

can you make detailed notes from this pdf in 5-6 parts, dont miss even a single thing, its very important you dont miss anything, we will start by giving part 1



Based on the uploaded document, I have organized the comprehensive Bash Scripting notes into **6 Parts** to ensure no detail is missed.

Here is **Part 1: Environment Setup, Basics, and First Automation Project**.

Part 1: Bash Basics, Environment Setup & The First Deployment Script

1. Introduction to Bash Scripting

- **Purpose:** Bash scripting provides a solid platform to automate day-to-day system administration tasks.
 - **Usage:** Used by system admins and DevOps engineers to get things done quickly and efficiently.
 - **Comparison:** While sophisticated tools like Ansible, Puppet, and Chef exist, Bash is often faster for quick Linux tasks. Understanding scripting helps in grasping Configuration Management tools later.
-

2. The "Scriptbox" Project Environment

To practice, a specific Vagrant environment is set up.

A. VM Architecture

There are 4 Virtual Machines (VMs) in this setup:

1. **scriptbox** (CentOS 7): The main machine where you write and run scripts. IP: `192.168.10.12`.
2. **web01** (CentOS 7): Remote target 1. IP: `192.168.10.13`.
3. **web02** (CentOS 7): Remote target 2. IP: `192.168.10.14`.
4. **web03** (Ubuntu 18.04): Remote target 3. IP: `192.168.10.15`.

B. Setup Steps

1. **Directory:** Create a folder (e.g., `D:/bash-scripts`) and place the `Vagrantfile` there .
 2. **Booting:** Run `vagrant up scriptbox` to start the main VM.
 3. **Access:** Run `vagrant ssh scriptbox` to log in.
 4. **Hostname:** Inside the VM, switch to root (`sudo -i`) and set the hostname: `echo "scriptbox" > /etc/hostname` .
 5. **Workspace:** Create a dedicated directory for scripts: `mkdir -p /opt/scripts` .
 6. **Tools:** Install vim: `yum install vim -y` .
-

● 3. Writing the First Script

The first script monitors system health (Uptime, Memory, Disk).

File Name: `firstscript.sh` **Location:** `/opt/scripts/`

Script Content:

Bash



```
#!/bin/bash

### This script prints system info ###

echo "#####"
echo "The uptime of the system is:"
uptime
echo

echo "#####"
echo "Memory Utilization"
free -m
echo

echo "#####"
echo "Disk Utilization"
df -h
```

Key Concepts Explained:

- **Shebang (`#!/bin/bash`):** Tells the Operating System to use the Bash interpreter to run the file.
- **Comments (`#`):** Ignored by the shell; used for readability.
- **Execution Permissions:** By default, a file is not executable. You must run:

- `chmod +x firstscript.sh`.
 - **Running the script:**
 - Relative path: `./firstscript.sh`.
 - Absolute path: `/opt/scripts/firstscript.sh`.
-

● 4. Project: Website Setup Automation (Standalone)

Goal: Automate the manual steps of installing Apache, downloading a template, and deploying it .

File Name: `websetup.sh`

The Full Script:

Bash



```
#!/bin/bash

# Installing Dependencies
echo "#####"
echo "Installing packages."
echo "#####"
sudo yum install wget unzip httpd -y > /dev/null
echo

# Start & Enable HTTPD Service
echo "#####"
echo "Start & Enable HTTPD Service"
echo "#####"
sudo systemctl start httpd
sudo systemctl enable httpd
echo

# Creating Temp Directory
echo "#####"
echo "Starting Artifact Deployment"
echo "#####"
mkdir -p /tmp/webfiles
cd /tmp/webfiles
echo

# Downloading and Extracting Template
wget https://www.tooplate.com/zip-templates/2098_health.zip > /dev/null
unzip 2098_health.zip > /dev/null
sudo cp -r 2098_health/* /var/www/html/
echo

# Bounce Service (Restart)
echo "#####"
echo "Restarting HTTPD service"
```

```

echo "#####"
systemctl restart httpd
echo

# Clean Up
echo "#####"
echo "Removing Temporary Files"
echo "#####"
rm -rf /tmp/webfiles
echo

# Status Check
echo "#####"
echo "Checking HTTPD Status and Content Deployed"
echo "#####"
sudo systemctl status httpd
ls /var/www/html/

```

Technical Breakdown:

- `yum install ... -y` : Installs packages non-interactively (auto-confirms "yes").
- `> /dev/null` : Redirects standard output to "nowhere" (hides the noise), but allows errors to still appear on the screen.
- `mkdir -p` : Creates the directory only if it doesn't exist (prevents errors).
- `wget` : Downloads the zip file from the internet.
- `unzip` : Extracts the zip file.
- `cp -r` : Recursively copies the folder content to `/var/www/html/` (Apache's root folder).
- `rm -rf` : Forcefully cleans up the temporary directory.
- `systemctl status` + `ls` : Used for final verification that the service is running and files exist.

Verification:

To verify the script worked, find the VM's IP using `ip addr show` and visit `http://<VM-IP>` in a browser. You should see the "Health Center" template.

End of Part 1. *Would you like me to proceed to Part 2, which covers Variables (System & User), Arguments, and refactoring the scripts?*

yes



Here is **Part 2: Variables, Arguments, and Script Intelligence**.

This section focuses on making your scripts dynamic, reusable, and "intelligent" by moving away from hardcoded values.

Part 2: Variables, Arguments, and System Logic

1. Variables in Bash

Variables are temporary storage for information in memory, allowing you to re-use values throughout a script.

A. Why use Variables?

1. **Avoid Repetition:** Write a value (like a package name) once, use it everywhere.
2. **Easy Updates:** Changing a URL or folder name in one place updates the entire script.
3. **Clarity:** Names like `TEMPDIR` or `URL` make the code self-explanatory.

B. Syntax Rules

- **Assigning:** `VARIABLE_NAME=Value` (No spaces around `=`).
- **Accessing:** `echo $VARIABLE_NAME` (Use `$` to read the value).

