

Answer

As a DevOps engineer managing a GitHub repository, you will learn several key concepts related to access management, API usage, and scripting. Here's a breakdown of these concepts:

Understanding Repository Access

1. **Repository Ownership and Access Management:**
 - You are responsible for maintaining the repository, which includes managing user access.
 - You can view and modify who has access to the repository by navigating to the **Settings** tab and selecting **Collaborators & teams**. This section allows you to see all users with access and their respective roles (read, write, admin) 1 2.
2. **Importance of Monitoring Access:**
 - Regularly checking who has access is crucial, especially when team members leave the organization. You may need to revoke access to maintain security.
 - You can also promote users from read to write access as needed, ensuring that team members have the appropriate permissions to perform their tasks 3 4.

Automation through Scripting

1. **Writing Shell Scripts:**
 - Instead of manually checking access each time, you can automate this process using shell scripts.
 - These scripts can integrate with GitHub's API to fetch and manage user access programmatically, saving time and reducing human error.
2. **GitHub Integration:**
 - To interact with GitHub programmatically, you can use either the GitHub API or the GitHub CLI.
 - The API allows you to send HTTP requests to GitHub and receive data in a

<https://www.perplexity.ai/search/extract-here-and-explain-in-depth-HFyrVRqesn9kRZqNL2A>

<https://chatgpt.com/c/95f80d26-d143-4655-a350-53e657f35b45>

<https://www.perplexity.ai/search/extract-here-and-explain-in-depth-HFyrVRqesn9kRZqNL2A>

2. GitHub Integration:

- To interact with GitHub programmatically, you can use either the GitHub API or the GitHub CLI.
- The API allows you to send HTTP requests to GitHub and receive data in a structured format (usually JSON), which can be parsed and used in your scripts

1

Understanding APIs

1. What is an API?:

- An API (Application Programming Interface) allows different software applications to communicate with each other. For GitHub, it provides endpoints to access repository data, user information, and more.
- You can think of the API as a bridge between your scripts and GitHub, enabling you to perform actions like listing collaborators or changing permissions without using the web interface

2 5

2. Using APIs in Scripts:

- You can use tools like `curl` in shell scripts or libraries like `requests` in Python to make API calls to GitHub.
- For example, you can write a script that checks the list of collaborators and their access levels, and then outputs this information in a readable format or takes action based on it.

Practical Application

1. Daily Operations:

- As part of your daily routine, you might want to run scripts that check for access levels, notify you of any changes, or even automate the process of granting or revoking access based on certain criteria (like a user's employment status).

2. Documentation and Reference:

- Each API has documentation that details how to use it, including the endpoints available, required parameters, and response formats. Familiarizing yourself with this documentation is essential for effective API usage [4](#) [5](#).

By understanding these concepts, you will be well-equipped to manage your GitHub repository efficiently, automate repetitive tasks, and maintain security through proper access management.

Explain in more detail above concept:

Answer

Let's break down the concepts you mentioned into simpler, more digestible parts, focusing on what a DevOps engineer needs to learn about managing a GitHub repository, using APIs, and automating tasks.

Overview of Responsibilities as a DevOps Engineer

Managing a GitHub Repository

As a DevOps engineer, you are responsible for maintaining a GitHub repository. This includes:

- **Creating Users:** You may need to add new users to the repository, assigning them appropriate access levels (read, write, admin).
- **Monitoring for Vulnerabilities:** Keeping an eye on the repository for any security issues or vulnerabilities.
- **CI/CD Pipelines:** Setting up Continuous Integration and Continuous Deployment (CI/CD) pipelines to automate testing and deployment processes.

Access Management

When someone asks about access to the repository, for example, a colleague named Abishek, you would typically:

1. **Check Access:** Go to the **Settings** tab of the repository and navigate to **Collaborators & Teams** to see who has access.

2. **Provide Information:** You might take a screenshot of the access list and share it with them.

However, this manual checking can become cumbersome, especially if you need to do it frequently or if a team member is leaving the organization.

Automating Access Management

Writing a Shell Script

Instead of manually checking access each time, you can automate this process with a shell script. This script would require integration with GitHub, allowing you to:

- **List Collaborators:** Automatically retrieve the list of users with access to the repository.
- **Revoke Access:** Easily revoke access for users who are leaving.
- **Modify Access Levels:** Change a user's access level (e.g., from read to write) without manual intervention.

GitHub Integration

To interact with GitHub programmatically, you have two main options: **API** and **CLI**.

