**Live Session I**

Basic scripts (part 1)

Refs.

<https://www.warp.dev/> Awesome terminal!

<https://gitforwindows.org/>

<https://phoenixnap.com/kb/linux-commands-cheat-sheet>

<https://linuxsimply.com/bash-scripting-tutorial/basics/examples/>

**Example 1**

#!/bin/bash

echo “Hello World”

#!/bin/bash

name=Tom

age=12

echo "$name's age is $age"

Environment variables in Bash are dynamic named values that define the operating environment for the shell session and its child processes. There are a few environment variables that contain information about the environment in which the shell is running. These variables are special and are accessible to all scripts within the environment. Some of the most common environment variables are:

**HOME**: The home directory of the current user.

**USER**: The username of the current user.

**PWD**: The current working directory.

**PATH**: A colon-separated list of directories in which the shell looks for executable files.

**SHELL**: The path to the current shell executable.

**Example 2**

#!/bin/bash

echo $HOME # Prints the home directory of the current user

echo $PATH # Prints the directories in which the shell searches for executable files

**Taking User Input in a Variable**

The [**read command**](https://linuxsimply.com/bash-scripting-tutorial/input-output/input/read-input/) is used to take user input. Once you use **-p,**the command is enabled to prompt a message to the user along with taking input. Later, you can use **echo $VARIABLE\_NAME** to display the user input on the screen.

To take user input in a variable, use the following bash script:

**Example 3**

#!/bin/bash

read -p "Enter a number:" num

echo "You entered: $num"

**Display Output in Terminal or Saving in a File**

Let’s see how to perform an arithmetic operation using variables and display the resulting output in the terminal or save it in a file. To perform any arithmetic calculation in Bash use **$(())** syntax. See the script below for more clarification about its usage:

#!/bin/bash

read -p "Enter a number: " num1

read -p "Enter another number: " num2

add=$((num1 + num2)) # NO SPACE WHEN ASSIGNING!

echo "Addition of numbers: $add"

Advanced Scripts (part 2 – using sed / awk)

First a breakdown of some course terms:

*Packer*

Packer is a tool for creating identical machine images for multiple platforms from a single source configuration. Its key features include:

It can create images for various platforms like VirtualBox, AWS, Azure, etc. from a single configuration file.

It automates the process of creating a base image by running provisioning scripts on a "vanilla" OS distribution.

It's often used to create standardized VM images before distributing them to users or deploying to cloud platforms.

*Vagrant*

Vagrant is a tool for building and managing virtual machine environments. Its main characteristics are:

It's primarily aimed at developers, with the slogan "Development Environments Made Easy".

It acts as a wrapper around virtualization software like VirtualBox, Hyper-V, or even cloud providers.

It allows easy creation, configuration, and destruction of development environments using simple commands.

It can mount directories from the host system to the guest system, making it easy to work on code

Key Differences and Relationships between *Packer* and *Vagrant*

Purpose:

Packer is for creating machine images.

Vagrant is for managing development environments.

Workflow:

Packer is typically used earlier in the process to create base images.

Vagrant uses these images (called "boxes") to spin up development environments.

Integration:

Packer can create Vagrant boxes, which Vagrant can then use.

You can use Packer to create custom Vagrant boxes for your development environment.

Usage:

Packer is often used in production workflows to create standardized images.

Vagrant is primarily used by developers for local development environments.

In a typical workflow, you might use Packer to create a custom Vagrant box, and then use Vagrant to spin up and manage development environments based on that box. This allows for consistent development environments across a team while providing the flexibility to customize as needed.