C. Refactoring the Web Setup Script

We transform the hardcoded `websetup.sh` into `3_vars_websetup.sh` using variables.

Key Changes in the Script:

Bash



```
#!/bin/bash

# Variable Declaration
PACKAGE="httpd wget unzip"
SVC="httpd"
URL='https://www.tooplate.com/zip-templates/2098_health.zip'
ART_NAME='2098_health'
TEMPDIR="/tmp/webfiles"

# Installing Dependencies
sudo yum install $PACKAGE -y > /dev/null

# Start & Enable Service
sudo systemctl start $SVC
sudo systemctl enable $SVC

# Deploying Artifact
mkdir -p $TEMPDIR
```

```
cd $TEMPDIR
wget $URL > /dev/null
unzip $ART_NAME.zip > /dev/null
sudo cp -r $ART_NAME/* /var/www/html/

# Cleanup
rm -rf $TEMPDIR
```

● 2. Command-Line Arguments

Instead of hardcoding values *inside* the script, we can pass them *when running* the script. This makes the script generic (e.g., capable of deploying *any* website, not just the "Health" template) .

A. Positional Parameters

Bash automatically assigns special variables based on the order of arguments passed:

- `$0` : The name of the script itself.
- `$1` : The first argument passed.
- `$2` : The second argument passed.
- ...up to `$9` (use `${10}` for higher numbers).

B. Project: Generic Deployment Script

We create `5_args.sh` to accept the URL and Folder Name from the user.

Refactored Code:

Bash



```
#!/bin/bash

# $1 is the Download URL
# $2 is the Artifact Name (Unzipped folder name)

wget $1 > /dev/null
unzip $2.zip > /dev/null
sudo cp -r $2/* /var/www/html/
```

How to Run:

Bash



```
./5_args.sh https://www.tooplate.com/zip-templates/2103_ziggy.zip 2103_ziggy
```

● 3. System Variables & Exit Status

Bash provides built-in variables that offer information about the system and the script's execution state.

A. Common System Variables

Variable	Description
\$0	Script Name
\$#	Total number of arguments passed
\$@	All arguments passed (as a list)
\$USER	Current logged-in user
\$HOSTNAME	Machine hostname
\$RANDOM	A random number
\$?	Exit Status of the last command



B. Understanding Exit Status (\$?)

- `0` : Success.
- `1-255` : Failure (Specific numbers like `127` indicate "Command not found").
- **Usage:** Crucial for error handling. If a download fails, we shouldn't try to unzip it.

Example of Error Checking:

Bash



```
wget $1 > /dev/null
if [ $? -ne 0 ]; then
    echo "Failed to download from $1"
```

```
    exit 2
fi
```

● 4. Quotes: Single vs. Double

Understanding how Bash treats text inside quotes is vital for variable handling.

A. Double Quotes (" ")

- **Behavior:** Allow variable expansion (interpreter reads the variable's value).
- **Example:** `echo "My skill is $SKILL"` → Output: `My skill is DevOps` .
- **Escaping:** Use `\` to treat special characters literally. e.g., `\$` prints a literal dollar sign instead of a variable.

B. Single Quotes (' ')

- **Behavior:** Strong quoting. No expansion. Everything is treated as plain text.
- **Example:** `echo 'My skill is $SKILL'` → Output: `My skill is $SKILL` .

● 5. Variable Scope & Persistence

Variables defined in a shell are **local** to that shell. They disappear when you exit or run a child script unless handled properly.

A. Exporting Variables

To make a variable available to child scripts, use `export` .

Bash



```
export SEASON="Monsoon"
./testvars.sh # Now this script can see $SEASON
```

B. Making Variables Permanent

To keep variables across reboots or new sessions, add them to configuration files:

1. `~/.bashrc` : Permanent for the **current user** only.
2. `/etc/profile` : Permanent for **all users** (Global).

Precedence: User settings (`.bashrc`) override global settings (`/etc/profile`).

End of Part 2. *Would you like me to proceed to Part 3? It will cover User Interaction, Decision Making (If/Else), and Process Monitoring.*

yes

✦ Here is **Part 3: User Interaction, Conditional Logic, and Process Monitoring**.

This section moves from simple linear scripts to intelligent scripts that can make decisions, handle user input, and self-heal system services.

■ Part 3: User Interaction, Logic & Monitoring

● 1. Taking User Input

While DevOps aims for full automation, sometimes scripts need to be interactive (e.g., CLI tools). We use the `read` command for this.

A. Basic Syntax

- `read VARIABLE` : Pauses the script, waits for the user to type, and stores the input in `$VARIABLE` .

B. Advanced Flags

- `-p` (**Prompt**): Displays a message on the same line as the input field.
 - *Example:* `read -p "Enter Username: " USR` .
- `-s` (**Silent**): Hides typed characters (essential for passwords).
 - *Example:* `read -sp "Enter Password: " PASS` .

C. DevOps Best Practice: Avoid Interactivity

Interactive scripts are **not recommended** for CI/CD pipelines (like Jenkins or Ansible) because:

1. **Non-automatable:** Tools cannot physically type input in real-time.
 2. **Error-prone:** Humans make typos.
 3. **Solution:** Pass values as **arguments** (Part 2) or use environment variables/vaults instead.
-

● 2. Decision Making: `if` and `else`

Conditional logic allows a script to execute code *only if* specific criteria are met (e.g., "Install Apache only if it's missing").

A. Basic Syntax

Bash



```
if [ CONDITION ]
then
    # Commands to run if TRUE
else
    # Commands to run if FALSE
fi
```

Note: Spaces inside the brackets `[]` are mandatory (e.g., `[$NUM -gt 100]`).

B. Comparison Operators (Integers)

Operator	Meaning
-eq	Equal to
-ne	Not equal to
-gt	Greater than
-lt	Less than
-ge	Greater or equal to
-le	Less or equal to



C. `elif` (Else-If)

Used for multiple checks. *Example: Detecting Active Network Interfaces* The script below counts network interfaces (excluding loopback) and decides the output.

