Shell Scripting Tutorial: From Basics to Advanced

1. Introduction to Shell Scripting

- **Definition**: Shell scripting involves writing scripts to automate tasks in a Unix/Linux environment. It is crucial for DevOps engineers to manage and automate day-to-day activities effectively.
- **Purpose**: Reduces manual efforts in tasks such as file creation, system maintenance, and other repetitive activities.

2. Basic Commands and Concepts

- **Creating and Listing Files**:
  - **Create a File**:

    ```
    touch filename.sh
    ```

  - **List Files**:

    ```
    ls
    ```

  - **List Files with Details**:

    ```
    ls -l
    ```

  - **List Files with Timestamps**:

    ```
    ls -ltr
    ```

- **Reading Manual Pages**:
  - **View Manual for a Command**:

    ```
    man command
    ```

  - Example: For the `touch` command:

    ```
    man touch
    ```

- **Editing Files**:
  - **Open a File with VI**:

    ```
    vi filename.sh
    ```

  - **Basic VI Commands**:
    - Enter Insert Mode: Press `i`
    - Save and Exit: `:wq`
    - Exit without Saving: `:q!`

- **Viewing File Content**:
  - **Display File Contents**:

    ```
    cat filename.sh
    ```

3. Writing and Executing Shell Scripts

- **Shebang Line**: Specifies the script's interpreter.

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

- **Basic Shell Script**: Print a message.

```
#!/bin/bash
echo "My name is Abhishek"
```

- **Make Script Executable**:

```
chmod +x filename.sh
```

- **Execute Script**:

```
./filename.sh
```

4. Permissions and Security

- **Change File Permissions**:
  - **Grant Full Permissions to Owner**:

```
chmod 777 filename.sh
```

  - **Grant Read Permission Only**:

```
chmod 444 filename.sh
```

5. Advanced Shell Scripting Concepts

- **Creating Multiple Files**:
  - **Create Multiple Files**:

```
touch file{1..1000}
```

- **Automated File Operations**:
  - **Create Directory and Files**:

```
mkdir Abhishek
cd Abhishek
touch firstfile.txt secondfile.txt
```

6. Common Commands and Their Uses

- **Present Working Directory**:

```
pwd
```

- **Change Directory**:

```
cd directory_name
```

- **Go Up One Directory**:

```
cd ..
```

- **History of Commands**:

```
history
```

- **Monitor System Performance**:
  - **Top Command**: Displays system performance.

```
top
```

7. Example Shell Script for DevOps Tasks

- **Script to Create Directory and Files**:

```
#!/bin/bash
mkdir Abhishek
cd Abhishek
touch firstfile.txt secondfile.txt
```

- **Execute Script**:

```
./example_script.sh
```

8. Use Cases in DevOps

- **Automating Routine Tasks**: Shell scripts are used to automate tasks such as system monitoring, file management, and deployment processes.
- **Example Use Case**: A DevOps engineer might write a script to log into multiple machines, check system status, and send notifications.

Summary

Shell scripting simplifies complex tasks and automates routine operations. Mastering basic commands, file permissions, and scripting syntax will enhance efficiency and productivity in DevOps roles.

Shell Scripting for DevOps | Zero 2 Hero Part-2

1. Introduction to Advanced Shell Scripting

- **Objective**: The session focuses on creating a more complex script to check the health of a virtual machine (VM) by leveraging Bash scripting.
- **Environment**: The instructor uses an Ubuntu EC2 instance, but the concepts are applicable to any Unix/Linux environment.

2. Setting Up the Script

- **Shebang Line**: Specifies that the script should be executed using the Bash shell.