**Example 1**

Show simple example of a packer file with data and a **sed/awk** command to change the data within to some other feasible response

First, create a basic Packer template file named template.json:

**{**

**"variables": {**

**"region": "us-east-1",**

**"instance\_type": "t2.micro"**

**},**

**"builders": [**

**{**

**"type": "amazon-ebs",**

**"region": "{{user `region`}}",**

**"instance\_type": "{{user `instance\_type`}}",**

**"source\_ami": "ami-0c55b159cbfafe1f0",**

**"ssh\_username": "ubuntu",**

**"ami\_name": "packer-example {{timestamp}}"**

**}**

**],**

**"provisioners": [**

**{**

**"type": "shell",**

**"script": "scripts/install.sh"**

**}**

**]**

**}**

Packer Configuration breakdown

## Variables

"variables": {

"region": "**us-east-1**",

"instance\_type": "**t2.micro**"

}

This section defines two variables:

- "region" is set to "us-east-1" (an AWS region)

- "instance\_type" is set to "t2.micro" (an AWS EC2 instance type)

**These variables can be referenced later in the configuration using the syntax `{{user `variable\_name`}}`.**

## Builders

"builders": [

{

"type": "amazon-ebs",

"region": "{{user `region`}}",

"instance\_type": "{{user `instance\_type`}}",

"source\_ami": "ami-0c55b159cbfafe1f0",

"ssh\_username": "ubuntu",

"ami\_name": "packer-example {{timestamp}}"

}

]

The above section defines a single builder of type "amazon-ebs" (Amazon Elastic Block Store):

- It uses the previously defined variables for region and instance type

- Specifies a source AMI (Amazon Machine Image) ID

- Sets the SSH username to "ubuntu"

- Names the resulting AMI "packer-example" followed by a timestamp

## Provisioners

"provisioners": [

{

"type": "shell",

"script": "scripts/install.sh"

}

]

The above section defines a single provisioner:

- It's of type "shell", which means it will run a shell script

- The script to be run is located at "scripts/install.sh"

This configuration will create an Amazon EBS-backed AMI in the specified region, using the specified instance type and source AMI. It will then run the install.sh script on the instance before creating the final image

**Why user templating of placeholders in packer?**

Using template placeholders in Packer, such as "region": "**{{user region}}**", provides several advantages:

Parameterization: Placeholders allow you to parameterize your templates, which means you can define variables that can be set at runtime. This makes your templates more flexible and reusable across different environments or configurations without needing to modify the template itself.

Separation of Concerns: By using placeholders, you can separate configuration data from the logic of the template. This allows you to manage configuration values independently, making it easier to update or change them without altering the core template.

Portability: Placeholders help in keeping sensitive information, such as API keys or passwords, out of the template! These values can be injected at runtime from secure sources like environment variables or secret management tools, enhancing the security and portability of the template.

Ease of Updates: When using placeholders, you can update the values of variables easily using command-line arguments, environment variables, or files without needing to edit the template file directly. This is particularly useful in automated build systems where different configurations might be needed for different builds.

Overall, using placeholders in Packer templates enhances flexibility, security, and maintainability, making it easier to manage infrastructure as code.

---

Suppose you want to change the ‘region’ and ‘instance\_type’ values in the Packer file to different values. You can use ‘sed’ to achieve this!

\*\*Command:\*\*

**sed -i.bak '/"variables": {/,/}/s/"region": "[^"]\*"/"region": "'"${NEW\_REGION}"'"/' "$PACKER\_FILE"**

**sed -i.bak '/"variables": {/,/}/s/"instance\_type": "[^"]\*"/"instance\_type": "'"${NEW\_INSTANCE\_TYPE}"'"/' "$PACKER\_FILE"**

\*\*Explanation:\*\*

-i: Edits the file in place.

- s/old/new/: Substitutes ‘old’ with ‘new’.

After running commands that follows, the ‘template.json’ file will have the ‘region’ set to ‘us-west-2’ and ‘instance\_type’ set to ‘t2.large’.

**#!/bin/bash**

**# Define the Packer file to modify**

**PACKER\_FILE="template.json"**

**# Define new values for the variables**

**NEW\_REGION="us-west-2"**

**NEW\_INSTANCE\_TYPE="t2.large"**

**# Check if the file exists**

**if [[ ! -f "$PACKER\_FILE" ]]; then**

**echo "File $PACKER\_FILE does not exist."**

**exit 1**

**fi**

**# Using sed to update the values in the JSON file**

**echo "Updating region and instance type in $PACKER\_FILE..."**

**# Update the value of the `region` variable in the JSON file**

**sed -i.bak '/"variables": {/,/}/s/"region": "[^"]\*"/"region": "'"${NEW\_REGION}"'"/' "$PACKER\_FILE"**

**# Update the value of the `instance\_type` variable in the JSON filec**

**sed -i.bak '/"variables": {/,/}/s/"instance\_type": "[^"]\*"/"instance\_type": "'"${NEW\_INSTANCE\_TYPE}"'"/' "$PACKER\_FILE"**

**echo "Update complete."**

**# Clean up the backup file created by sed**

**rm -f "${PACKER\_FILE}.bak"**

\*Note, the above script ensures that it doesn't rely on specific existing values and can dynamically replace the placeholders with the new ones provided by the script's variables.

