Bash Scripting

08 June 2025 21:29

Bash Scripting Session

1. Welcome & Motivation

- **Audience:** Especially valuable for those new to scripting or with little programming experience.
- **Encouragement:**
    - Don't doubt yourself—follow along and type every script shown.
    - Scripts are provided in resources, but hand-writing them reinforces learning.
    - Compare your scripts against provided versions to troubleshoot errors.

2. Troubleshooting Tip

- **Common Cause of Failures:** Typographical mistakes (misspellings, missing spaces, stray characters).
- **Debugging Strategy:**
    1. Carefully compare your script line-by-line with the reference.
    2. Focus on syntax and spacing.
    3. Practice repeatedly—errors become easier to spot with experience.

3. What Is Bash Scripting?

1. **Repetitive Administrative Tasks:**
    - System health checks, patching, backups, user-management, log rotations, etc.
    - Tasks often run daily or weekly.
2. **Automation via Scripts:**
    - Compile commands into a plain-text file.
    - Execute the file to perform all tasks automatically.
3. **Definition of Bash:**
    - **Bash Shell:** "Bourne Again SHell"—default interactive shell on many Linux distributions.
    - **Bash Scripting:** Writing scripts specifically for the bash shell.
    - Other shells exist (sh, ksh, zsh), but "bash scripting" refers exclusively to bash.

4. Role of Shell Scripting in System Administration

- **"Robotic Tasks":** Any task performed repeatedly by a sysadmin can be scripted.
- **Efficiency Gains:**
    - Saves time and reduces manual errors.
    - Ensures consistency across executions.
- **Script vs. Manual Execution:**
    - Manual: Typing each command each time.
    - Scripted: Writing commands once in a .sh file, then running it.

5. Relationship to Modern Automation Tools

- **Popular Tools:** Ansible, Puppet, Chef, SaltStack, Terraform.
- **Why Learn Bash First?**
    - Many automation tools' concepts derive from shell scripting.
    - Strong bash foundation eases the learning curve for configuration-management frameworks.

    - Empowers you to customize or extend built-in modules when needed.

6. Purpose of This Course's Bash Module

- **Objectives:**
    - Build confidence in scripting.
    - Understand core scripting concepts before diving into higher-level tools.
    - Develop hands-on skills in writing, debugging, and executing bash scripts.
- **Outcome:**
    - Gain the ability to automate day-to-day sysadmin tasks.
    - Establish a solid scripting base for advanced DevOps practices.

7. Next Steps

- **In Next Lecture:**
    - Introduction to actual bash script structure and syntax.
    - Hands-on examples demonstrating variables, control flow, functions, and more.
- **Action Item:**
    - Ensure your environment is ready (Linux shell access).
    - Review provided resource scripts.
    - Prepare to write and execute your first bash scripts.

Vagrant-Powered Bash Scripting Lab

1. Objective

- **Goal:** Create three CentOS **7 virtual machines via Vagrant for practicing Bash scripts.**
- **VMs Defined in Vagrantfile:**
    - scriptbox
    - web01
    - web02
- **Primary Workstation:** scriptbox — all scripts are authored and tested here, then (later) pushed to web01/web02.

2. Preparing the Environment

1. **Create a Working Directory** (e.g., D:/…/bash scripts dir).
2. **Copy the Vagrantfile** from your Downloads folder to this new directory:
   ```
   cp ~/Downloads/Vagrantfile /d/.../bash_scripts_dir/
   ```
3. **Enter the Directory**:
   ```
   cd /d/.../bash_scripts_dir/
   ```
4. **Open the Vagrantfile** for review/editing (using vim, Notepad++, etc.):
   ```
   vim Vagrantfile
   ```
    - Confirm it defines three CentOS 7 boxes with unique IPs.

3. Bringing Up the scriptbox VM

1. **Boot Only scriptbox** (avoiding all three VMs):

   ```
   vagrant up scriptbox
   ```
   - This will import the CentOS 7 box, configure networking, forward SSH (guest 22 → host 2222), and set up shared folders.
2. **Verify VM Is Ready**
   - Look for messages:
     - "Machine booted and ready!"
     - "Setting hostname…"
     - "Rsyncing folder: … → /vagrant"

4. Connecting to scriptbox

1. **SSH into the VM**:

   ```
   vagrant ssh scriptbox
   ```
   - Initial MOTD indicates EuroLinux box.
   - If prompted, simply re-run vagrant ssh scriptbox.
2. **Elevate to Root**:

   ```
   sudo -i
   ```
   - Switches you to the root user for script editing and system changes.

5. Setting the Hostname

1. **Edit /etc/hostname**:

   ```
   vi /etc/hostname
   ```
   - Replace existing name with scriptbox.
2. **Apply the Hostname** (immediate effect):

   ```
   hostname scriptbox
   ```

3. **Verify**:
   - Run hostname or observe prompt change to root@scriptbox.
4. **Log Out & Back In** to ensure the prompt persists:

   ```
   logout
   vagrant ssh scriptbox
   sudo -i
   ```
   - Confirm you see root@scriptbox in the shell prompt.

6. Directory & File Synchronization

- **Shared Folder**: Your host directory is synced to /vagrant inside scriptbox.
- **Usage**: Place or edit your Bash scripts in the host folder; they'll be available in /vagrant on the VM.

7. Next Steps

- **Upcoming Lecture:** Begin writing and running Bash scripts on scriptbox.
- **Future Work:** Test scripts locally on scriptbox, then push and execute them on web01 and web02.

  Tip: Always work as root (sudo -i) when creating system-level scripts to avoid permission issues.

Bash Scripting: Writing Your First Script

1. Setup Directory & Editor

- **Create scripts directory**
  ```
  mkdir -p /opt/scripts
  cd /opt/scripts
  ```
- **Install Vim** (if not already present)
  ```
  yum install -y vim
  ```
- **Use Vim** (or any text editor) to write scripts on the CentOS box.

2. Creating the Script File

- **Filename convention:** .sh extension (e.g., firstscript.sh)
- **Open file in Vim:**
  ```
  vim firstscript.sh
  ```

3. Shebang & Basic Commands

1. **Shebang (Line 1):**
   ```
   #!/bin/bash
   ```
   - Instructs the system to use the Bash interpreter.
2. **Print a Welcome Message:**
   ```
   echo "Welcome to bash script."
   echo
   ```
3. **System Uptime:**
   ```
   echo "The uptime of the system is:"
   uptime
   echo
   ```
4. **Memory Utilization:**
   ```
   echo "Memory Utilization"
   free -m
   echo
   ```
5. **Disk Utilization:**
   ```
   echo "Disk Utilization:"
   df -h
   ```

4. Saving & Executing the Script

1. **Save & Quit Vim:** :wq
2. **Attempt to Run (relative path):**
   ```
   ./firstscript.sh
   ```
   - **Error:** Permission denied (no execute bit).
3. **Make Executable:**
   ```
   chmod +x firstscript.sh
   ```
4. **Run Again:**
   ```
   ./firstscript.sh
   ```

5. Improving Readability

1. **Add Spacing:** Insert blank echo lines between sections.
2. **Enable Line Numbers (in Vim):**
   ```
   :se nu
   ```
3. **Insert Comment Blocks:**
   ```
   ### This script prints system info ###
   # Checking system uptime
   # Memory Utilization
   # Disk Utilization
   ```
   - Comments start with # and are ignored by the interpreter.