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

- **Adding Metadata**: It's crucial to include details such as the author, date, and purpose of the script for documentation and version control.

```
#!/bin/bash
# Author: Abhishek
# Date: 1st December
# Purpose: This script outputs the node health.
# Version: v1
```

3. Basic Commands in the Health Check Script

- **Disk Space Check**:
  - **Command**: `df -h`
  - **Explanation**: This command displays disk space usage in a human-readable format.

```
echo "Disk Space:"
df -h
```

- **Memory Usage Check**:
  - **Command**: `free -g`
  - **Explanation**: Displays the memory usage in gigabytes.

```
echo "Memory:"
free -g
```

- **CPU Core Count**:
  - **Command**: `nproc`
  - **Explanation**: Shows the number of CPU cores available on the system.

```
echo "CPU:"
nproc
```

4. Improving Script Readability

- **Adding Labels for Output Sections**: Use `echo` statements to label each section, making the output easier to understand.

```
echo "Disk Space:"
df -h
echo "Memory:"
free -g
echo "CPU:"
nproc
```

5. Debugging and Script Execution

- **Enable Debugging Mode**:
  - **Command**: `set -x`
  - **Explanation**: This command prints each command and its arguments as they are executed, which is helpful for troubleshooting.

```
#!/bin/bash
set -x
echo "Disk Space:"
df -h
echo "Memory:"
free -g
echo "CPU:"
nproc
```

- **Run the Script**:
  - **Making the Script Executable**:

```
chmod +x node_health.sh
```

  - **Executing the Script**:

```
./node_health.sh
```

6. Advanced Scripting Concepts Mentioned

- **Piping (`|`)**:
  - **Example**: Combines commands by passing the output of one command as input to another.

```
df -h | grep "/dev/sda1"
```

- **Using `grep` and `awk`**:
  - **Explanation**: These commands are used for searching patterns and processing text within scripts.
  - **Example**: Filtering disk usage information and extracting specific fields.

```
df -h | grep "/dev/sda1" | awk '{print $5}'
```

- **Script Error Handling**:
  - **Stop Script on Error**:

```
set -e
```

  - **Pipeline Error Handling**:

```
set -o pipefail
```

  - **Combined Example**:

```
#!/bin/bash
set -e
set -o pipefail
df -h | grep "/dev/sda1" | awk '{print $5}'
```

7. Best Practices Recap

- **Include Metadata**: Always start your script with metadata for clarity and documentation.
- **Use Debugging Tools**: Utilize `set -x` to help identify issues during script execution.
- **Version Control**: Manage your scripts using version control systems like Git to track changes and collaborate effectively.

8. Interview Tips and Common Questions

- **Frequently Asked Questions**:
  - Explain how `grep` and `awk` are used in shell scripting.
  - Describe how to handle script errors using `set -e` and `set -o pipefail`.
  - Discuss the difference between `sh` and `bash`.
- **Key Commands to Master**:
  - **`grep`**: For searching text patterns.
  - **`awk`**: For extracting and manipulating text.
  - **`df`**, **`free`**, **`nproc`**: For system resource monitoring.

Summary

This session advanced your understanding of shell scripting by focusing on practical applications such as system health checks. By mastering these concepts, you'll be able to create powerful scripts to automate tasks, which is crucial for efficiency in DevOps roles.

Here's the detailed format for the 20 shell scripting and Linux interview questions mentioned in the video, including the explanations and examples provided:

---

1. List Some Most Commonly Used Shell Commands

- **Purpose**: Understand basic shell commands frequently used in daily tasks.
- **Example Commands**:
  - `ls` – List files and directories.
  - `cp` – Copy files or directories.
  - `mv` – Move or rename files or directories.
  - `mkdir` – Create a new directory.
  - `touch` – Create an empty file.
  - `vim` – Open a file in the Vim editor.
  - `grep` – Search text using patterns.
  - `df` – Report disk space usage.
  - `top` – Display system tasks and resource usage.
  - **Notes**: Avoid mentioning less common commands used only for debugging unless specifically asked.

2. Write a Simple Shell Script to List All Processes

- **Purpose**: Demonstrate understanding of process listing and filtering.
- **Example Command**:

```
ps -ef
```

- **Filtering Process IDs**:

```
ps -ef | awk '{print $2}'
```

- **Explanation**: `ps -ef` lists all processes. `awk '{print $2}'` extracts the process IDs from the output.