1. API (Application Programming Interface):

- An API allows your scripts to communicate with GitHub directly, enabling you to perform actions like retrieving user access or modifying permissions without using the web interface.

- APIs expose endpoints that you can call using HTTP requests. For example, you can use the GitHub API to get a list of collaborators.

2. CLI (Command Line Interface):

- GitHub also offers a CLI that allows you to run commands directly from your terminal. While this is useful, APIs provide more flexibility for automation.

Understanding APIs

What is an API?

- **Definition:** An API is a set of rules that allows one piece of software to interact with another. In this case, it lets your scripts communicate with GitHub.
- **User Interface vs. API:** When you use GitHub in a web browser, you interact with its user interface. In contrast, using an API means you can programmatically request data or perform actions without a graphical interface.

How to Use APIs

1. Making Requests:

- You can use tools like `curl` in shell scripts or libraries like `requests` in Python to make HTTP requests to the GitHub API.
- For example, to list collaborators, you would send a GET request to the appropriate GitHub API endpoint.

2. Programming Languages:

- You can use various programming languages to interact with APIs, including Python, JavaScript, or Bash. Each language has libraries or modules to simplify making HTTP requests.

Consuming APIs

- As a DevOps engineer, you typically **consume** APIs rather than create them. This means you use existing APIs to automate tasks and gather information.
- For example, when working with AWS, you might use the `boto3` library in Python to interact with AWS services programmatically, similar to how you would use the GitHub API for repository management.

Documentation

- Every API has documentation that explains how to use it, including available endpoints, required parameters, and response formats.
- Familiarizing yourself with this documentation is crucial, as it will guide you in making the correct API calls.

Conclusion

By understanding these concepts, you will be able to:

- Efficiently manage access to your GitHub repository.
- Automate repetitive tasks using shell scripts and APIs.
- Use APIs to gather information and perform actions programmatically, enhancing your productivity as a DevOps engineer.

This foundational knowledge will empower you to streamline your workflows and improve collaboration within your team. If you have any further questions or need clarification on specific points, feel free to ask!

 Share  Rewrite

  ...

Answer

To understand how to interact with GitHub's API, particularly for fetching pull requests from a repository, let's break down the process step-by-step, focusing on the concepts of API usage, how to find the correct API endpoint, and how to implement it in a script.

Understanding APIs and Their URLs

What is an API?

An **API (Application Programming Interface)** allows different software applications to communicate with each other. In the context of GitHub, the API enables you to programmatically access repository data, such as pull requests, issues, and user information, without needing to use the web interface.

API URL Structure

When you want to access a specific resource using an API, you typically need to know the **URL (Uniform Resource Locator)** or endpoint. For GitHub, the base URL for the API is:

text

```
https://api.github.com
```



To access specific resources, you append paths to this base URL. For example, to get pull requests for a repository, the URL format is:

text

```
https://api.github.com/repos/{owner}/{repo}/pulls
```



- **{owner}**: This is the username or organization name that owns the repository.
- **{repo}**: This is the name of the repository.

Example

If you have a repository called `devops-hero` owned by a user `iamvamala`, the URL to access the pull requests would be:

text

```
https://api.github.com/repos/iamvamala/devops-hero/pulls
```



Finding the Right API Documentation

To understand how to use the API, you should refer to the [GitHub API documentation](#). Here's how to navigate it:

1. **Search for GitHub API Documentation:** You can find it by searching for "GitHub API documentation" in your browser.
2. **Locate Pull Requests Section:** Once in the documentation, look for the section on pull requests. This section will provide details on how to retrieve pull requests, including the necessary URL and any required parameters.
3. **Sample Code:** The documentation often includes sample code snippets for various programming languages (like Shell, Python, JavaScript) to help you get started.

Making API Calls

Using cURL in Shell Scripting

In shell scripting, you can use the `curl` command to make HTTP requests. Here's how you would use it to fetch pull requests:

bash

```
curl -H "Accept: application/vnd.github.v3+json"  
https://api.github.com/repos/iamvamala/devops-hero/pulls
```

- `-H "Accept: application/vnd.github.v3+json"`: This header specifies that you want to use version 3 of the GitHub API.