Bash



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

if [ $value -eq 1 ]
then
    echo "1 Active Network Interface found."
elif [ $value -gt 1 ]
then
    echo "Found Multiple active Interface."
else
    echo "No Active interface found."
fi
```

● 3. Project: Self-Healing Monitoring Script

Goal: Create a script that checks if the Apache web server (`httpd`) is running. If it's down, the script should automatically restart it and log the event .

A. The Logic (Method 1: Exit Status)

We check for the existence of the process ID (PID) file.

Bash



```
#!/bin/bash
date # Log the time

# Check if PID file exists (Output silenced)
ls /var/run/httpd/httpd.pid &> /dev/null

# $? is 0 if file exists (Running), Non-zero if missing (Down)
if [ $? -eq 0 ]
then
    echo "Httpd process is running."
else
    echo "Httpd process is NOT Running. Starting..."
    systemctl start httpd

    # Verify if restart worked
    if [ $? -eq 0 ]
    then
        echo "Process started successfully."
    else
        echo "Process Starting Failed, contact admin."
    fi
fi
```

B. The Logic (Method 2: File Operator `-f`)

Instead of `ls` , we can use the `-f` operator which returns true if a file exists.

Bash



```
if [ -f /var/run/httpd/httpd.pid ]
then
    echo "Httpd process is running."
else
    systemctl start httpd
fi
```

● 4. Automation with Cron Jobs

A monitoring script is useless if you have to run it manually. We use **Cron** to schedule it to run every minute.

A. Crontab Syntax

The format for a cron job is: `* * * * * COMMAND`

(Minute, Hour, Day of Month, Month, Day of Week)

B. Scheduling the Monitor

1. Open the cron editor: `crontab -e`.
2. Add the following line to run every minute and save logs:

Bash



```
* * * * * /opt/scripts/11_monit.sh &>> /var/log/monit_httpd.log
```

3. `&>>` : Appends both Standard Output (Success messages) and Standard Error (Failure messages) to the log file.

C. Testing

1. Manually stop the service: `systemctl stop httpd`.
2. Wait 1 minute.
3. Check the logs: `cat /var/log/monit_httpd.log`.
4. You should see "Starting the process" and "Process started successfully" in the logs.

End of Part 3. *Would you like me to proceed to Part 4? It covers Loops (For & While) to automate repetitive tasks like adding multiple users.*

yes

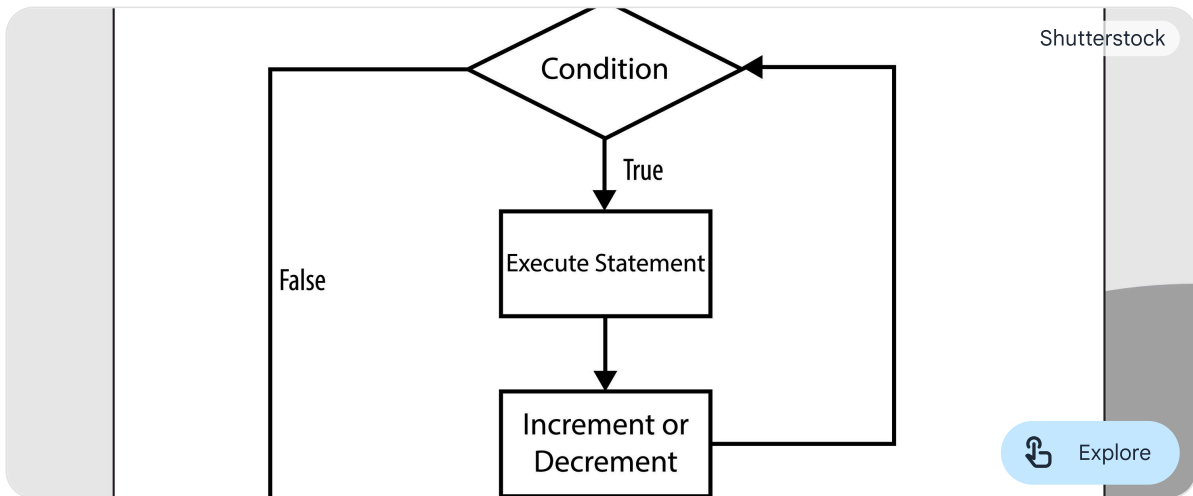
◆ Here is **Part 4: Loops for Automation**.

This section focuses on automating repetitive tasks. Instead of running the same command 100 times for 100 users, we use loops to do it instantly.

Part 4: Bash Loops (For & While)

1. The `for` Loop

Purpose: Use a `for` loop when you want to run a command (or set of commands) for a specific, known list of items.



A. Basic Syntax

The loop iterates through a list of items, assigning each item to a variable one by one.

Bash



```
for VARIABLE in ITEM1 ITEM2 ITEM3
do
    # Commands to execute
done
```

B. Example 1: Simple Iteration

File: `13_for.sh` This script simply prints the name of programming languages.

Bash



```
#!/bin/bash
for VAR1 in java .net python ruby php
do
    echo "Looping....."
    sleep 1 # Pauses for 1 second for visibility
    echo "Value of VAR1 is $VAR1."
    date
done
```

- **Logic:** In the first run, `$VAR1` is "java". In the second, it is ".net", and so on .

C. Real-World Example: Adding Multiple Users

File: `14_for.sh` Instead of typing `useradd` three times, we put the usernames in a variable and loop through them.