3. Write a Shell Script to Print Only Errors from a Remote Log

- **Purpose**: Retrieve and filter specific information from a remote file.
- **Example Commands**:
  - Retrieve log using `curl`:

    ```
    curl http://example.com/logfile
    ```

  - Filter errors using `grep`:

    ```
    curl http://example.com/logfile | grep "ERROR"
    ```

- **Explanation**: `curl` fetches the log, and `grep` filters lines containing "ERROR".

4. Write a Shell Script to Print Numbers Divisible by 3 and 5 but Not by 15

- **Purpose**: Show ability to use loops and conditional statements in shell scripting.
- **Example Script**:

```
#!/bin/bash
for i in {1..100}
do
  if [[ $((i % 3)) -eq 0 || $((i % 5)) -eq 0 ]] && [[ $((i % 15)) -ne 0 ]]
  then
    echo $i
  fi
done
```

- **Explanation**: Loop through numbers 1 to 100, check if divisible by 3 or 5 but not 15, and print them.

5. Write a Shell Script to Print Occurrences of a Specific Alphabet in a Word

- **Purpose**: Use text processing tools to count occurrences.
- **Example Script**:

```
#!/bin/bash
word="Mississippi"
echo $word | grep -o "s" | wc -l
```

- **Explanation**: `grep -o "s"` extracts occurrences of 's', and `wc -l` counts them.

6. What is Cron and How Do You Use It?

- **Purpose**: Schedule tasks using cron jobs.
- **Example Command**:

```
crontab -e
```

  - **Example Cron Job**:

```
0 18 * * * /path/to/script.sh
```

- **Explanation**: `crontab -e` opens the cron table for editing; this job runs a script daily at 6 PM.

7. How to Open a File in Read-Only Mode

- **Purpose**: Demonstrate file handling in read-only mode.
- **Example Command**:

```
vim -R filename
```

- **Explanation**: `vim -R` opens a file in read-only mode.

8. Explain Soft Links and Hard Links

- **Purpose**: Understand file linking mechanisms.
- **Soft Link Example**:

```
ln -s /path/to/original /path/to/softlink
```

- **Hard Link Example**:

```
ln /path/to/original /path/to/hardlink
```

- **Explanation**: Soft links point to the file name; hard links point to the file's inode, making both links indistinguishable from the original file.

9. What is the Difference Between `continue` and `break` in Loops?

- **Purpose**: Differentiate between loop control commands.
- **Example Commands**:
  - `continue` – Skip the current iteration and continue with the next iteration of the loop.
  - `break` – Exit the loop entirely.
- **Example**:

```
for i in {1..10}
do
  if [[ $i -eq 5 ]]; then
    continue
  fi
  echo $i
done
```

  - This loop will skip printing the number 5.

10. What is `set -x` and When to Use It?

- **Purpose**: Enable debugging in scripts.
- **Example Command**:

```
set -x
# Your script here
set +x
```

- **Explanation**: `set -x` prints each command before executing it, helpful for debugging.

11. What is `logrotate` and How Does It Work?

- **Purpose**: Manage log files by rotating and compressing them.
- **Example Configuration**:

```
/var/log/myapp.log {
  weekly
  rotate 4
  compress
  missingok
  notifempty
  create 640 root root
}
```

- **Explanation**: This configuration rotates logs weekly, keeps four weeks of logs, compresses them, and creates new logs with specific permissions.

12. How to Handle Unset Variables in Shell Scripts

- **Purpose**: Prevent issues with unset variables.
- **Example Command**:

```
set -u
# Your script here
```

- **Explanation**: `set -u` causes the script to exit if an unset variable is used.

13. Different Kinds of Loops in Shell Scripting

- **Purpose**: Explain loop types.
- **Examples**:
  - `for` loop:

```
for i in {1..5}
do
  echo $i
done
```

  - `while` loop:

```
i=1
while [ $i -le 5 ]
do
  echo $i
  ((i++))
done
```

  - `until` loop:

```
i=1
until [ $i -gt 5 ]
do
  echo $i
```

```
  ((i++))
done
```

14. Are Shell Scripts Dynamically or Statically Typed?

- **Purpose**: Understand typing in shell scripting.
- **Explanation**: Shell scripting is dynamically typed, meaning variable types are interpreted at runtime.

