commit -m "Add Python hello script"

# View history
git log --oneline
```

Task 3: Practice ignoring files

bash

```
# Create some files
```

```
touch debug.log
touch test.tmp
mkdir temp_data

# Create .gitignore
cat > .gitignore << EOF
*.log
*.tmp
temp_data/
EOF

# Check status (ignored files won't appear)
git status

# Commit .gitignore
git add .gitignore
git commit -m "Add gitignore file"
```

Exercise 5: Combining Scripts and Git

Task: Create a monitored script project

bash

```
# Create project
mkdir system-monitor
cd system-monitor
git init

# Create monitoring script
cat > monitor.sh << 'EOF'
#!/bin/bash

echo "=== System Monitor ==="
echo "Date: $(date)"
echo "Uptime: $(uptime -p)"
echo "Disk Usage:"
df -h | grep "^/dev" | awk '{print $1 ": " $5}'
echo "Top 5 Processes by CPU:"
```

```
ps aux --sort=-%cpu | head -6 | tail -5
EOF

chmod +x monitor.sh

# Commit initial version
git add monitor.sh
git commit -m "Add initial system monitor script"

# Enhance the script
cat >> monitor.sh << 'EOF'

echo ""
echo "Memory Usage:"
free -h | grep "Mem:" | awk '{print "Used: " $3 " / Total: " $2}'
EOF

# Commit enhancement
git add monitor.sh
git commit -m "Add memory usage monitoring"

# View history
git log --oneline
```

Part 5: Practical Application Scenarios

Scenario 1: Automated Log Cleanup

Create `cleanup_logs.sh`:

bash

```
#!/bin/bash

log_dir="/var/log"
days_old=7

echo "=== Log Cleanup Script ==="
echo "Searching for logs older than $days_old days in $log_dir"
```

```
# Find and list old log files
old_logs=$(find $log_dir -name "*.log" -type f -mtime +$days_old 2>/dev/null)

if [ -z "$old_logs" ]; then
    echo "No old log files found"
else
    echo "Found old log files:"
    echo "$old_logs"

    echo ""
    echo "Do you want to delete these files? (yes/no)"
    read answer

    if [ "$answer" = "yes" ]; then
        find $log_dir -name "*.log" -type f -mtime +$days_old -delete 2>/dev/null
        echo "Old logs deleted"
    else
        echo "Deletion cancelled"
    fi
fi
```

Scenario 2: User Management Script

Create `user_manager.sh`:

bash

```
#!/bin/bash

show_menu() {
    echo "=== User Management ==="
    echo "1. List all users"
    echo "2. Check if user exists"
    echo "3. Show current user"
    echo "4. Exit"
    echo "Enter choice:"
}
```

```
list_users() {
    echo "System users:"
    cut -d: -f1 /etc/passwd | sort
}

check_user() {
    echo "Enter username to check:"
    read username
    if id "$username" &>/dev/null; then
        echo "User $username exists"
        id "$username"
    else
        echo "User $username does not exist"
    fi
}

while true
do
    show_menu
    read choice

    case $choice in
        1) list_users ;;
        2) check_user ;;
        3) echo "Current user: $(whoami)" ;;
        4) echo "Goodbye!"; exit 0 ;;
        *) echo "Invalid choice" ;;
    esac

    echo ""
done
```

Lab Assessment Tasks

Complete the following tasks and document your work:

Task 1: Personal Information Script

Create a script called `myinfo.sh` that:

1. Asks for your name, student ID, and favorite programming language
2. Displays a formatted summary
3. Saves the information to a file called `student_info.txt`
4. Displays the current date and time
5. Shows your current directory

Task 2: File Organizer Script

Create `organize.sh` that:

1. Creates directories: `documents`, `images`, `scripts`, `others`
2. Moves `.txt` and `.pdf` files to `documents/`
3. Moves `.jpg` and `.png` files to `images/`
4. Moves `.sh` files to `scripts/`
5. Moves everything else to `others/`
6. Displays a summary of moved files

Task 3: Git Project Setup

Create a new project called `devops-scripts`:

1. Initialize a Git repository
2. Create a README.md describing the project
3. Create at least 3 shell scripts from previous exercises
4. Create a .gitignore file to ignore `.log` and `.tmp` files
5. Make at least 5 meaningful commits
6. Use `git log --oneline --graph` to display history
7. Save the output to `git-history.txt`

Task 4: System Health Check Script

Create `health_check.sh` that:

1. Checks if disk usage is above 80% (warn if true)
2. Checks if memory usage is above 80% (warn if true)
3. Lists top 5 CPU-consuming processes
4. Checks if system uptime is less than 1 day (warn if true)
5. Saves the report to `health_report_$(date +%Y%m%d).txt`
6. Uses conditional statements and functions

Task 5: Git Workflow Documentation

Create a document called `git-workflow.md` that:

1. Explains the purpose of Git
2. Lists and explains 10 Git commands you learned
3. Describes the three Git areas (working, staging, repository)
4. Provides examples of good commit messages
5. Explains when to use .gitignore

Submit all scripts with proper permissions and ensure they run without errors.