\*\*Make sure to run script

do chmod to execute shell

./ to run

Also

Show history / Route history

Detailed breakdown of shell script!

if statement provided in the shell script

### Statement:

```bash

if [[ ! -f "$PACKER\_FILE" ]]; then

echo "File $PACKER\_FILE does not exist."

exit 1

fi

```

Shows…

1. \*\*`if` Statement\*\*:

- The `if` statement is used to evaluate a condition. If the condition is true, the commands inside the `then` block will execute. If the condition is false, the script will skip to the next section after `fi`.

2. \*\*`[[ ... ]]` (Double Square Brackets)\*\*:

- `[[ ... ]]` is a test command in bash that is used for conditional expressions. It is more powerful and versatile than the single square brackets `[ ... ]`.

- The double brackets allow for advanced string comparison, pattern matching, and logical operations.

3. \*\*`-f` Option\*\*:

- The `-f` option is used to check if a file exists and is a regular file (not a directory or a special file like a socket or symlink).

- It returns `true` if the file exists and is a regular file, and `false` otherwise.

4. \*\*`!` (Logical NOT)\*\*:

- The exclamation mark `!` is a logical NOT operator. It inverts the result of the condition that follows it.

- In this context, `! -f` means "not a regular file" or "file does not exist."

5. \*\*`"$PACKER\_FILE"`\*\*:

- This is a variable that holds the name of the file you want to check. The `$` symbol is used to dereference the variable, meaning it gets replaced by the value stored in `PACKER\_FILE`.

- The double quotes around the variable ensure that the script handles file names with spaces or special characters correctly.

6. \*\*`then` Keyword\*\*:

- Thus the If the condition in the `if` statement is true (in this case, if the file does not exist), the commands following `then` will execute.

7. \*\*`echo "File $PACKER\_FILE does not exist."`\*\*:

- The `echo` command prints the message to the terminal. In this case, it notifies the user that the file specified by the `PACKER\_FILE` variable does not exist.

8. \*\*`exit 1`\*\*:

- The `exit` command terminates the script. The `1` passed to `exit` indicates that the script terminated with an error. By convention, a non-zero exit status signifies an error.

9. \*\*`fi`\*\*:

- This marks the end of the `if` block. Everything between `if` and `fi` is part of the conditional execution.

### Summary:

This `if` statement checks whether a file specified by the `PACKER\_FILE` variable exists and is a regular file. If the file does not exist, the script prints an error message and exits with a status code of `1` to indicate a failure. If the file does exist, the script continues executing the commands after the `fi`.

----

For the sed lines: we make use of a range expression /"variables": {/,/}/ to limit the sed operations to only within the *variables* section of the JSON file.

Within this range, it performs the substitution for region and instance\_type.

The substitution pattern "[^"]\*" matches the value within quotes, *ensuring* we only replace the value and not any surrounding syntax.

This approach should modify only the values in the variables section, leaving the placeholders in the builders section intact.

---

For the Backup File Creation: The -i.bak option creates a backup of the original file before making changes. This is useful to avoid accidental data loss. After updating, the script deletes the backup.

Note, for Cross-Platform Compatibility: The use of -i.bak makes the script compatible with both macOS and Linux.

**Ref:**

**Regex syntax!**

[**https://pkg.go.dev/regexp/syntax#hdr-Syntax**](https://pkg.go.dev/regexp/syntax#hdr-Syntax)

Ending Summary

- \*\*`sed`\*\*: Use for in-place text replacement within files.

- \*\*`awk`\*\*: Use for more complex text processing and to create new files with modifications.

Both tools are powerful for scripting and automating tasks like modifying configuration files in your CI/CD pipelines.