6. Absolute vs. Relative Path Execution

- **Relative Path:**
  ```
  ./firstscript.sh
  ```
- **Absolute Path:**
  ```
  /opt/scripts/firstscript.sh
  ```

7. Key Takeaways

- **Script = Text File** of Bash commands.
- **Shebang** tells which interpreter to use.
- **Executable Permission** is required (chmod +x).
- **Comments (#)** and **spacing** make scripts and output more readable.
- **Run scripts** via relative or absolute path.

Automating Website Setup with a Bash Script

1. Use Case & Script Goal

Rather than manually running setup commands each time, we'll record them in a single script—**websetup.sh**—that:

1. Installs required packages (wget, unzip, httpd)
2. Starts and enables Apache HTTPD
3. Downloads and deploys a website template
4. Cleans up temporary files
5. Shows service status and deployed files

2. Initial Command Sequence

Executed manually, the steps are:

```
#!/bin/bash
sudo yum install wget unzip httpd -y
sudo systemctl start httpd
sudo systemctl enable httpd
mkdir -p /tmp/webfiles
cd /tmp/webfiles
wget https://www.tooplate.com/zip-templates/2098_health.zip
unzip 2098_health.zip
sudo cp -r 2098_health/* /var/www/html/
```

```
systemctl restart httpd
rm -rf /tmp/webfiles
```

3. Enhancing Readability & User Feedback

1. **Add Shebang** (line **1**):
   ```
   #!/bin/bash
   ```
2. **Section Headers & Echo Statements**—print clear messages before each major step.
3. **Comments** (#) to document intent.
4. **Suppress Unnecessary Output**—redirect successful command output to /dev/null but allow errors to display.

4. Final Script: websetup.sh

```
#!/bin/bash
# Installing Dependencies
echo "########################################"
echo "Installing packages."
echo "########################################"
sudo yum install wget unzip httpd -y > /dev/null
echo
# Start & Enable Service
echo "########################################"
echo "Start & Enable HTTPD Service"
echo "########################################"
sudo systemctl start httpd
sudo systemctl enable httpd
echo
# Creating Temp Directory & Changing to It
echo "########################################"
echo "Starting Artifact Deployment"
echo "########################################"
mkdir -p /tmp/webfiles
cd /tmp/webfiles
echo
# Download & Unzip Website Template
echo "########################################"
echo "Downloading and Unzipping Web Template"
echo "########################################"
wget https://www.tooplate.com/zip-templates/2098_health.zip > /dev/null
unzip 2098_health.zip > /dev/null
echo
# Deploy to Apache Document Root
echo "##########################################"
echo "Copying Website Files to /var/www/html/"
echo "##########################################"
sudo cp -r 2098_health/* /var/www/html/
echo
# Restart HTTPD Service
echo "##########################################"
echo "Restarting HTTPD service"
echo "##########################################"
sudo systemctl restart httpd
echo
```

```
# Clean Up Temporary Files
echo "########################################"
echo "Removing Temporary Files"
echo "########################################"
rm -rf /tmp/webfiles
echo
# Show Final Status & Deployed Files
echo "########################################"
echo "Final HTTPD Status & Deployed Files"
echo "########################################"
sudo systemctl status httpd
ls /var/www/html/
```

5. Testing the Script

1. **Make Executable**
   ```
   chmod +x websetup.sh
   ```
2. **Run via Absolute Path**
   ```
   /opt/scripts/websetup.sh
   ```
3. **Verify in Browser**
   - Obtain VM IP via ifconfig
   - Navigate to http://<VM_IP>/—the Tooplate "Health Center" site should display.
   - http://192.168.10.12/

6. Key Improvements

- **Readability for Maintainers:** Clear section headers, comments, and echo messages.
- **User-Friendly Output:** Only critical messages and errors appear on-screen.
- **Automation:** Single command execution replaces multiple manual steps.

  Next Up: Learn how to make your scripts more flexible using **variables** and **parameterization**.

Leveraging ChatGPT for Bash Scripting

1. The Role of ChatGPT vs. Scripting Knowledge

- **Key Point:** ChatGPT can generate scripts, but only if **you** understand scripting.
- **Without scripting skills:** You'll struggle to use or refine AI-generated code.
- **With scripting skills:** ChatGPT becomes a powerful assistant to write, enhance, and debug scripts.

2. Example Prompt & Generated Script

- **Prompt Given:**
  "Bash Script to install httpd package, start httpd service, download HTML template from tooplate.com and deploy to /var/www/html. At the end, restart the httpd service and check the status of httpd service."
- **AI Response Highlights:**
  1. Installs httpd
  2. Starts service
  3. Downloads and unzips template directly into /var/www/html
  4. Restarts and checks service status

3. Testing & Refinement

- **Issue Identified:** Template ZIP contains a subdirectory; extracting blindly creates an extra nested folder.
- **Action:**
    1. **Test** the generated script in your environment.
    2. **Observe** unexpected behavior (nested folder).
    3. **Modify** the script to unpack correctly (e.g., extract from inside subfolder).

4. Enhancements via ChatGPT

- **Ask for Improvements:**
  "Enhance my script" + paste your current script.
- **Typical AI Enhancements:**
    - **Readability:** clearer echo messages, better formatting
    - **Error Handling:** set -e, checks after critical commands
    - **Use of Variables:** e.g. TMP_DIR=/tmp/webfiles reused throughout
    - **Comments & Structure:** section headers, inline documentation

5. Best Practices for AI-Assisted Scripting

1. **Know the Basics First:**
    - Understand shebangs, permissions, echo, conditionals, loops.
2. **Iterative Development:**
    - Generate a draft with ChatGPT → test locally → refine prompt or code → repeat.
3. **Customize to Your Environment:**
    - Adjust paths, service names, download URLs, and directory structures.
4. **Incorporate Advanced Logic:**
    - Later lectures cover OS detection, looping across multiple servers, user input, etc.

6. Next Steps

- **Hands-On:** Chat with ChatGPT to generate variants:
    - Add logging
    - Parameterize versions or URLs
    - Introduce conditional checks (e.g., if service already running)
- **Upcoming Lecture:** Learn about **variables**, **conditionals**, and more complex scripting constructs—now with AI assistance as a guide.