15. What are Some Networking Troubleshooting Tools?

- **Purpose**: Identify tools for network debugging.
- **Example Tools**:
  - `ping` – Test connectivity.
  - `traceroute` – Trace the path to a network host.
  - `netstat` – Display network connections and statistics.

16. How to Use `awk` for Field Extraction?

- **Purpose**: Extract specific fields from text.
- **Example Command**:

```
ps -ef | awk '{print $1}'
```

- **Explanation**: `awk` processes text and prints specified fields.

17. What is `grep` and How to Use It?

- **Purpose**: Search text using patterns.
- **Example Command**:

```
grep "pattern" filename
```

- **Explanation**: `grep` searches for lines matching a pattern.

18. How to Create and Use Aliases in Shell?

- **Purpose**: Simplify commands using aliases.
- **Example Command**:

```
alias ll='ls -l'
```

- **Explanation**: `alias` creates a shortcut for a command.

19. Difference Between `sh` and `bash`

- **Purpose**: Understand shell differences.
- **Explanation**:
  - `sh` – Bourne shell, basic scripting.
  - `bash` – Bourne Again Shell, enhanced features and scripting capabilities.

20. What is `set -e` and How is it Useful?

- **Purpose**: Ensure script execution halts on errors.
- **Example Command**:

```
set -e
# Your script here
```

- **Explanation**: `set -e` exits the script if any command fails.

This format provides a concise yet comprehensive overview of each question and answer, suitable for preparing for shell scripting and Linux interviews.

Certainly! Here are detailed notes including commands with explanations based on the transcript titled **"Day-7 | Live AWS Project using SHELL SCRIPTING for DevOps | AWS DevOps project| #devops #aws #2023"**:

Title: Day-7 | Live AWS Project using SHELL SCRIPTING for DevOps | AWS DevOps project| #devops #aws #2023

Transcript Summary:

Introduction:

- The video focuses on a shell scripting project within a DevOps course to manage AWS resources effectively.

Why Move to Cloud Infrastructure:

1. **Manageability:** Simplifies maintenance by avoiding physical servers.
2. **Cost-Effectiveness:** Uses a pay-as-you-go model instead of fixed costs.

Resource Usage Tracking:

- Methods: Shell scripting, Python scripts, Lambda functions, and AWS CLI.

Project Overview:

- Objective: Create a shell script to track and report AWS resources such as EC2 instances, S3 buckets, Lambda functions, and IAM users.
- Integration: The script will be scheduled with a Cron job for automated daily reporting.

Detailed Shell Script Instructions:

1. **Script Creation:**
    - **Script Name:** `AWS_resource_tracker.sh`
    - **Initial Setup:**

      ```
      #!/bin/bash
      # This script tracks AWS resources and generates a report
      ```

        - `#!/bin/bash`: Shebang to specify the script should be executed with Bash.
        - `#`: Comment line for documentation.
    - **Commands to Track AWS Resources:**
        - **List S3 Buckets:**

          ```
          aws s3 ls
          ```

- `aws s3 ls`: Lists all S3 buckets in the AWS account.

- **Describe EC2 Instances:**

```
aws ec2 describe-instances
```

  - `aws ec2 describe-instances`: Provides details about EC2 instances, including IDs, statuses, and more.

- **List Lambda Functions:**

```
aws lambda list-functions
```

  - `aws lambda list-functions`: Lists all Lambda functions in the AWS account.

- **List IAM Users:**

```
aws iam list-users
```

  - `aws iam list-users`: Lists all IAM users in the AWS account.

- **Improving Script Output:**

  - **Redirection to File:**

```
aws s3 ls > s3_buckets_report.txt
aws ec2 describe-instances > ec2_instances_report.txt
aws lambda list-functions > lambda_functions_report.txt
aws iam list-users > iam_users_report.txt
```

    - `>`: Redirects command output to a specified file.