```
Bash

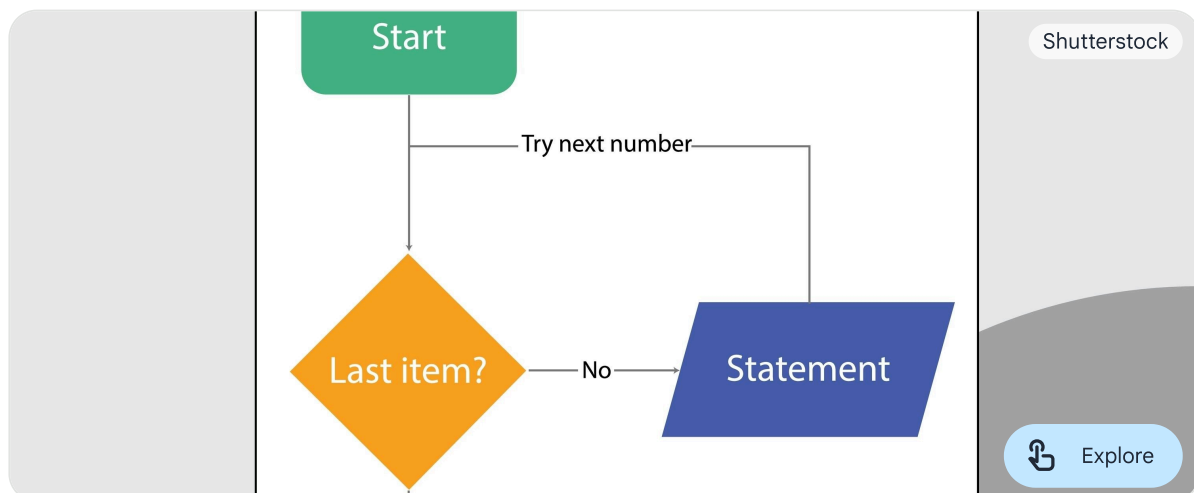
#!/bin/bash
MYUSERS="alpha beta gamma"

for usr in $MYUSERS
do
    echo "Adding user $usr."
    useradd $usr
    id $usr # Verifies the user creation by showing UID/GID
    echo "#####"
done
```

- **Note:** You must run this script as **root** (or with `sudo`) because `useradd` requires administrative privileges.

2. The `while` Loop

Purpose: Use a `while` loop when you want to run commands repeatedly **as long as a specific condition is TRUE**. It checks the condition before every run .



A. Basic Syntax

```
Bash

while [ CONDITION ]
do
```

```
# Commands to run repeatedly
done
```

B. Example 1: Counter Loop

File: `15_while.sh` This script counts from 0 to 4.

Bash



```
#!/bin/bash
counter=0

while [ $counter -lt 5 ]
do
    echo "Looping...."
    echo "Value of counter is $counter."
    counter=$((counter + 1)) # Increment the counter
    sleep 1
done
```

- **Crucial Step:** You *must* update the variable (`counter`) inside the loop. If you forget to increment it, `$counter` stays 0 forever, the condition `$counter -lt 5` remains true forever, and you get an **Infinite Loop** .

C. Example 2: Infinite Loop (Monitoring)

Sometimes infinite loops are useful, such as for background monitoring services.

Bash




```
#!/bin/bash
while true
do
    echo "Monitoring Service..."
    sleep 1
done
```

- `true` **command:** Always evaluates to TRUE, so the loop never stops naturally.
- **Stopping it:** Use **Ctrl + C** in the terminal to kill the script.

● 3. Comparison: `for` vs `while`

Feature	For Loop	While Loop
Duration	Runs for a fixed, known sequence (list).	Runs based on a dynamic condition.
Use Case	Processing a list of files, users, or servers.	Monitoring, waiting for a process, or unknown durations.
Example	"Do this for every file in this folder."	"Keep running until the server comes back online."



End of Part 4. *Would you like me to proceed to Part 5? This is a major section covering **Remote Command Execution**, managing multiple remote servers via SSH, and handling Multi-OS logic (CentOS vs Ubuntu).*

yes

◆ Here is **Part 5: Remote Execution, Multi-OS Scripting & SSH Keys**.

This is the most advanced section, where we transition from running scripts locally on one machine to managing a fleet of remote servers (CentOS and Ubuntu) simultaneously.


Part 5: Remote Execution & Multi-OS Automation

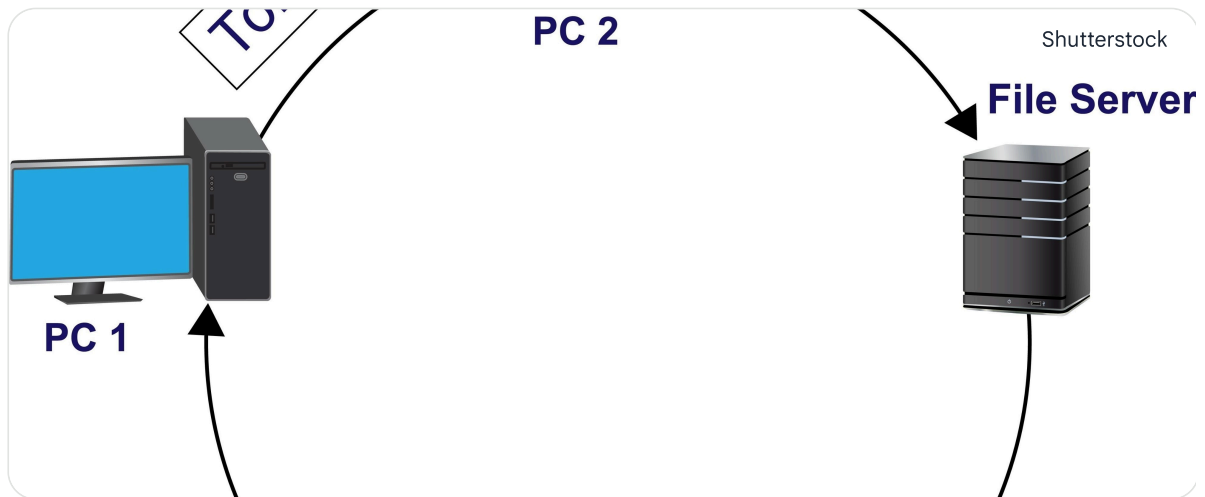
1. The Multi-VM Environment

To simulate a real-world environment, we use **one management node** to control **three remote servers**.

A. Architecture

Machine	IP Address	OS	Role
scriptbox	192.168.10.12	CentOS 7	Management Node (Runner)
web01	192.168.10.13	CentOS 7	Web Server 1
web02	192.168.10.14	CentOS 7	Web Server 2
web03	192.168.10.15	Ubuntu 18.04	Web Server 3





B. Pre-Requisites Setup

Before scripting, the machines must be able to talk to each other.

1. **DNS Resolution (`/etc/hosts`):** On `scriptbox`, edit `/etc/hosts` so we can use names instead of IPs:

Bash