  Tip: Always review and **test** AI-generated scripts line-by-line before running them in production!

Bash Scripting

1. Definition of Variables

- **Variables** are temporary storage locations in a process's memory (RAM).
- When a process terminates, its variables (and all data in RAM) are lost—unlike data on disk.
- In Bash, variables live within the shell process and provide a way to reuse and parameterize values.

2. Declaring & Using Variables

1. **Declaration Syntax**

   ```
   VARIABLE_NAME=value
   ```

   - **No spaces** around =.
   - Variable names are conventionally **uppercase** (e.g., SKILL, PACKAGE).
2. **Retrieval (Interpolation)**

   ```
   echo $VARIABLE_NAME
   ```

   - The **$** prefix tells Bash to substitute with the variable's value.
   - Without $, Bash treats the name as literal text.

3. Simple Examples

1. **Storing a Text Value**

   ```
   SKILL=DevOps
   echo $SKILL       # Outputs: DevOps
   echo SKILL        # Outputs: SKILL
   ```
2. **Storing Multiple Words**

   ```
   PACKAGE="httpd wget unzip"
   echo yum install $PACKAGE -y
   # Executes: yum install httpd wget unzip -y
   ```

4. Benefits of Variables in Scripts

- **Maintainability:** Change a value in one place (at the top) instead of many.
- **Reusability:** Use the same variable in multiple commands (e.g., service name).
- **Flexibility:** Easily adapt scripts for different environments by modifying variables only.

5. Applying Variables to the Website Setup Script

1. **Renaming for Clarity**
   - Original script: 1_firstscript.sh
   - Enhanced with variables: 3_vars_website.sh
2. **Variable Declarations (Top of Script)**

   ```
   #!/bin/bash

   PACKAGE="httpd wget unzip"
   SVC="httpd"
   URL="https://www.tooplate.com/zip-templates/2098_health.zip"
   ARTIFACT="2098_health"
   TMP_DIR="/tmp/webfiles"
   ```
3. **Using Variables Throughout**

   ```
   # Install dependencies
   sudo yum install $PACKAGE -y > /dev/null

   # Manage service
   sudo systemctl start $SVC
   sudo systemctl enable $SVC

   # Prepare temp directory
   mkdir -p $TMP_DIR && cd $TMP_DIR
   ```

```
# Download & unzip
wget $URL -q
unzip ${ARTIFACT}.zip > /dev/null

# Deploy
sudo cp -r ${ARTIFACT}/* /var/www/html/

# Restart service
sudo systemctl restart $SVC

# Clean up
rm -rf $TMP_DIR

# Final checks
sudo systemctl status $SVC
ls /var/www/html/
```

4. **Smart Variable Usage**
    - **${ARTIFACT}.zip** vs. **${ARTIFACT}/***: same base name used for both download and deployment.
    - **$SVC** reused for start, enable, restart, and status commands.

6. Verifying and Testing

1. **Dismantle Environment**
    - Create a small teardown script to stop the service, remove website files, and uninstall packages.
    - Ensures a clean state before testing the new variable-driven script.
2. **Execute Enhanced Script**
    ```
    chmod +x 3_vars_website.sh
    ./3_vars_website.sh
    ```
3. **Validation**
    - Check in-browser that the website template is correctly deployed.
    - Confirm service is **active (running)** and files exist in /var/www/html/.

Next Up: Explore **conditional logic** and **loops** in Bash to further enhance script flexibility and robustness.

Bash Scripting: Command-Line Arguments

1. Introduction to Command-Line Arguments

- Many shell commands accept arguments (e.g., ls /path, cp src dest).
- **Goal:** Learn how to make our scripts accept and use arguments provided on the command line.
- Use case: Pass the URL of a website template and its artifact name into our deployment script, making it reusable for any template.

2. Positional Parameters in Bash

- **$0 – Name or path of the script itself.**
- **$1** to **$9 – First through ninth arguments passed to the script.**
- If an argument isn’t provided, its corresponding variable is empty.

3. Demonstration Script: 4_args.sh

```
#!/bin/bash
echo "Value of zero is: $0"
echo "Value of one   is: $1"
echo "Value of two   is: $2"
echo "Value of three is: $3"
```

1. **Make Executable:**
   ```
   chmod +x 4_args.sh
   ```
2. **Run Without Arguments:**
   ```
   ./4_args.sh
   # Outputs:
   # Value of zero is: ./4_args.sh
   # Value of one   is:
   # Value of two   is:
   # Value of three is:
   ```
3. **Run With Arguments:**
   ```
   ./4_args.sh Linux Bash Scripting
   # Outputs:
   # Value of zero is: ./4_args.sh
   # Value of one   is: Linux
   # Value of two   is: Bash
   # Value of three is: Scripting
   ```

4. Integrating Arguments into the Website Setup Script

1. **Copy Existing Script:**
   ```
   cp 3_vars_websetup.sh 5_args_websetup.sh
   ```
2. **Remove Hard-Coded Variables:**
   - Comment out or delete declarations of URL and ARTIFACT.
3. **Use $1 and $2:**
   - Replace all occurrences of the URL variable with $1.
   - Replace all occurrences of the artifact name with $2.
     ```
     wget $1 -q
     unzip ${2}.zip > /dev/null
     sudo cp -r ${2}/* /var/www/html/
     ```
4. **Execution Requirement:**
   - The user **must** supply two arguments:
     1. **$1 – Download URL of the website template.**
     2. **$2 – Artifact (folder) name inside the ZIP.**

5. Hands-On Example with a New Template

1. **Choose a Different Template** on **tooplate.com (e.g., “Zigi Health”).**
2. **Copy Download URL** via browser DevTools (Network tab).
3. **Clean Up Previous Deployment.**
4. **Run Script with Arguments:**
   ```
   ./5_args_websetup.sh \
   ```

```
https://www.tooplate.com/zip-templates/2099_zigi.zip \
2099_zigi
```

5. **Verify:**
   - Check service status.
   - Browse to http://<VM_IP>/ and see “Welcome to Zigi”.

6. Key Takeaways

- **$0 – Script name/path.**
- **$1–$9 – Positional arguments.**
- Arguments enable **reusable**, **flexible** scripts.
- Always **validate** or **check** for required arguments in production scripts (e.g., exit if $# -lt 2).

Bash Scripting: Special System Variables

1. Review of Positional Parameters

- **$0 – Name or path of the script.**
- **$1…$9 – Command**-line arguments 1 through 9.

