Live Session I

Git coms

git remote set-url origin <https://github.com/igitiit/356-556_LiveSessions>

git remote get-url origin

git --no-pager remote

if [ ! -f .gitignore ]; then echo '\*~\*' > .gitignore; else echo '\*~\*' >> .gitignore; fi

git add .gitignore

git commit -m "Add .gitignore file to ignore files with ~ in their names"

git push -u origin main

file removal

git rm '~$M0 356-556 Live Session I FA A 24.docx'

git commit -m "Remove file that begins with ~"

git push -u origin main

or

git rm --cached '~$M0 356-556 Live Session I FA A 24.docx'

git commit -m "Remove all tracked files beginning with ~"

git push -u origin main

to remove all files

git ls-files | grep '^~' | xargs git rm --cached

git commit -m "Remove all tracked files beginning with ~"

git push -u origin main

----

To connect a local git repository to the remote repository at "https://github.com/igitiit/356-556\_LiveSessions" and push your changes, you can follow these steps:

1. Initialize the local repository (if you haven't already):

**git init**

zsh

2. Add the remote repository:

**git remote add origin https://github.com/igitiit/356-556\_LiveSessions**

zsh

3. Add your files to the staging area (if you haven't already):

**git add .**

zsh

4. Commit your changes:

**git commit -m "Your commit message"**

zsh

5. Push your changes to the remote repository:

**git push -u origin main**

zsh

Make sure you replace `"Your commit message"` with an appropriate commit message describing your changes. If your default branch is not `master`, replace `master` with the name of your default branch (e.g., `main`).

Basic scripts

Refs

<https://www.warp.dev/>

<https://gitforwindows.org/>

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

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

#!/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.

#!/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:

#!/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

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 Relationship

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.

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

Certainly! Here’s a simple example of a Packer template and how you can use `sed` or `awk` to modify the data within it.

### Example Packer File

Let's create a basic Packer template file named `template.json`:

```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"**

**}**

**]**

**}**

```

This JSON file appears to be a configuration for Packer, a tool used to create machine images. Let's break it down section by section:

## Variables

```json

"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

```json

"builders": [

{

"type": "amazon-ebs",

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

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

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

"ssh\_username": "ubuntu",

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

}

]

```

This 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

```json

"provisioners": [

{

"type": "shell",

"script": "scripts/install.sh"

}

]

```

This 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:\*\*

```bash

sed -i 's/"region": "us-east-1"/"region": "us-west-2"/' template.json

sed -i 's/"instance\_type": "t2.micro"/"instance\_type": "t2.large"/' template.json

```

\*\*Explanation:\*\*

- `-i`: Edits the file in place.

- `s/old/new/`: Substitutes `old` with `new`.

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

### Using `awk` to Modify the Packer File

While `awk` is less commonly used for direct file edits compared to `sed`, you can use it to create a modified version of the file. For example:

\*\*Command:\*\*

```bash

awk '

/"region": "us-east-1"/ { sub(/"region": "us-east-1"/, "\"region\": \"us-west-2\"") }

/"instance\_type": "t2.micro"/ { sub(/"instance\_type": "t2.micro"/, "\"instance\_type\": \"t2.large\"") }

{ print }

' template.json > modified\_template.json

```

\*\*Explanation:\*\*

- `sub(pattern, replacement)`: Replaces the first occurrence of `pattern` with `replacement`.

- `{ print }`: Prints each line after modifications.

This command will create a new file named `modified\_template.json` with the updated values.

### 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.

2. Certainly! Here’s a simple example of a Packer template and how you can use `sed` or `awk` to modify the data within it.

### Example Packer File

Let's create a basic Packer template file named `template.json`:

```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"

}

]

}

```

### Using `sed` to Modify the Packer File

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:\*\*

```bash

sed -i 's/"region": "us-east-1"/"region": "us-west-2"/' template.json

sed -i 's/"instance\_type": "t2.micro"/"instance\_type": "t2.large"/' template.json

```

\*\*Explanation:\*\*

- `-i`: Edits the file in place.

- `s/old/new/`: Substitutes `old` with `new`.

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

### Using `awk` to Modify the Packer File

While `awk` is less commonly used for direct file edits compared to `sed`, you can use it to create a modified version of the file. For example:

\*\*Command:\*\*

```bash

awk '

/"region": "us-east-1"/ { sub(/"region": "us-east-1"/, "\"region\": \"us-west-2\"") }

/"instance\_type": "t2.micro"/ { sub(/"instance\_type": "t2.micro"/, "\"instance\_type\": \"t2.large\"") }

{ print }

' template.json > modified\_template.json

```

\*\*Explanation:\*\*

- `sub(pattern, replacement)`: Replaces the first occurrence of `pattern` with `replacement`.

- `{ print }`: Prints each line after modifications.

This command will create a new file named `modified\_template.json` with the updated values.

### 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.

Above revised to retain template placeholders

#!/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 "s/\"region\": \"us-east-1\"/\"region\": \"${NEW\_REGION}\"/" "$PACKER\_FILE"

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

sed -i "s/\"instance\_type\": \"t2.micro\"/\"instance\_type\": \"${NEW\_INSTANCE\_TYPE}\"/" "$PACKER\_FILE"

echo "Update complete."

Certainly! Here is the breakdown of the `sed` command in plain text:

---

### Breakdown of the `sed` Command