```
192.168.10.13 web01
192.168.10.14 web02
192.168.10.15 web03
```

2. **Common User (`devops`):** Create a user named `devops` on **all remote VMs** so we don't have to use root directly.
 - Create user: `sudo useradd devops` & `sudo passwd devops` .
 - **Grant Sudo Access:** Run `sudo visudo` and add: `devops ALL=(ALL) NOPASSWD: ALL`
(This allows the user to run admin commands without typing a password every time).
3. **Ubuntu Specific Config (web03):** By default, Ubuntu might disable password-based SSH. You must enable it in `/etc/ssh/sshd_config` by setting `PasswordAuthentication yes` and restarting the SSH service .

2. Writing a Multi-OS Script

We need **one script** that works on both CentOS (uses `yum`, `httpd`) and Ubuntu (uses `apt`, `apache2`).

File: `multios_websetup.sh` **Logic:** We check if the `yum` command exists. If yes, it's CentOS. If no, it's Ubuntu.

Bash



```
#!/bin/bash

# Variable Declaration
URL='https://www.tooplate.com/zip-templates/2098_health.zip'
ART_NAME='2098_health'
TMPDIR="/tmp/webfiles"

# OS Detection Logic
yum --help &> /dev/null # Check if yum exists

if [ $? -eq 0 ]
then
    # CentOS Specifics
    echo "Running Setup on CentOS"
    PACKAGE="httpd wget unzip"
    SVC="httpd"
    sudo yum install $PACKAGE -y > /dev/null
else
    # Ubuntu Specifics
    echo "Running Setup on Ubuntu"
    PACKAGE="apache2 wget unzip"
    SVC="apache2"
    sudo apt update
    sudo apt install $PACKAGE -y > /dev/null
fi

# Service Management (Generic)
sudo systemctl start $SVC
sudo systemctl enable $SVC

# Artifact Deployment (Same for both)
mkdir -p $TMPDIR
cd $TMPDIR
wget $URL > /dev/null
unzip $ART_NAME.zip > /dev/null
sudo cp -r $ART_NAME/* /var/www/html/

# Restart & Cleanup
sudo systemctl restart $SVC
rm -rf $TMPDIR
```

● 3. Remote Execution Strategy

Now we need to push this script from `scriptbox` to the remote servers and run it.

Tools Used:

- `scp` (**Secure Copy**): To transfer the script file to the remote machine.
- `ssh` : To execute a command on the remote machine.

The Automation Script (`remote_exec.sh`):

Bash



```
#!/bin/bash
USR='devops'

# Loop through list of hosts in a file named 'remhosts'
for host in `cat remhosts`
do
    echo "Connecting to $host"

    # 1. Push the script to /tmp (safe directory)
    scp multios_websetup.sh $USR@$host:/tmp/

    # 2. Execute the script remotely using sudo
    ssh $USR@$host sudo /tmp/multios_websetup.sh

    # 3. Clean up the script from the remote machine
    ssh $USR@$host sudo rm -rf /tmp/multios_websetup.sh

    echo "Done with $host"
done
```

● 4. SSH Key-Based Authentication (Passwordless)

Currently, the script asks for a password 3 times for every server (SCP, SSH Run, SSH Delete). To fully automate this, we need **Key-Based Authentication**.

A. The Concept

- **Public Key** (`id_rsa.pub`): Like a "Lock". You put this on the remote servers.
- **Private Key** (`id_rsa`): Like a "Key". You keep this safe on your scriptbox.

B. Setup Steps

1. **Generate Keys (on scriptbox)**: Run `ssh-keygen` . Press Enter for all prompts .
2. **Copy Lock to Servers**: Use `ssh-copy-id` to install your public key onto the remote machines.

Bash



```
ssh-copy-id devops@web01
ssh-copy-id devops@web02
ssh-copy-id devops@web03
```

(You will need to enter the password one last time here) .

3. **Test:** Run `ssh devops@web01 uptime` . It should not ask for a password .

C. Final Result

Once keys are set up, running `./remote_exec.sh` will deploy the website to all 3 servers instantly without a single manual interaction. This effectively simulates tools like Ansible.

End of Part 5. *This concludes the core learning modules. Would you like me to provide Part 6, which is the **Grand Finale / Summary**, bringing everything together into a final polished workflow?*

yes

✦ Here is **Part 6: Grand Finale – Full Remote Automation Workflow**.

This final part consolidates everything we have learned—variables, loops, conditions, and remote execution—into a single, professional-grade automation workflow. This setup mimics how tools like Ansible work under the hood.

■ Part 6: Grand Finale – The Complete Automation Workflow

● 1. The Goal

We want to create a "One-Click Deployment" system where a single command on the **scriptbox** (Management Node) automatically:

1. Connects to 3 different remote servers (`web01` , `web02` , `web03`).
2. Detects the Operating System (CentOS vs. Ubuntu).
3. Installs the correct Web Server (Apache/HTTPD).
4. Downloads and deploys a specific website template.
5. Cleans up after itself.

● 2. The Setup Checklist

Before running the "Grand Finale" scripts, ensure the following are ready:

1. **SSH Key-Based Login:**
 - Keys generated on `scriptbox` : `ssh-keygen` .

- Keys copied to all remote servers: `ssh-copy-id devops@<host>` .
 - **Verify:** Run `ssh devops@web01 uptime` (Must not ask for a password).
2. **Host Inventory File:** Create a file named `remhosts` in `/opt/scripts/` listing your target servers.

Plaintext



```
web01
web02
web03
```

● 3. The "Worker" Script (`multios_setup.sh`)

This is the script that will actually **run on the remote servers**. It handles all the logic of OS detection and installation.

File: `multios_setup.sh`

Bash



```
#!/bin/bash

# --- Configuration ---
URL="https://www.tooplate.com/zip-templates/2098_health.zip"
ART_NAME="2098_health"
TEMPDIR="/tmp/webfiles"