2. Special System Variables

Variable	Meaning
$#	Number of arguments passed to the script
$@	All arguments as separate words (array of "$1" "$2"…)
$?	Exit status of the last command
$USER	Username of the current user
$HOSTNAME	Hostname of the machine
$RANDOM	A random integer between 0 and 32767

3. Understanding Exit Status ($?)

1. **Successful Command**
   ```
   free -m
   echo $?    # Outputs: 0
   ```
   - 0 indicates **success**.
2. **Failed Command**
   ```
   freeee -m
   # bash: freeee: command not found
   echo $?    # Outputs: 127
   ```
   - Non-zero indicates **error**; 127 for “command not found.”
3. **Another Failure Example**
   ```
   ls nonexistentfile
   # ls: cannot access 'nonexistentfile': No such file or directory
   echo $?    # Outputs: 1
   ```

  - 1 commonly means a general error.

4. Examples of Other System Variables

1. **All Arguments ($@)**
```
./script.sh one two three
echo "$@"    # Outputs: one two three
```
2. **Argument Count ($#)**
```
./script.sh a b c d
echo $#      # Outputs: 4
```
3. **Current User ($USER)**
```
echo $USER   # e.g., outputs: vagrant or your username
```
4. **Hostname ($HOSTNAME)**
```
echo $HOSTNAME   # e.g., outputs: scriptbox
```
5. **Random Number ($RANDOM)**
```
echo $RANDOM    # e.g., outputs: 23901 (changes each run)
```

5. Usage in Scripts

- **Error Handling:**
```
some_command
if [ $? -ne 0 ]; then
  echo "Error: some_command failed!"
  exit 1
fi
```
- **Looping Over All Args:**
```
for arg in "$@"; do
  echo "Processing: $arg"
done
```

  Practice: Execute and echo each of these special variables in your shell to observe their behavior. These will be used extensively in upcoming scripts.

Bash Scripting: Quotes

1. Types of Quotes in Bash

- **Double Quotes** ("…"):
  - Allow **variable interpolation** and **interpretation** of special characters (e.g., $, `, \).
- **Single Quotes** ('…'):
  - Treat everything **literally**, disabling interpolation and special-character processing.

2. Variable Assignment & Printing

1. **Assign with Double Quotes**
```
SKILL="DevOps"
echo $SKILL       # Outputs: DevOps
```
2. **Reassign with Single Quotes**
```
SKILL='Containerization'
echo $SKILL       # Outputs: Containerization
```

    - **No difference** in assignment; both quotes store literal text.

3. **Interpolation in Double Quotes**
   ```
   echo "My skill is $SKILL"
   # Outputs: My skill is Containerization
   ```
4. **Literal in Single Quotes**
   ```
   echo 'My skill is $SKILL'
   # Outputs: My skill is $SKILL
   ```

3. Special Characters & Quotes

- **Single Quotes**:
    - **Disable** all special meanings (e.g., $, `, \).
    - Use when you want text exactly as written.
- **Double Quotes**:
    - **Enable** variable expansion and certain escapes.
    - Use when you need to embed variables or command substitutions.

4. Printing Mixed Literal & Variable Text

Scenario: You need to print both a literal dollar sign and a variable:

“Due to $VIRUS virus, company have lost 9 million dollars.”

1. **Risk with Double Quotes Only**
   ```
   VIRUS="WannaCry"
   echo "Due to $VIRUS virus, company have lost $9 million dollars"
   # Outputs: Due to WannaCry virus, company have lost  million dollars
   # ($9 is interpreted as the 9th script argument, empty here)
   ```
2. **Risk with Single Quotes Only**
   ```
   echo 'Due to $VIRUS virus, company have lost $9 million dollars'
   # Outputs literally:
   # Due to $VIRUS virus, company have lost $9 million dollars
   # (No variable expansion)
   ```
3. **Solution: Escape the Dollar Sign**
   ```
   echo "Due to $VIRUS virus, company have lost \$9 million dollars"
   # Outputs: Due to WannaCry virus, company have lost $9 million dollars
   ```
    - **\"** keeps double-quote context (allowing $VIRUS),
    - **\$** prints a literal $ by escaping its special meaning.

5. Key Takeaways

- Choose **double quotes** when you need **interpolation**.
- Choose **single quotes** to treat text **literally**.
- **Escape** special characters (e.g., \$, \", \`) within double quotes to print them literally.

Bash Scripting: Command Substitution

1. What Is Command Substitution?

- **Purpose:** Capture the **output** of a shell command and store it in a variable.

- **Syntax Options:**
  1. **Backticks**: <code>`command`</code>
  2. **Dollar-Parentheses**: <code>$(command)</code>

2. Basic Examples

1. **Storing uptime Output**
   ```
   # Using backticks
   up=`uptime`
   echo "Uptime: $up"

   # Using $()
   up=$(uptime)
   echo "Uptime: $up"
   ```
2. **Storing who Output**
   ```
   users=$(who)
   echo "Current users logged in:"
   echo "$users"
   ```

3. Filtering & Storing Specific Fields

1. **Extract Free RAM (in MB)**

   # Pipeline: free -m → grep Mem → awk to print 4th field
   ```
   FREE_RAM=`free -m | grep Mem | awk '{print $4}'`
   echo "Free RAM: ${FREE_RAM}MB"
   ```
2. **General Pattern**
   - **Command substitution** wraps a pipeline or single command.
   - Use grep, awk, etc., to isolate exactly the data you need.

4. Complete “System Health” Script Example

```
#!/bin/bash
# Welcome message with system variables
echo "Welcome $USER on $HOSTNAME"
# Free RAM (MB)
FREE_RAM=$(free -m | grep Mem | awk '{print $4}')
echo "Available free RAM: ${FREE_RAM}MB"
# Load average (1-minute)
LOAD=$(uptime | awk -F 'load average: ' '{print $2}' | cut -d',' -f1)
echo "Current load average: $LOAD"
# Free space on root partition
ROOT_FREE=$(df -h / | tail -1 | awk '{print $4}')
echo "Free space on /: $ROOT_FREE"
```

- **Key Points:**
  - $USER and $HOSTNAME are built-in system variables.
  - Command substitution ($(...)) is used to assign complex command outputs to variables.
  - awk, grep, and cut help parse only the needed information.

5. Practical Use

- Such a script can run at **login** or via **cron** to report system health.
- You’ll reuse command-substitution techniques in more advanced scripts (e.g., multi-server monitoring).

Bash Scripting: Exporting Variables

1. Variable Scope in Bash

- **Local Variables:**
  - Declared with NAME=value in a shell session.
  - **Lost** when the shell exits (e.g., on logout) or in child shells.

2. Demonstration of Local Scope

1. **Declare & Use:**
   ```
   SEASON="Monsoon"
   echo $SEASON      # Outputs: Monsoon
   ```
2. **After Logout & Re-login:**
   ```
   exit          # Closes shell, variable gone
   sudo -i         # Starts new root shell
   echo $SEASON      # No output (variable not defined)
   ```
3. **Child Shell Inheritance:**
   - Running a script launches a **child shell**.
   - Local parent variables are **not** available in the child.