  - **Adding Echo Statements:**

```
echo "S3 Buckets Report" > aws_report.txt
aws s3 ls >> aws_report.txt
echo "EC2 Instances Report" >> aws_report.txt
aws ec2 describe-instances >> aws_report.txt
echo "Lambda Functions Report" >> aws_report.txt
aws lambda list-functions >> aws_report.txt
echo "IAM Users Report" >> aws_report.txt
aws iam list-users >> aws_report.txt
```

    - `echo "text"`: Adds descriptive text to the output file.
    - `>>`: Appends output to the file.

- **Using `jq` for JSON Parsing:**

```
aws ec2 describe-instances | jq '.Reservations[].Instances[].InstanceId'
```

  - `jq`: Command-line tool for parsing JSON data.
  - `'.Reservations[].Instances[].InstanceId'`: Extracts EC2 instance IDs from the JSON output.

2. **Script Execution:**
    - **Setting Permissions:**

      ```
      chmod +x AWS_resource_tracker.sh
      ```

        - `chmod +x`: Grants execute permissions to the script.
    - **Running the Script:**

      ```
      ./AWS_resource_tracker.sh
      ```

        - `./AWS_resource_tracker.sh`: Executes the script.
3. **Cron Job Integration:**
    - **Setting Up a Cron Job:**

      ```
      crontab -e
      ```

        - Opens the Crontab editor.
    - **Adding a Cron Job Entry:**

      ```
      0 0 * * * /path/to/AWS_resource_tracker.sh
      ```

        - `0 0 * * *`: Schedules the script to run at midnight every day.
        - `/path/to/AWS_resource_tracker.sh`: Path to the shell script.

Conclusion:

- The script should be tested to ensure it generates accurate reports and is correctly integrated with Cron for automation.

Feel free to ask if you need any further explanations or additional details!

Here are detailed notes on the shell scripting project explained in the video:

Overview

The project involves writing a shell script to interact with AWS services. The script will:

1. Connect to AWS.
2. List active resources based on user input.
3. Be adaptable to other cloud providers like Azure or GCP.

Script Requirements

- **Arguments**: Region and Service Name
- **Services**: EC2, S3, EBS, DynamoDB, RDS, etc.

Initial Setup

1. **Script Header Comments**
    - `#!/bin/bash`: Shebang to specify the script interpreter.
    - Comments to explain the script's purpose, author, and version.

- List supported AWS services.
- Usage example.

2. **Validation**
   - Check if required arguments are provided.
   - Example command: `if [ "$#" -ne 2 ]; then echo "Usage: $0 <region> <service>"; exit 1; fi`

AWS CLI Setup

1. **Check Installation**
   - Verify if AWS CLI is installed.
   - Command: `aws --version`
   - If not installed, install it using package managers (`apt`, `yum`, etc.).
   - Example command to check: `if ! command -v aws &> /dev/null; then echo "AWS CLI not installed"; exit 1; fi`
2. **Check Configuration**
   - Verify AWS CLI configuration.
   - Check if the configuration directory exists: `if [ ! -d ~/.aws ]; then echo "AWS CLI not configured"; exit 1; fi`
3. **Configure AWS CLI**
   - Command: `aws configure`
   - Inputs: Access Key, Secret Key, Region, Output format.

Main Script Logic

1. **Service Commands**
   - Use AWS CLI commands for different services:
     - EC2: `aws ec2 describe-instances`
     - S3: `aws s3 ls`
     - EBS: `aws ec2 describe-volumes`
     - DynamoDB: `aws dynamodb list-tables`
     - RDS: `aws rds describe-db-instances`
2. **Switch Case for Services**
   - Implement a `case` statement to handle different services.
   - Example:

```
case $2 in
  ec2) aws ec2 describe-instances --region $1 ;;
  s3) aws s3 ls --region $1 ;;
  ebs) aws ec2 describe-volumes --region $1 ;;
  *) echo "Unsupported service: $2" ;;
esac
```

Security and Best Practices

1. **File Permissions**

- Set script permissions: `chmod 711 script.sh`
- Explanation:
  - `7` for owner (read, write, execute)
  - `1` for group (execute only)
  - `1` for others (execute only)

2. **Optional: Automate Execution**
   - Set up a cron job to run the script periodically.
   - Example cron job entry: `0 21 * * * /path/to/script.sh region service`

Testing and Debugging

1. **Run Script**
   - Command: `./script.sh <region> <service>`
   - Verify output and error messages.
2. **Verify AWS Resources**
   - Check AWS Console to ensure the script's output matches the actual resources.