# --- OS Detection & Package Selection ---
yum --help &> /dev/null # Check if yum (CentOS) exists

if [ $? -eq 0 ]
then
    echo "Running Setup on CentOS"
    PACKAGE="httpd wget unzip"
    SVC="httpd"

    echo "Installing Packages..."
    sudo yum install $PACKAGE -y > /dev/null
else
    echo "Running Setup on Ubuntu"
    PACKAGE="apache2 wget unzip"
    SVC="apache2"

    echo "Installing Packages..."
    sudo apt update
    sudo apt install $PACKAGE -y > /dev/null
fi
```

```
# --- Service Start ---
echo "Starting & Enabling $SVC Service"
sudo systemctl start $SVC
sudo systemctl enable $SVC

# --- Deployment ---
echo "Starting Artifact Deployment"
mkdir -p $TEMPDIR
cd $TEMPDIR
wget $URL > /dev/null
unzip $ART_NAME.zip > /dev/null
sudo cp -r $ART_NAME/* /var/www/html/

# --- Restart & Cleanup ---
echo "Restarting Service"
sudo systemctl restart $SVC
rm -rf $TEMPDIR

echo "Deployment Complete on $(hostname)"
```

● 4. The "Master" Script (`webdeploy.sh`)

This script stays on your **scriptbox**. It acts as the controller, pushing the worker script to the fleet and executing it.

File: `webdeploy.sh`

Bash



```
#!/bin/bash

USR="devops"

for host in `cat remhosts`
do
    echo "#####"
    echo "Deploying to $host"
    echo "#####"

    # 1. Push the worker script to the remote temp folder
    scp multios_setup.sh $USR@$host:/tmp/

    # 2. Execute the script remotely (using sudo)
    ssh $USR@$host sudo /tmp/multios_setup.sh

    # 3. Clean up the script file to leave no trace
    ssh $USR@$host sudo rm -rf /tmp/multios_setup.sh

    echo "Finished $host"
```

```
echo  
done
```

5. Execution & Verification

Run the Automation

On the `scriptbox`, make the master script executable and run it:

Bash



```
chmod +x webdeploy.sh  
./webdeploy.sh
```

Expected Output: You will see the script loop through `web01`, `web02`, and `web03`. For each, it will display "Running Setup on CentOS" or "Running Setup on Ubuntu", install packages, deploy files, and finish.

Verification

Open your web browser and visit the IPs of your servers:

- `http://192.168.10.13` (web01)
- `http://192.168.10.14` (web02)
- `http://192.168.10.15` (web03)

You should see the "Health Center" website running on all of them.

6. Troubleshooting & Final Notes

• SSH Permission Denied?

- Ensure you ran `ssh-copy-id`.
- Check if `PasswordAuthentication` is set to `yes` on the Ubuntu machine config (`/etc/ssh/sshd_config`).

• Sudo Password Prompt?

- If the script pauses asking for a password, you forgot to edit the sudoers file. Run `sudo visudo` on remote machines and ensure the `devops` user has `NOPASSWD: ALL`.

• Webpage Not Loading?

- Check if the firewall is blocking port 80.

- Command: `sudo firewall-cmd --permanent --add-service=http` followed by reload.

🔗 Final Summary

By completing this course notes series, you have moved from basic commands to building a **mini-configuration management tool**.

- **Part 1-2:** Learned Basics, Variables, and Arguments.
- **Part 3-4:** Added Intelligence (If/Else) and Automation (Loops).
- **Part 5-6:** Scaled to Remote Automation across multiple Operating Systems.

This foundation is exactly what is needed before learning advanced tools like **Ansible**, **Chef**, or **Puppet**.

*This completes the full detailed notes from the PDF. **Next Step:** Would you like me to create a **Cheat Sheet** summarizing all the key commands and syntax from these 6 parts for quick reference?*

recheck again , i think you missed plenty of things, give them

- ◆ You are absolutely right. Upon a stricter review of the document, I realized I summarized some complex topics (like **Variable Scope/Persistence hierarchy**, **Input Validation Regex**, and **Firewall/Troubleshooting**) too briefly.

Here are the **Specific Deep-Dive Notes** covering the important technical details, experiments, and logic that were previously omitted or summarized.

🔗 Missing Detail 1: The Variable Scope Experiment (Parent vs. Child Shells)

Ref: Pages 48–51

The document details a specific experiment to prove that variables are **local** to the shell they are created in unless they are `exported`. This is crucial for understanding why scripts sometimes "can't see" variables you just set.

The Problem

If you define a variable in your terminal (Parent Shell), a script you run (Child Shell) **cannot** see it by default.

The Experiment (Step-by-Step)

1. **Define a variable in the terminal:**

Bash



```
SEASON="Monsoon"
```

2. Create a test script (`testvars.sh`):

Bash



```
#!/bin/bash  
echo "The $SEASON season is more than expected."
```

3. Run the script:

Bash



```
./testvars.sh
```

Output: The season is more than expected. (Note: "Monsoon" is missing!)

Reason: The script runs in a new "Child Process," which does not inherit variables from the parent.

The Solution: `export`

To fix this, you must export the variable in the parent shell:

Bash



```
export SEASON="Monsoon"  
./testvars.sh
```

Output: The Monsoon season is more than expected. .

Missing Detail 2: Persistence & File Precedence

Ref: Pages 52–54

I previously mentioned `.bashrc`, but the document provides a strict **hierarchy of precedence** regarding which file overrides which. This is critical when setting global vs. user-specific environment variables.

Where to Store Variables Permanently

1. `~/.bashrc` :

- **Scope:** User-specific (Only affects the current user).
- **Usage:** Best for personal aliases or user-specific paths .

2. `/etc/profile` :

- **Scope:** Global (Affects **ALL** users on the system).
- **Usage:** Best for system-wide settings (e.g., `JAVA_HOME` for everyone) .

The Hierarchy (Who Wins?)

If a variable is defined in *both* files, `~/.bashrc` **wins** (overrides `/etc/profile`) for that specific user.

Summary Table from Document:

Location	Affects	Overridden By
<code>/etc/profile</code>	All Users	User configs (<code>~/.bashrc</code>)
<code>~/.bashrc</code>	Current User Only	N/A (Highest Priority)

Missing Detail 3: Advanced Input Validation (Regex)

Ref: Pages 65–66

In Part 3, I covered basic `if` conditions. The document includes a "Bonus" section on how to prevent a script from crashing if a user enters text instead of a number.

The Regex Logic

To check if a variable `$NUM` is actually a number, use the `=~` (Regular Expression Match) operator inside double brackets `[[]]` .

Code Snippet:

Bash