3. Making Variables Available to Child Shells

- **export NAME=value**
  - Marks NAME for **inheritance** by all child processes of the current shell.
  - **Temporary**: Remains only for the duration of the current login session.

  ```
  export SEASON
  bash testvars.sh   # In testvars.sh, echo $SEASON now works
  ```

4. Persisting Variables Across Logins

Scope	File to Edit	Applied When
User-specific	~/.bashrc or ~/.bash_profile	Each time that **user** logs in
Global	/etc/profile	Every **user** on the system at login

1. **Per-User Persistence**
   - Add to ~/.bashrc:
     ```
     export SEASON="Monsoon"
     ```
   - On next login, echo $SEASON outputs Monsoon.
2. **System-Wide Persistence**
   - Add to /etc/profile:
     ```
     export SEASON="Winter"
     ```
   - Affects **all users** on next login.

5. File Sourcing Order & Precedence

1. **Global**: /etc/profile is sourced first.
2. **User-specific**: ~/.bashrc (or ~/.bash_profile) is sourced next.

3. **Result:**
    - If the **same variable** is set in both, the **user's** setting in ~/.bashrc **overrides** the global one for that user.
    - Other users without a local override will use the global value.

6. Summary of Commands & Files

- **Export Command:**
  export NAME=value
- **Per-User Files:** ~/.bashrc, ~/.bash_profile
- **System-Wide File:** /etc/profile
  Reminder:
    - Use export to share variables with child shells.
    - Edit the appropriate file to make variables persist across logins.

Bash Scripting: User Input with read

1. Making Scripts Interactive

- **Goal:** Prompt the user for input and use their responses within the script.
- **Command:** read

2. Basic read Usage

1. **Prompt & Store**

   ```
   #!/bin/bash
   echo "Enter your skill:"
   read SKILL
   echo "Your skill is: $SKILL"
   ```

    - Script prints a prompt, then **pauses** at read until the user types a value and presses **Enter**.
    - The entered value is assigned to the variable (SKILL).
2. **Accessing the Input**

   ```
   echo "You entered: $SKILL"
   ```

3. read Options

Option	Description
-p	**Prompt**: Display text before waiting for input
-s	**Silent**: Don't echo typed characters (for secrets)

Example with -p

```
read -p "Enter your skill: " SKILL
echo "Your skill is: $SKILL"
```

Example with -s (password entry)

```
read -p "Username: " USERNAME
read -sp "Password: " PASSWD
echo
echo "Username is $USERNAME"
# Do NOT echo $PASSWD to keep it secret
```

4. When to Use (and Avoid) Interactive Scripts

- **Use Cases:**
  - One-off administrative scripts requiring confirmation or password input.
- **Caution:**
  - **Not recommended** for automated DevOps pipelines or background jobs, where **non-interactive** execution is essential.
  - User prompts can cause scripts to hang or fail when no one is present to respond.

Bash Scripting: Decision Making with if Statements

1. Why Decision Making?

- Up until now, scripts are **linear**: they run commands in sequence.
- **Decision making** allows scripts to be **smart**, branching logic based on conditions (success/failure, variable values).

2. if Statement Structure

```
if [ <condition> ]; then
  # Commands to run when condition is true
elif [ <another_condition> ]; then
  # Commands for alternate condition
else
  # Commands when all conditions fail
fi
```

- **if**: Begins the decision block.
- **[…]**: Test expression; spaces around brackets and operators are **mandatory**.
- **-gt**, **-lt**, **-eq**, etc.: Numeric comparison operators.
- **then**: Follows the condition.
- **elif**: (Else if) Optional additional test.
- **else**: Fallback when no prior condition is true.
- **fi**: Ends the if block (reverse of if).

3. Example 1: Simple if / No else

Script: 8if1.sh

```
#!/bin/bash
# Prompt the user
echo "Enter a number:"
read NUM
# Test if NUM > 100
if [ $NUM -gt 100 ]
  then
    echo "Entered the IF block..."
    sleep 3
    echo "Your number ($NUM) is greater than 100."
    echo
    date
fi
```

```
echo "Execution completed."
```

- **Test:**
    - -gt means “greater than.”
    - Both operands and brackets must be separated by spaces: [$NUM -gt 100].
- **Behavior:**
    - If **true**, runs inside the if block (sleep, message, date).
    - If **false**, skips the block and continues after fi.

4. Testing Example 1

- **Case 1:** Input 120
  ```
  Enter a number:
  120
  Entered the IF block...
  (3-second pause)
  Your number (120) is greater than 100.

  <current date>
  Execution completed.
  ```
- **Case 2:** Input 50
  ```
  Enter a number:
  50
  Execution completed.
  ```

5. Example 2: Adding an else Block

Script: 9if1_else.sh

```
#!/bin/bash
echo "Enter a number:"
read NUM
if [ $NUM -gt 100 ]
 then
  echo "Your number ($NUM) is greater than 100."
else
  echo "Your number ($NUM) is less than or equal to 100."
fi
```

- **else** block executes only when the if condition is **false**.
- Provides clear output for both branches.

6. Extending to elif

```
#!/bin/bash
echo "Enter a number:"
read NUM
if [ $NUM -gt 100 ]
  then
  echo "Greater than 100."
elif [ $NUM -eq 100 ]
  then
  echo "Equal to 100."
else
  echo "Less than 100."
fi
```

- **elif** allows multiple conditional checks in sequence.

7. Key Points & Best Practices

- Always include **spaces** around [ ] and operators (-gt, -lt, etc.).
- Use fi to close every if.
- Combine if / elif / else for comprehensive branching.
- Indentation and blank lines improve readability (optional but recommended).

Bash Scripting: Multiple Conditions with elif

1. Purpose of elif

- **if … else** handles a single test and a fallback.
- **elif** (“else if”) allows **multiple** conditional checks in sequence before a final else.
- Structure:

  ```
  if [ condition1 ]; then
    # commands for condition1 true
  elif [ condition2 ]; then
    # commands for condition2 true
  else
    # commands if all conditions false
  fi
  ```

2. Use Case: Counting Active Network Interfaces

1. **Goal:** Count non-loopback network interfaces and branch on the count:
   - **1 interface** → “One active interface found”
   - **>1 interfaces** → “Multiple active interfaces found”
   - **0 interfaces** → “No active interface found”
2. **Identify Active Interfaces:**

   ```
   ip addr show | grep -v LOOPBACK | grep -ic mtu
   ```

   - ip addr show → lists all interfaces.
   - grep -v LOOPBACK → exclude loopback entries.
   - grep -ic mtu → match lines containing “mtu” (case-insensitive).