Additional Notes

- Always refer to AWS CLI documentation for specific commands and their arguments.
- Secure sensitive information by limiting script access and permissions.

These notes cover the steps and commands required to create and manage the shell script, ensuring it is both functional and secure.

Day-8 | DevOps Zero to Hero | Shell Scripting Project Used In Real Time | GitHub API Integration

Overview

In this session, the focus is on integrating GitHub API with shell scripting. The goal is to automate the process of listing people who have access to a GitHub repository, which is a common task for DevOps engineers.

Concepts Covered

1. **Introduction to GitHub API and CLI**
   - **User Interface (UI) vs API**: UI involves interacting with a web interface, while API allows programmatic access.
   - **CLI vs API**: CLI tools provide command-line interfaces (e.g., `kubectl` for Kubernetes), whereas APIs enable interaction via HTTP requests (e.g., GitHub API).
2. **What is an API?**
   - **API (Application Programming Interface)**: Allows programs to communicate with each other. For GitHub, APIs enable you to perform tasks programmatically that you would typically do through the UI.

3. **Using GitHub API**
    - **API Documentation**: Always refer to API documentation to understand how to make requests and what URLs to use.
    - **Example**: GitHub API for listing pull requests or issues.

Shell Script Project

1. **Objective**
    - Automate the task of listing users who have access to a GitHub repository.
2. **GitHub API Integration**
    - **API Token**: Required for authentication. Generated from GitHub under Developer Settings -> Personal Access Tokens.
3. **Creating the Shell Script**
    - **Script Overview**: The script uses `curl` to interact with GitHub's API and `jq` to parse the JSON response.
    - **Basic Commands and Functions**:
        - `curl` : A command-line tool to make HTTP requests.
        - `jq` : A tool to parse and filter JSON data.
4. **Script Walkthrough**
    - **Generating API Token**:

```
# Go to GitHub > Settings > Developer Settings > Personal Access Tokens
# Generate a new token with appropriate permissions.
```

    - **Exporting Token and Username**:

```
export GITHUB_TOKEN=your_generated_token
export GITHUB_USER=your_username
```

    - **Script Commands**:
        - **Get Repository Collaborators**:

```
curl -H "Authorization: token $GITHUB_TOKEN" \
     "https://api.github.com/repos/owner/repo/collaborators"
```

        - **List Issues**:

```
curl -H "Authorization: token $GITHUB_TOKEN" \
     "https://api.github.com/repos/owner/repo/issues"
```

5. **Handling Output**
    - **Parsing JSON with `jq`**:

```
# Extracting usernames with read access
curl -H "Authorization: token $GITHUB_TOKEN" \
     "https://api.github.com/repos/owner/repo/collaborators" | \
     jq '.[] | select(.permissions.pull == true) | .login'
```

6. **Script Functionality**
    - **Forming and Executing `curl` Commands**:
        - Form the API URL and request parameters.
        - Use `jq` to filter and display relevant information.
7. **Error Handling and Improvements**
    - **Permissions and Access**: Ensure the script has the correct permissions and API tokens.
    - **Helper Functions**: Add helper functions to check command-line arguments and improve script robustness.