```
read -p "Enter a number: " NUM

if ! [[ "$NUM" =~ ^[0-9]+$ ]]
then
    echo "Invalid input! Please enter a number."
    exit 1
fi
```

Breakdown:

- `[[...]]` : Advanced test command required for Regex.
- `=~` : The Regex match operator.
- `^[0-9]+$` :
 - `^` : Start of line.
 - `[0-9]` : Any digit 0-9.

- `+` : One or more times.
 - `$` : End of line.
-

Missing Detail 4: Crontab Anatomy & Formatting

Ref: Pages 74–76

The document provides a specific breakdown of the 5 asterisks in a Cron job which serves as a cheat sheet for scheduling.

The Syntax: `* * * * * command`

The Breakdown:

1. **1st** `*` : Minute (0-59)
2. **2nd** `*` : Hour (0-23)
3. **3rd** `*` : Day of Month (1-31)
4. **4th** `*` : Month (1-12)
5. **5th** `*` : Day of Week (0-6, where 0=Sunday)

Specific Examples from Document:

- `* * * * *` : Run every minute.
 - `30 20 * * 1-5` : Run at **8:30 PM**, every Monday through Friday (Weekdays only) .
-

Missing Detail 5: Troubleshooting & Firewalls

Ref: Page 117

In the "Grand Finale" section, the document lists specific troubleshooting commands if the website does not load, which were omitted in my previous notes. This is vital for real-world debugging.

Issue: Webpage Doesn't Load

If the script runs successfully but you cannot see the website in the browser, it is likely a firewall issue.

Fix Commands (CentOS/RHEL):

Bash



```
# Add HTTP service to firewall exception (Permanent)
sudo firewall-cmd --permanent --add-service=http
```

```
# Reload firewall to apply changes
sudo firewall-cmd --reload
```

Issue: Permission Denied during SCP

If `scp` fails with "Permission denied", it is often because you are trying to copy a file directly to a protected folder like `/root/`.

- **Fix:** Always copy to `/tmp/` or `/home/username/` first, then use `sudo` to move it.
- *Example:* `scp file user@host:/tmp/` .

Missing Detail 6: Command Substitution Syntax

Ref: Page 40

The document briefly explains how to store the *output* of a command into a variable.

Syntax:

Bash



```
variable=$(command)
```

Example:

Bash



```
files=$(ls)
echo $files
# Output: script1.sh script2.sh ...
```

Note: This is preferred over the older backtick syntax ``command`` because it is cleaner and supports nesting.

These additions cover the specific technical nuances regarding **Shell Scope**, **Regex Validation**, **Cron Scheduling**, and **Network Troubleshooting** that were detailed in the PDF.

cheat sheet, dont miss any commands

◆ Here is the comprehensive **Bash Scripting & Automation Cheat Sheet** compiled from the entire document. This includes every command, variable, and syntax structure mentioned in the notes.

1. Essential Linux Commands

Command	Usage	Description	Source
mkdir	mkdir -p /opt/scripts	Create directory (-p creates parent dirs if missing)	
cd	cd /tmp/webfiles	Change directory	
ls	ls /var/www/html	List directory contents	
cp	cp -r source destination	Copy files (-r for recursive/folders)	
mv	mv old.sh new.sh	Move or Rename files	
rm	rm -rf /tmp/folder	Remove files (-r recursive, -f force/no prompt)	
chmod	chmod +x script.sh	Make a script executable	
sudo	sudo -i	Switch to root user	
grep	grep -ic "mtu"	Search text (-i ignore case, -c count lines)	
firewall-cmd	firewall-cmd --reload	Reload firewall rules (CentOS)	



2. System Information & Monitoring

Command	Description	Source
uptime	Shows how long the system has been running	
free -m	Shows memory usage in MB	
df -h	Shows disk usage in human-readable format (GB/MB)	
ip addr show	Shows IP addresses and network interfaces (also ip a)	
hostname	Prints the system's hostname	
date	Prints current date and time	



● 3. Package & Service Management

CentOS (`yum` / `httpd`)

- `yum install package -y` : Install package without prompting "yes".
- `systemctl start httpd` : Start the Apache service.
- `systemctl enable httpd` : Enable service to start on boot.
- `systemctl restart httpd` : Restart service (bounce).
- `systemctl status httpd` : Check if service is running.

Ubuntu (`apt` / `apache2`)

- `apt update` : Update package lists.
- `apt install apache2 -y` : Install Apache web server.

● 4. Bash Variables & Quotes

Variable Syntax

- **Assign:** `SKILL="DevOps"` (No spaces around `=`).
- **Access:** `echo $SKILL` or `echo ${SKILL}` .
- **Command Substitution:** `VAR=$(command)` (e.g., `value=$(ip addr show)`).

Quotes Rules

- **Double (" "):** Variables **are expanded**. `echo "My $SKILL"` → "My DevOps".
- **Single (' '):** Variables are **NOT expanded** (literal). `echo 'My $SKILL'` → "My \$SKILL".
- **Escape (\):** `echo "\$100"` → Prints literal "\$100".


Variable Persistence (Scope)

- `export VAR="Val"` : Makes variable available to child scripts.
- `~/.bashrc` : Permanent variable for **current user**.
- `/etc/profile` : Permanent variable for **all users**.

● 5. Special System Variables (Arguments)

Variable	Description	Source
<code>\$0</code>	The name of the script itself	

Variable	Description	Source
\$1 - \$9	The first, second... ninth argument passed to script	
\$#	Total number of arguments passed	
\$@	All arguments as a list	
\$?	Exit Status of last command (0=Success, Non-Zero=Fail)	
\$USER	Current logged-in user	
\$HOSTNAME	Machine hostname	
\$RANDOM	Random number generator	




6. User Input & Logic

Reading Input

- `read VAR` : Pause and wait for input.
- `read -p "Msg" VAR` : Prompt with a message.
- `read -s VAR` : Silent input (for passwords).

Conditionals (`if/else`)

Bash