3. Storing Count in a Variable

```
count=$(ip addr show | grep -v LOOPBACK | grep -ic mtu)
```

4. Script Example: 10_ifelif.sh

```
#!/bin/bash
value=$(ip addr show | grep -v LOOPBACK | grep -ic mtu)

# Decision making with if-elif-else
if [ $value -eq 1 ]
 then
  echo "Found one active interface."
elif [ $value -gt 1 ]
 then
```

```
   echo "Found multiple active interfaces ($value)."
else
  echo "No active interface found."
fi
```

- **[$value-eq 1] – true if exactly one.**
- **elif [$value-gt 1] – next test if more than one.**
- **else – fallback when count is zero.**

5. Testing the Script

1. **Make Executable:**
   chmod +x 10_ifelif.sh
2. **Run:**
   ./10_ifelif.sh
3. **Expected Output (on this system):**
   Found multiple active interfaces (2).

6. Key Takeaways

- **elif** enables multiple, ordered condition checks within a single if block.
- Only **one** branch executes—the first whose condition is true.
- Always close with a **single** fi.

Bash Scripting: Process Monitoring & Automated Recovery

1. Operator Refresher

- **Negation:** ! condition → true becomes false, false becomes true.
- **String Tests:**
  - -n str → true if string length > 0
  - -z str → true if string length = 0
- **Numeric Comparisons:**
  - -eq, -ne, -gt, -lt, -ge, -le
  - Alternatives: = for equality, != for inequality
- **File Tests (single-operand):**
  - -f file → true if **file** exists
  - -d dir → true if **directory** exists
  - -e path → true if **file or directory** exists
  - -r file → true if **read permission** is set

2. Using Exit Codes ($?)

- **$?** holds the exit status of the last command:
  - **0** → success (interpreted as **true** in Bash)
  - **non-zero** → failure (**false**)
- **Example:**

```
ls /nonexistent
echo $?   # Outputs: 1 (error)
```

3. Detecting httpd Status via PID File

- **PID file path:** /var/run/httpd/httpd.pid
  - Exists only when Apache (httpd) is running.
- **Manual Check:**
  ```
  sudo systemctl stop httpd
  ls /var/run/httpd/httpd.pid   # “No such file” → exit code ≠ 0
  echo $?                       # Non-zero indicates stopped
  sudo systemctl start httpd
  ls /var/run/httpd/httpd.pid   # File now exists → exit code = 0
  echo $?                       # 0 indicates running
  ```

4. Writing the Monitor Script (11_monit.sh)

```
#!/bin/bash
# Constants
PIDFILE="/var/run/httpd/httpd.pid"
LOG_TS=$(date '+%Y-%m-%d %H:%M:%S')
echo "[$LOG_TS] Checking httpd process..."
# 1. Check if PID file exists
if [ -f "$PIDFILE" ]; then
  echo "httpd process is running."
else
  echo "httpd process is NOT running. Attempting to start..."
  sudo systemctl start httpd >/dev/null
# 2. Verify restart succeeded
  if [ $? -eq 0 ]; then
    echo "Successfully started httpd."
  else
    echo "Failed to start httpd! Please contact admin."
  fi
fi
```

- **Nested if** inside the else block to handle start-failure.
- Redirected systemctl start output to /dev/null—only the exit code matters.
- Timestamps and clear echo statements improve log readability.
- Mention of **Monit** tool: this script mimics its basic behavior.

5. Automating with Cron

1. **Locate Script:**
   SCRIPT=/opt/scripts/11_monit.sh
2. **Edit Crontab:**
   crontab -e
3. **Cron Schedule Fields:**
   ```
   ┌───────────── Minute (0–59)
   │ ┌─────────── Hour (0–23)
   │ │ ┌───────── Day of Month (1–31)
   │ │ │ ┌─── Month (1–12)
   │ │ │ │ ┌─ Day of Week (0–7; Sunday=0 or 7)
   │ │ │ │ │
   * * * * *  command-to-run
   ```

4. **Example Entry:**
   ```
   * * * * * /opt/scripts/11_monit.sh >> /var/log/monit_httpd.log 2>&1
   ```
   - Runs **every minute**.
   - **>> /var/log/monit_httpd.log 2>&1** appends both **stdout** and **stderr** to the log file.

6. Verifying Automated Monitoring

- **Stop Apache:**
  ```
  sudo systemctl stop httpd
  ```
- **Wait One Minute**, then inspect the log:
  ```
  tail /var/log/monit_httpd.log
  ```
- **Expected Log Sequence:**
  ```
  [YYYY-MM-DD HH:MM:SS] Checking httpd process...
  httpd process is NOT running. Attempting to start...
  Successfully started httpd.
  [Next timestamp] Checking httpd process...
  httpd process is running.
  ```
- **Repeat Stop/Start:** Logs capture each recovery attempt.

7. Alternative Implementation Using -f

- Replace exit-code test with file-test operator:
  ```
  if [ -f "$PIDFILE" ]; then
    echo "httpd process is running."
  else
    # restart logic...
  fi
  ```
- Both approaches are valid; choose based on preference and clarity.

8. Next Steps

- **Loops:** In the next lecture, learn **while** and **for** loops to handle repetitive tasks and batch processing.

Bash Scripting: Loops

1. Introduction to Loops

- **Purpose:** Automate repetitive tasks by running a block of commands multiple times.
- **Types Covered:**
  - **for loops:** Iterate over a fixed list or sequence.
  - **while loops:** (To be covered next) run until a condition becomes false.

2. When to Use a for Loop

- **Fixed iterations:** You know in advance how many times to run (e.g., process a list).
- **Examples:**
  - Adding multiple users where only the username changes.
  - Executing commands on a predefined list of servers.
  - Iterating over arrays, filenames, or numeric ranges.

3. for Loop Syntax

```
for VAR in item1 item2 item3 …; do
  # Commands using $VAR
done
```

- **VAR**: Loop variable that takes each value in turn.
- **in**: Introduces the list to iterate over.
- **do**…**done**: Encloses the loop body.

4. Example 1: Looping Over Languages (13_for.sh)

```
#!/bin/bash
for VAR1 in Java .NET Python Ruby PHP; do
  echo "Looping… VAR1 = $VAR1"
  sleep 1
done
```

- **Sequence:** Java → .NET → Python → Ruby → PHP
- **Behavior:**
  1. Assign VAR1=Java, run echo & sleep.
  2. Assign VAR1=.NET, repeat.
  3. Continue until PHP.
- **sleep 1** slows output for readability.

5. Example 2: Adding Multiple Users (14_for.sh)

```
#!/bin/bash
# Define list of usernames
myusers="Alpha Beta Gamma"
for usr in $myusers; do
  echo "####################################"
  echo "Adding user: $usr"
  echo "####################################"
  sudo useradd "$usr"
  id "$usr"
  echo
done
```

- **myusers**: Space-separated list of usernames.
- **Loop Body:**
  1. Print section header and the username.
  2. Run useradd $usr to create the user.
  3. Run id $usr to confirm creation and display UID/GID.
  4. Blank line for separation.