```bash

sed -i "s/\"region\": \"us-east-1\"/\"region\": \"$NEW\_REGION\"/" "$PACKER\_FILE"

```

1. \*\*`sed`\*\*: The command-line utility used for stream editing. It is commonly used for text transformations.

2. \*\*`-i`\*\*: This option tells `sed` to edit files in place. It modifies the file directly instead of sending the output to standard output.

3. \*\*`"s/old/new/"`\*\*: This is the substitution command syntax in `sed`:

- \*\*`s`\*\*: Stands for "substitute".

- \*\*`old`\*\*: The pattern to search for.

- \*\*`new`\*\*: The text to replace the `old` pattern with.

- \*\*`/`\*\*: Delimiters separating the parts of the substitution command.

4. \*\*`\"region\": \"us-east-1\"`\*\*: This is the `old` pattern that `sed` will search for in the file. The backslashes (`\`) are used to escape the double quotes (`"`), so they are treated as literal characters.

5. \*\*`\"region\": \"$NEW\_REGION\"`\*\*: This is the `new` text that will replace the `old` pattern:

- \*\*`\"region\": \"`\*\*: Part of the new text that matches the `region` key in the JSON file.

- \*\*`$NEW\_REGION`\*\*: This is a shell variable that holds the new value (e.g., `us-west-2`).

- \*\*`\"`\*\*: Closing double quote for the `region` value.

6. \*\*`"$PACKER\_FILE"`\*\*: This is the filename to be processed. It contains the path to the file where the substitution will occur.

### What Happens When the Command Runs

1. \*\*Search for Pattern\*\*: `sed` searches through the specified file (`$PACKER\_FILE`) for lines containing the exact text `"region": "us-east-1"`.

2. \*\*Replace Pattern\*\*: When it finds a match, `sed` replaces that text with `"region": "$NEW\_REGION"`, where `$NEW\_REGION` is replaced with the value of the `NEW\_REGION` variable.

3. \*\*In-Place Editing\*\*: The `-i` option means that the change is made directly in the file rather than producing modified output to the console or another file.

### Example

If `$NEW\_REGION` is set to `us-west-2`, and `template.json` initially contains:

```json

{

"variables": {

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

"instance\_type": "t2.micro"

}

}

```

After running the command, `template.json` will be updated to:

```json

{

"variables": {

"region": "us-west-2",

"instance\_type": "t2.micro"

}

}

```

The `region` value is changed to `us-west-2` while the `instance\_type` remains unchanged.

difference between these lines that follow particularly the use of /\ vs / for "region"

sed -i 's/"region": "us-east-1"/"region": "us-west-2"/' template.json

sed -i "s/\"region\": \"us-east-1\"/\"region\": \"${NEW\_REGION}\"/" "$PACKER\_FILE"

Let's break down the differences between the two `sed` commands:

### `sed` Commands

#### Command 1: Single Quotes

```bash

sed -i 's/"region": "us-east-1"/"region": "us-west-2"/' template.json

```

- \*\*Single Quotes (`'`)\*\*:

- The entire pattern and replacement string is enclosed in single quotes.

- Single quotes in shell scripting prevent the shell from interpreting special characters within them. Everything inside single quotes is treated as literal text.

- This means that the double quotes and the content within them are treated as literal characters. No variable substitution or special interpretation happens within single quotes.

- \*\*Usage\*\*:

- Directly replaces `"region": "us-east-1"` with `"region": "us-west-2"` in `template.json`.

- No shell variables or special characters are involved in this example.

#### Command 2: Double Quotes and Escaping

```bash

sed -i "s/\"region\": \"us-east-1\"/\"region\": \"${NEW\_REGION}\"/" "$PACKER\_FILE"

```

- \*\*Double Quotes (`"`)\*\*:

- The pattern and replacement string are enclosed in double quotes.

- Double quotes allow for variable substitution and special character interpretation by the shell. The shell will expand variables and interpret escape sequences inside double quotes.

- \*\*Escaping (`\`)\*\*:

- The backslashes (`\`) are used to escape the double quotes (`"`). This ensures that the quotes are treated as part of the search and replace pattern rather than as delimiters for the string.

- \*\*Variable Substitution\*\*:

- `${NEW\_REGION}` is a shell variable that gets substituted with its value before `sed` processes the command. This allows dynamic replacement of the value based on the variable’s content.

- \*\*Usage\*\*:

- Replaces `"region": "us-east-1"` with `"region": "VALUE\_OF\_NEW\_REGION"` in `$PACKER\_FILE`, where `VALUE\_OF\_NEW\_REGION` is the value of the `NEW\_REGION` variable.

### Summary

- \*\*Single Quotes (`'`)\*\*:

- Used when you want to keep everything literal.

- No variable substitution or special character interpretation by the shell.

- \*\*Double Quotes (`"`)\*\*:

- Used when you want to allow for variable substitution or when you need to escape special characters.

- Shell variables and escape sequences are processed before the command is executed.

### Example

Given `NEW\_REGION="us-west-2"`:

- \*\*Single Quotes Example\*\*:

```bash

sed -i 's/"region": "us-east-1"/"region": "us-west-2"/' template.json

```

This will replace `"region": "us-east-1"` with `"region": "us-west-2"` directly, regardless of any variables.

- \*\*Double Quotes Example\*\*:

```bash

sed -i "s/\"region\": \"us-east-1\"/\"region\": \"${NEW\_REGION}\"/" "$PACKER\_FILE"

```

This will replace `"region": "us-east-1"` with `"region": "us-west-2"` (assuming `NEW\_REGION` is `us-west-2`), allowing dynamic substitution of the variable value.

Using double quotes with escaping and variables is particularly useful when you need to dynamically modify file contents based on variable values in your scripts.

so the \ escape character ensures the escape of the double quotes ". doesn't the / do the same here?

In `sed` substitution commands, the `/` character is used as a delimiter to separate the `s/old/new/` parts of the command, and it does not escape characters. The `\` escape character is necessary for escaping special characters, including double quotes, within the pattern and replacement strings.

Here's a more detailed breakdown:

### Role of `/` in `sed`

- \*\*Delimiter\*\*: In the `sed` substitution command (`s/old/new/`), the `/` character serves as the delimiter. It separates the parts of the substitution command:

- `s` is the substitution command.

- `old` is the pattern to search for.

- `new` is the text to replace the pattern with.

- The final `/` ends the command.

### Role of `\` (Escape Character)

- \*\*Escaping Special Characters\*\*: The `\` character is used to escape special characters, such as double quotes (`"`), within the `sed` command. This allows you to include these characters in the pattern or replacement text without confusing `sed` or the shell.

### Example Breakdown

1. \*\*Without Escaping\*\*:

```bash

sed -i "s/"region": "us-east-1"/"region": "us-west-2"/" file.json

```

- This would cause a syntax error because the shell and `sed` would misinterpret the double quotes as delimiters rather than part of the pattern.

So just use single outer quotes to avoid error

sed -i 's/"region": "us-east-1"/"region": "us-west-2"/' file.json

2. \*\*With Escaping\*\*:

```bash

sed -i "s/\"region\": \"us-east-1\"/\"region\": \"us-west-2\"/" file.json

```

- \*\*`\"`\*\*: The backslashes are used to escape the double quotes inside the double-quoted string. This tells `sed` to treat `"` as a literal character in the pattern and replacement.

### Why Escaping is Necessary

- \*\*Inside Double Quotes\*\*: When using double quotes (`"`), the shell interprets them for string delimitation. To include literal double quotes in the string, you must escape them with `\`.

- \*\*Role of `/`\*\*: The `/` characters are purely delimiters in the `sed` substitution command and do not escape characters. They separate the different parts of the `sed` command.

### Summary

- \*\*`/`\*\*: Used as a delimiter in `sed` commands to separate the search pattern from the replacement text.

- \*\*`\`\*\*: Used to escape special characters like double quotes within the `sed` command when using double quotes for the command.

If you were using single quotes (`'`) for the entire `sed` command, you wouldn't need to escape the double quotes inside it:

```bash

sed -i 's/"region": "us-east-1"/"region": "us-west-2"/' file.json

```

Here, single quotes prevent the shell from interpreting any characters inside, so no escaping is required.

Following script This updated 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.

**#!/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 file**

**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"**

do chmod to execute shell

./ to run

Show history

Route history

Let's break down the `if` statement you provided in the shell script step by step:

### Statement:

```bash

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

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

exit 1

fi

```

### Detailed Breakdown:

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\*\*:

- 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`.

----

It uses 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. The resulting template.json file should look like this:

{

"variables": {

"region": "us-west-2",

"instance\_type": "t2.large"

},

"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"

}

]

}

----

Here's a detailed breakdown:

/"variables": {/,/}/: This is the range expression.

It starts matching from the line containing "variables": {

It continues until it finds a line with just }

This effectively limits the operation to within the "variables" section of the JSON

s/: This indicates the start of a substitution command in sed.

"region": "[^"]\*": This is the pattern to match.

"region": ": Matches the literal string "region": "

[^"]\*: This is a wildcard pattern

[^"] means "any character that is not a double quote"

\* means "zero or more of the preceding character"

So [^"]\* matches any string not containing a double quote

/"region": "'"${NEW\_REGION}"'": This is the replacement pattern.

It replaces the matched pattern with "region": " followed by the value of ${NEW\_REGION}, and then a closing quote.

/: This marks the end of the substitution command.

The wildcard [^"]\* is crucial here because it allows matching any current value for the region, regardless of what it is, without accidentally capturing more than intended.

This regular expression pattern ensures that only the value within the "region" field in the "variables" section is replaced, leaving all other occurrences of "region" in the file (such as those in placeholders) untouched.

---

Wildcard in sed: The sed command uses ".\*" to match any existing value, making it dynamic. It will replace whatever value is currently set, whether it's "us-east-1", "us-west-1", or anything else.

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.

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

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