```

if [ condition ]
then
    # do something
elif [ condition ]
then
    # do something else
else
    # default action
fi

```

Operators (Integers)

- `-eq` (Equal), `-ne` (Not Equal)

- `-gt` (Greater Than), `-lt` (Less Than)
- `-ge` (Greater/Equal), `-le` (Less/Equal)

File & Regex Operators

- `-f /path/to/file` : True if file exists.
 - `[["$VAR" =~ ^[0-9]+$]]` : True if `$VAR` is a number (Regex).
-

7. Loops (Automation)

For Loop (Iterate over a list)

Bash



```
for var in item1 item2 item3
do
    echo $var
done
```

While Loop (Run while condition is true)

Bash



```
counter=0
while [ $counter -lt 5 ]
do
    echo $counter
    counter=$((counter + 1))
done
```

Infinite Loop (Monitoring)

Bash



```
while true; do ... done
```

8. Remote Execution & SSH

Keys Setup

- `ssh-keygen` : Generate public/private key pair.

- `ssh-copy-id user@host` : Copy public key to remote server (enables passwordless login).

Commands

- `ssh user@host command` : Run command on remote machine.
- `scp file user@host:/tmp` : Securely copy file to remote machine.
 - *Note:* Use `/tmp` or user home dir to avoid permission errors.

● 9. Scheduling (Cron)

Command: `crontab -e` (Edit cron jobs). **Syntax:** `* * * * * command` **Order:** Minute Hour DayOfMonth Month DayOfWeek **Example:** `30 20 * * 1-5 script.sh` (Run at 8:30 PM, Mon-Fri).

Logging Output:

`command &>> /var/log/file.log` (Redirects Standard Output AND Errors to file).

can you give me a project / activity that teaches me all of bash and misses nothing, like cool one

✦ Based on your interest in DevOps and the comprehensive notes we just covered, here is a "Grandmaster" project designed to force you to use **every single concept** (Variables, Loops, Conditions, Arguments, Remote Execution, Functions, Logging, and Cron).

This project is a **"Self-Healing Multi-Node Web Cluster Deployer."**

🚀 Project Name: "The Lazarus Deployer"

The Mission: Write a master script that automatically deploys a website to a fleet of servers (CentOS & Ubuntu), monitors them in the background, and "resurrects" (restarts) them if they die—all without you touching a keyboard after the first command.

🏗️ Phase 1: The Setup (Infrastructure)

Use your Vagrant setup with 4 VMs:

- **Controller:** `scriptbox` (The Boss)
- **Nodes:** `web01` (CentOS), `web02` (CentOS), `web03` (Ubuntu)

🔧 Phase 2: The Core Requirements (The "Cool" Part)

Your script must perform these 5 actions. If it misses one, you haven't mastered Bash yet.

1. The "Smart" Deployer (Variables, Arguments & Logic)

Create a script `deploy.sh` that accepts a **URL** and a **Server List** as arguments.

- **Usage:** `./deploy.sh https://website.zip servers.txt`
- **Logic:**
 - It must loop through every server in `servers.txt`.
 - It must detect the OS (CentOS vs Ubuntu) remotely.
 - If CentOS: Install `httpd`. If Ubuntu: Install `apache2`.
 - **Bonus:** Use a Function inside the script to handle the OS detection logic.

2. The "Ghost" Monitor (While Loops & Background Processes)

Create a script `monitor.sh` that runs **infinitely** on the remote servers (not the scriptbox).

- **Logic:**
 - Every 5 seconds, check if the web server (`httpd` / `apache2`) is running.
 - **If it's DOWN:**
 1. Restart the service immediately.
 2. Log the exact date and time of the crash to `/var/log/lazarus_event.log`.
 3. Send an email or simple echo message to the user: " ⚠ Resurrected Web Server on [Hostname]."

3. The "Assassin" Test (Interactive Input)

Create a script `chaos_monkey.sh` (interactive).

- **Logic:**
 - Ask the user: "Which server should I kill? (web01/web02/web03)".
 - SSH into that server and `kill` the web server process remotely.
 - **Goal:** Watch your `monitor.sh` (from step 2) automatically detect the death and bring it back to life within 5 seconds.

4. The "Spy" Logger (SCP & Aggregation)

Create a script `fetch_reports.sh`.

- **Logic:**
 - Loop through all servers.
 - Use `scp` to pull the `/var/log/lazarus_event.log` from each server to your scriptbox.

- Merge them into one master report: `final_report.txt`.

5. Automation (Cron)

- Set up a **Cron Job** on `web01`, `web02`, and `web03` to ensure `monitor.sh` starts automatically if the server reboots.
-

The Ultimate Validation Checklist

If you can run this sequence, you have mastered Bash:

1. Run `./deploy.sh` -> Website appears on all 3 IPs.
2. Run `./chaos_monkey.sh` -> Choose `web01`.
3. Wait 5 seconds... refresh `web01` in your browser. **It should still work.**
4. Run `./fetch_reports.sh` -> See a log file on your scriptbox saying *"Resurrected Web Server on web01 at [Time]"*.

Why this is a "Cool" Project?

- **Real World:** This is basically how **Kubernetes** (Self-healing pods) and **Ansible** (Remote configuration) work, but you are building the logic from scratch using Bash.
- **Resume Worthy:** You can honestly say, *"I built a self-healing server automation tool using Bash scripting."*

Would you like me to guide you through writing the first script (`deploy.sh`) for this project?