Example Shell Script

```
#!/bin/bash

# Validate the number of command-line arguments
if [ "$#" -ne 2 ]; then
  echo "Usage: $0 <repository> <organization>"
  exit 1
fi

REPO=$1
ORG=$2

# Get the list of collaborators
curl -H "Authorization: token $GITHUB_TOKEN" \
     "https://api.github.com/repos/$ORG/$REPO/collaborators" | \
     jq '.[] | select(.permissions.pull == true) | .login'
```

Running the Script

1. **Clone Repository**:

    ```
    git clone https://github.com/yourusername/shell-scripting-project.git
    ```

2. **Navigate to Script Directory**:

    ```
    cd shell-scripting-project
    ```

3. **Run the Script**:

    ```
    bash list_users.sh your-repo your-org
    ```

Conclusion

This session demonstrates how to automate GitHub repository management tasks using shell scripting and API integration. The provided script lists users with read access to a repository, improving efficiency for DevOps engineers.

#The script helps you find out about different things (called resources) in your AWS account, like your virtual servers (EC2), databases (RDS), and storage buckets (S3).

##

Check if the required number of arguments are passed

if [$# -ne 2]; then echo "Usage: ./aws_resource_list.sh " echo "Example: ./aws_resource_list.sh us-east-1 ec2" exit 1 fi

Assign the arguments to variables and convert the service to lowercase

aws_region=$1 aws_service=$2

Check if the AWS CLI is installed

if ! command -v aws &> /dev/null; then echo "AWS CLI is not installed. Please install the AWS CLI and try again." exit 1 fi

Check if the AWS CLI is configured

if [! -d ~/.aws]; then echo "AWS CLI is not configured. Please configure the AWS CLI and try again." exit 1 fi

List the resources based on the service

case $aws_service in ec2) echo "Listing EC2 Instances in $aws_region" aws ec2 describe-instances --region $aws_region ;; rds) echo "Listing RDS Instances in $aws_region" aws rds describe-db-instances --region $aws_region ;; s3) echo "Listing S3 Buckets in $aws_region" aws s3api list-buckets --region $aws_region ;; cloudfront) echo "Listing CloudFront Distributions in $aws_region" aws cloudfront list-distributions --region $aws_region ;; vpc) echo "Listing VPCs in $aws_region" aws ec2 describe-vpcs --region $aws_region ;; iam) echo "Listing IAM Users in $aws_region" aws iam list-users --region $aws_region ;; route5) echo "Listing Route53 Hosted Zones in $aws_region" aws route53 list-hosted-zones --region $aws_region ;; cloudwatch) echo "Listing CloudWatch Alarms in $aws_region" aws cloudwatch describe-alarms --region $aws_region ;; cloudformation) echo "Listing CloudFormation Stacks in $aws_region" aws cloudformation describe-stacks --region $aws_region ;; lambda) echo "Listing Lambda Functions in $aws_region" aws lambda list-functions --region $aws_region ;; sns) echo "Listing SNS Topics in $aws_region" aws sns list-topics --region $aws_region ;; sqs) echo "Listing SQS Queues in $aws_region" aws sqs list-queues --region $aws_region ;; dynamodb) echo "Listing DynamoDB Tables in $aws_region" aws dynamodb list-tables --region $aws_region ;; ebs) echo "Listing EBS Volumes in $aws_region" aws ec2 describe-volumes --region $aws_region ;; *) echo "Invalid service. Please enter a valid service." exit 1 ;; esac

Detailed Notes for the Bash Script

This bash script helps you list various resources in an AWS (Amazon Web Services) account. I'll break down how it works in a simple way, like explaining it to a 10-year-old.

Script Overview

1. **What Does the Script Do?**
    - The script helps you find out about different things (called resources) in your AWS account, like your virtual servers (EC2), databases (RDS), and storage buckets (S3).
2. **What Services Can It Check?**
    - EC2 (virtual servers)
    - RDS (databases)
    - S3 (storage buckets)
    - CloudFront (content delivery network)
    - VPC (network setup)
    - IAM (user permissions)
    - Route53 (DNS service)
    - CloudWatch (monitoring)
    - CloudFormation (setup templates)
    - Lambda (serverless functions)
    - SNS (notification service)
    - SQS (message queue)
    - DynamoDB (NoSQL database)
    - EBS (extra storage volumes)

How It Works

1. **Script Start**

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

    - This line tells the computer that this is a bash script.
2. **Information About the Script**