6. Key Takeaways

- **for loops** are ideal when you have a known set of items.
- Always close with **done**.
- Loop variable (VAR1, usr, etc.) holds the current item.
- Use **quotes** around variables when they may contain spaces.
- **Enhance readability** with headers, blank lines, and optional delays (sleep).

Bash Scripting: while Loops

1. Introduction to while Loops

- **Purpose:** Execute a block of commands **repeatedly** as long as a **condition** remains true.
- Contrasts with for loops, which iterate over a fixed list or range.

2. Basic while Loop Syntax

```
while [ condition ]; do
  # Commands to execute while condition is true
done
```

- **[condition]**: Test expression, same syntax as in if statements.
- **do**…**done**: Encloses the loop body.

3. Example 1: Counting Up (15_while.sh)

```
#!/bin/bash
counter=0
while [ $counter -lt 5 ]; do
  echo "Looping… counter = $counter"
  sleep 1
  # Increment counter to avoid infinite loop
  counter=$((counter + 1))
done

echo "Out of the loop."
```

1. **Initialization:** counter=0
2. **Condition:** [$counter -lt 5] (less than 5)
3. **Body:**
   - Prints the current value.
   - Sleeps one second for readability.
   - **Critical:** Updates counter (counter=$((counter + 1))).
4. **Termination:** When counter reaches 5, the condition is false and the loop exits.

4. Infinite Loop Warning

- **Without incrementing counter,** the condition stays true forever (0 < 5) → **infinite loop**.
- **Termination:** Must press **Ctrl +C** to kill the script.
- Caution: Running infinite loops in background (e.g., via cron) can overload the system.

5. Example 2: Explicit Infinite Loop (16_while.sh)

```
#!/bin/bash
value=2
while true; do
  echo "Current value: $value"
  sleep 1
  # Double the value each iteration
  value=$((value * 2))
```

done

- **while true**: A Boolean expression that always evaluates to true → never exits on its own.
- **Dynamic Behavior:** The loop body multiplies value by 2 each time, demonstrating real-time changes.
- **Use Case:** Infinite loops can be useful for daemons or watchers but **must** include internal exit logic or external controls (signals).

6. Key Takeaways

- **while loops** depend on a **runtime condition** rather than a fixed list.
- Always ensure the loop's condition will eventually become false, or provide an explicit break.
- Use sleep to pace loops when monitoring or producing human-readable output.
- For guaranteed infinite loops, use while true; do …; done but handle termination carefully.

Bash Scripting: Remote Execution

1. Objective

- Execute commands **from** the **scriptbox** VM **on** multiple target VMs (web01, web02, and a new web03 Ubuntu VM) via SSH.

2. Extending the Vagrant Setup

- **Add a third VM** (web03) in your **Vagrantfile**:
  - **Box:** ubuntu/bionic64 (Ubuntu 18.04)
  - **IP:** 10.0.0.15 (matching your network but in lecture: 10.13, 10.14, 10.15)
- **Bring up** all VMs:
  ```
  vagrant up
  ```

3. Hostname Configuration on Each VM

1. **SSH into each** (from host or scriptbox):
   ```
   vagrant ssh web01
   sudo -i
   echo web01 > /etc/hostname
   logout
   ```
2. **Repeat** for web02 and web03, setting /etc/hostname to web02 and web03 respectively.

4. Name Resolution on scriptbox

- **Edit** /etc/hosts on **scriptbox** to map VM names to IPs:
  ```
  10.0.0.13   web01
  ```

```
10.0.0.14   web02
10.0.0.15   web03
```

- **Test** name resolution and connectivity:
  ```
  ping -c1 web01
  ping -c1 web02
  ping -c1 web03
  ```

5. SSH Access & Creating a devops User

1. **Initial SSH Login**
   - web01/web02 (CentOS): ssh vagrant@web01 → password vagrant works.
   - web03 (Ubuntu): initial SSH fails with “Permission denied (publickey)”—password login disabled by default.
2. **Enable Password Login on web03**
   ```
   sudo -i
   sed -i 's/^PasswordAuthentication no/PasswordAuthentication yes/' /etc/ssh/sshd_config
   systemctl restart ssh
   ```
   - Now ssh vagrant@web03 prompts for and accepts the vagrant password.
3. **Create devops User on All VMs**
   ```
   sudo adduser devops
   sudo passwd devops
   ```
4. **Grant Sudo Without Password**

   ```
   export EDITOR=vim
   visudo
   # Add line:
   devops ALL=(ALL) NOPASSWD:ALL
   ```

6. Remote Command Execution via SSH

- **Basic Remote Command:**
  ```
  ssh devops@web01 uptime
  ```
  - Prompts for devops password, runs uptime **on** web01, then returns control to **scriptbox**.
- **Benefit:** No interactive shell needed; commands can be scripted.

7. Next Steps

- Use SSH-based execution in your scripts to automate actions (e.g., package installs, user additions) across all target VMs in a single loop or function.
- Incorporate **error checking** ($?) and **logging** for each remote step.

Bash Scripting: SSH Key-Based Authentication

1. Password-Based vs. Key-Based SSH

- **Password-based:**
  - Prompts each time for a user’s password.
  - Less secure; vulnerable to brute-force.
- **Key-based:**
  - Uses an asymmetric key pair (private + public).
  - More secure and **password-less** after setup.

2. Generating an SSH Key Pair on scriptbox

1. **Run**
   ```
   ssh-keygen
   ```
2. **Prompts:**
   - File location (default: ~/.ssh/id_rsa) → **Enter**
   - Passphrase → press **Enter** twice (empty passphrase for simplicity)
3. **Result:**
   - **Private key:** ~/.ssh/id_rsa
   - **Public key:** ~/.ssh/id_rsa.pub
4. **Analogy:**
   - Public key = **lock**
   - Private key = **key** that opens it

3. Distributing the Public Key to Target VMs

- **Command:**
  ```
  ssh-copy-id devops@web01
  ssh-copy-id devops@web02
  ssh-copy-id devops@web03
  ```
- **Process:**
  1. Prompts for devops password on each host.
  2. Appends id_rsa.pub contents to ~/.ssh/authorized_keys on the remote VM.

4. Verifying Key-Based Login

- **Without Password Prompt:**
  ```
  ssh devops@web01 uptime
  ```
  - No password requested; uses private key automatically.
  - Default invocation is equivalent to:
    ```
    ssh -i ~/.ssh/id_rsa devops@web01 uptime
    ```

5. Understanding Key Files

- **Private Key (id_rsa):**
  ```
  cat ~/.ssh/id_rsa
  # Begins with: -----BEGIN RSA PRIVATE KEY-----
  # Longer blob of characters.
  ```
- **Public Key (id_rsa.pub):**
  ```
  cat ~/.ssh/id_rsa.pub
  # Begins with: ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQC...
  # Shorter than the private key.
  ```
- **Pairing:**
  - The **public key** in authorized_keys is the “lock.”
  - The **private key** in ~/.ssh is the “key.”