    ```
    ################################################################################
    # Author: Abhishek Veeramalla
    # Version: v0.0.1
    ```

    - Tells who wrote the script and its version.
3. **How to Use the Script**

    ```
    # Usage: ./aws_resource_list.sh <aws_region> <aws_service>
    # Example: ./aws_resource_list.sh us-east-1 ec2
    ```

    - You need to give two pieces of information:
        - **AWS Region** (where your resources are located)
        - **AWS Service** (what kind of resource you want to list)
4. **Checking for Correct Usage**

```
if [ $# -ne 2 ]; then
    echo "Usage: ./aws_resource_list.sh <aws_region> <aws_service>"
    echo "Example: ./aws_resource_list.sh us-east-1 ec2"
    exit 1
fi
```

- **Checks** if you've given exactly 2 pieces of information. If not, it shows how to use the script and stops.

5. **Getting the Inputs**

```
aws_region=$1
aws_service=$2
```

- Saves the two pieces of information into variables so the script can use them.

6. **Checking for AWS CLI**

```
if ! command -v aws &> /dev/null; then
    echo "AWS CLI is not installed. Please install the AWS CLI and try again."
    exit 1
fi
```

- **Checks** if AWS CLI (a tool to interact with AWS) is installed. If not, it tells you to install it and stops.

7. **Checking AWS CLI Configuration**

```
if [ ! -d ~/.aws ]; then
    echo "AWS CLI is not configured. Please configure the AWS CLI and try
again."
    exit 1
fi
```

- **Checks** if AWS CLI is set up correctly. If not, it tells you to configure it and stops.

8. **Listing Resources**

```
case $aws_service in
    ec2)
        echo "Listing EC2 Instances in $aws_region"
        aws ec2 describe-instances --region $aws_region
        ;;
...
```

- **Checks** which service you want to list resources for and runs the appropriate command to get the list.
- Each service has a specific command to list resources (e.g., `aws ec2 describe-instances` for EC2).

9. **Handling Invalid Services**

```
*)
    echo "Invalid service. Please enter a valid service."
```

```
    exit 1
    ;;
```

- **If** you type an incorrect service name, it tells you that the service is invalid and stops.

Commands Explained

- **`aws ec2 describe-instances --region $aws_region`**
  - Lists all EC2 instances (virtual servers) in the specified region.
- **`aws rds describe-db-instances --region $aws_region`**
  - Lists all RDS instances (databases) in the specified region.
- **`aws s3api list-buckets --region $aws_region`**
  - Lists all S3 buckets (storage) in the specified region.
- **`aws cloudfront list-distributions --region $aws_region`**
  - Lists all CloudFront distributions (content delivery) in the specified region.
- **`aws ec2 describe-vpcs --region $aws_region`**
  - Lists all VPCs (network setups) in the specified region.
- **`aws iam list-users --region $aws_region`**
  - Lists all IAM users (permissions) in the specified region.
- **`aws route53 list-hosted-zones --region $aws_region`**
  - Lists all Route53 hosted zones (DNS settings) in the specified region.
- **`aws cloudwatch describe-alarms --region $aws_region`**
  - Lists all CloudWatch alarms (monitoring) in the specified region.
- **`aws cloudformation describe-stacks --region $aws_region`**
  - Lists all CloudFormation stacks (setup templates) in the specified region.
- **`aws lambda list-functions --region $aws_region`**
  - Lists all Lambda functions (serverless functions) in the specified region.
- **`aws sns list-topics --region $aws_region`**
  - Lists all SNS topics (notifications) in the specified region.
- **`aws sqs list-queues --region $aws_region`**
  - Lists all SQS queues (message queues) in the specified region.
- **`aws dynamodb list-tables --region $aws_region`**
  - Lists all DynamoDB tables (NoSQL databases) in the specified region.
- **`aws ec2 describe-volumes --region $aws_region`**

- Lists all EBS volumes (extra storage) in the specified region.

This should help you understand the script and how it works! If you have any more questions or need more details, feel free to ask.