    - SSH authenticates by matching them.

6. Summary

1. **Generate** an SSH key pair (ssh-keygen).
2. **Deploy** the public key to each target (ssh-copy-id).
3. **Verify** password-less SSH and remote command execution.
4. **Benefit:** Scripts can SSH and run commands on remote servers without interactive password entry.

Bash Scripting: Remote Multi-OS Web Setup Framework

1. Overview & Goal

- **Objective:** Combine all learned concepts to create a **remote execution framework** that:
    1. Reads a list of target hosts from a file
    2. SSHes into each host in a loop
    3. Detects the OS (CentOS vs. Ubuntu)
    4. Performs Web server setup on each accordingly

2. Preparing the Inventory (“hosts”) File

1. **Create a directory** for remote-web-setup scripts:
   ```
   mkdir -p ~/remote_web_setup
   cd ~/remote_web_setup
   ```
2. **Create** a plain-text inventory file (e.g., remote_hosts):
   ```
   web01
   web02
   web03
   ```
    - Each line is a target hostname (already defined in /etc/hosts on scriptbox).
    - Can list IPs instead of names.

3. Testing SSH in a Loop

- **Loop over hosts** using a for loop with **command substitution** to read lines from the file:
  ```
  for host in $(cat remote_hosts); do
    echo "Connecting to $host..."
    ssh devops@$host hostname
  done
  ```
- **Explanation:**
    - for host in $(cat remote_hosts); do … done iterates one hostname per iteration.
    - ssh devops@$host hostname runs hostname remotely and returns to the local shell.

4. Demonstrating OS-Specific Failures

- **Example:** Running sudo yum install git on each host:
  ```
  for host in $(cat remote_hosts); do
    ssh devops@$host "sudo yum install git -y"
  done
  ```
- **Result:**
  - **web01/web02 (CentOS):** succeeds.
  - **web03 (Ubuntu):** fails with "yum: command not found"—needs apt.

5. Building the Multi-OS Setup Script (multios_websetup.sh)

1. **Copy Base Script:**
   ```
   cp ~/scripts/3_vars_websetup.sh ./multios_websetup.sh
   ```
2. **Wrap in OS Detection Logic:**
   ```
   #!/bin/bash

   # Determine OS by testing 'yum' availability
   if yum --help >/dev/null 2>&1; then
     echo "   Running setup on CentOS"
     PACKAGE="httpd wget unzip"
     SVC="httpd"
     # (Other CentOS-specific variable assignments...)
   else
     echo "   Running setup on Ubuntu"
     PACKAGE="apache2 wget unzip"
     SVC="apache2"
     sudo apt update >/dev/null
     # (Other Ubuntu-specific variable assignments...)
   fi

   # Common deployment steps using $PACKAGE and $SVC
   sudo yum install $PACKAGE -y    # or 'sudo apt install' on Ubuntu
   sudo systemctl start $SVC
   sudo systemctl enable $SVC
   mkdir -p /tmp/webfiles && cd /tmp/webfiles
   wget $URL -q
   unzip ${ARTIFACT}.zip >/dev/null
   sudo cp -r ${ARTIFACT}/* /var/www/html/
   sudo systemctl restart $SVC
   rm -rf /tmp/webfiles
   ```
3. **Key Points:**
   - **OS Detection:** if yum --help >/dev/null 2>&1; then … else … fi.
   - **Variable Reuse:** $PACKAGE, $SVC, $URL, and $ARTIFACT drive all commands.
   - **Spacing & Comments:** Improves readability of each block.

6. Next Steps

- **Remote Deployment Execution:** In the following lecture, wrap this multios_websetup.sh in a loop that SSHes to each entry in remote_hosts, copies the script, and executes it remotely—achieving **fully automated, multi-OS web setup** across all target servers.

Bash Scripting: Remote Deployment & Execution

1. Objective

- **Automate** pushing and executing the multi-OS web setup script (multios_websetup.sh) on remote hosts (web01, web02, web03).
- Leverage SSH and SCP within a loop to manage multiple machines in one go.

2. Using scp to Push Files

1. **Basic Syntax:**
   ```
   scp source_file devops@hostname:/destination/path/
   ```
2. **Example:**
   ```
   echo "test" > testfile.txt
   scp testfile.txt devops@web01:/tmp/
   ```
3. **Permissions:**
   - Cannot write to root-owned directories without sudo.
4. **Underlying Mechanism:**
   - Uses SSH and your private key (~/.ssh/id_rsa) for password-less transfers once keys are set up.

3. Testing SCP

- **Fetch Files:** Reverse scp usage—swap source and destination.
- **No Password Prompt:** Key-based SSH authentication handles SCP seamlessly.

4. Writing the Deployment Script (web_deploy.sh)

1. **Shebang & Loop Over Hosts:**
   ```
   #!/bin/bash

   USR=devops
   for host in $(cat remhosts); do
     echo "#####################################"
     echo "Connecting to $host..."
     echo "#####################################"
   ```
2. **Push the Script via SCP:**
   ```
     echo "Pushing multios_websetup.sh to $host:/tmp/"
     scp multios_websetup.sh $USR@$host:/tmp/
   ```
3. **Execute Remotely via SSH:**
   ```
     echo "Executing setup on $host..."
     ssh $USR@$host "bash /tmp/multios_websetup.sh"
   ```
4. **Cleanup Remote Script:**
   ```
     echo "Cleaning up on $host..."
     ssh $USR@$host "rm /tmp/multios_websetup.sh"
   ```

done

5. **Cosmetic Echoes:**
   - Section headers (####) and descriptive messages for each phase: connecting, pushing, executing, cleaning.

5. Making & Running the Script

1. **Make Executable:**
   chmod +x web_deploy.sh
2. **Execute:**
   ./web_deploy.sh
   - Processes each host in sequence.
   - Detects CentOS vs. Ubuntu within the deployed script and runs appropriate commands.

6. Verification via Browser

- **Access each Web server** by IP or hostname in a browser:
  - http://web01/, http://web02/, http://web03/
- **Confirm** the website template is deployed and service is running.

7. Key Takeaways

- **SCP + SSH in a loop** provides powerful, scalable remote automation—ideal for hundreds of servers.
- The **multi-OS script** demonstrates OS detection (yum vs. apt) with variables and conditionals.
- **Cleanup** steps ensure no residual scripts remain on targets.
- Mastery of these Bash patterns is a solid foundation before moving to sophisticated tools like Ansible.