Using Python with Requests

If you prefer Python, you can use the ``requests`` library to achieve the same:

```
python
```

```
❶ Explain ⌂ import requests

    url = "https://api.github.com/repos/iamvamala/devops-
❷ hero/pulls"
    response = requests.get(url, headers={"Accept":
"application/vnd.github.v3+json"})

    if response.status_code == 200:
        pull_requests = response.json()
        for pr in pull_requests:
            print(f"PR Title: {pr['title']}, URL:
{pr['html_url']}")
        else:
            print("Failed to retrieve pull requests:",
response.status_code)
```

Explanation of the Code

- `requests.get(url, headers={ ... })`: This sends a GET request to the specified URL with the required headers.
- `response.json()`: This converts the JSON response into a Python dictionary for easy access.
- **Loop through pull requests:** You can iterate through the list of pull requests and print details like the title and URL.

Conclusion

By understanding how to construct API URLs and how to use tools like `curl` or libraries like `requests`, you can effectively interact with GitHub's API to automate tasks related to your repositories. This knowledge allows you to manage pull requests, issues, and other repository data programmatically, enhancing your efficiency as a DevOps engineer.

If you have any further questions or need clarification on specific aspects, feel free to ask!

 Share  Rewrite

  ...

13:33 - 25:09

Answer

To effectively manage access to GitHub repositories, especially as a DevOps engineer, it is crucial to understand the processes involved in checking who has access and how to automate these tasks using shell scripts. Below is a detailed explanation of why we use specific methods and tools, along with the steps to implement them.

Importance of Managing Repository Access

Why Check Access?

1. **Security:** Ensuring that only authorized users have access to sensitive code is crucial for protecting intellectual property and maintaining compliance with security policies. If a user who should no longer have access retains it, it poses a significant security risk.
2. **Compliance:** Organizations often need to comply with regulations that require detailed records of who has access to what resources. Regularly auditing access helps in maintaining compliance with these regulations.
3. **Onboarding and Offboarding:** When employees join or leave the organization, it's important to promptly grant or revoke access to repositories to ensure that only current team members can access the code.

Challenges of Manual Access Management

- **Time-Consuming:** Manually logging into GitHub, navigating to each repository, and checking the access list can be tedious and inefficient, especially for organizations with multiple repositories.
- **Error-Prone:** Human error can lead to oversight, such as failing to revoke access for a departing employee, which can pose security risks.

Automating Access Management with Shell Scripts

Using EC2 Instances

1. **Why Use EC2?**: Amazon EC2 (Elastic Compute Cloud) instances provide a scalable environment where you can run scripts without depending on local machines. This is particularly useful for running scripts that interact with external services like GitHub.
2. **Launching an EC2 Instance:**
 - You can launch an EC2 instance from the AWS Management Console. Select an appropriate AMI (e.g., Ubuntu) and configure security settings.
 - Example steps:
 - Choose an instance type.
 - Configure security groups to allow SSH access.
 - Launch the instance and note the public IP address for SSH access.

3. Logging In:

- Use SSH to connect to your EC2 instance. If you're using PuTTY, ensure you convert your PEM file to PPK format for authentication.
- Example command:

```
bash
```

```
ssh -i your-key.pem ubuntu@your-ec2-public-ip
```



Cloning the Shell Script

1. **Cloning the Repository:** Once logged in, you can clone the repository containing your shell script that manages GitHub access.

- Example command:

```
bash
```

```
git clone https://github.com/your-username/your-repo.git
```



2. Navigating to the Script Directory:

- Change into the directory where your shell script is located.
- Example command:

```
bash
```

```
cd your-repo/shell-script-directory
```



Setting Up Environment Variables

1. GitHub Username and Token:

- You need to export your GitHub username and a personal access token as environment variables for the script to authenticate API requests.
- Example commands:

```
bash
```

```
export GITHUB_USERNAME="your_github_username"  
export GITHUB_TOKEN="your_personal_access_token"
```



2. Creating a Personal Access Token:

- Navigate to **Settings** on GitHub.
- Go to **Developer settings > Personal access tokens**.
- Click **Generate new token**, name it, and select the necessary scopes (e.g., `repo` for full control of private repositories).

Running the Shell Script

After setting up the environment variables, you can run the shell script to list collaborators:

```
bash
```

```
bash list_users.sh
```



Example Shell Script to List Collaborators

Here's a simple example of what the shell script might look like:

bash

Explain

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



```
# GitHub repository details
OWNER="your_org_or_username"
REPO="your_repository_name"

# Fetch the list of collaborators
response=$(curl -s -H "Authorization: token $GITHUB_TOKEN" -H
"Accept: application/vnd.github.v3+json"
"https://api.github.com/repos/$OWNER/$REPO/collaborators")

# Output the list of collaborators
echo "Collaborators with access to the repository:"
echo "$response" | jq -r '.[].login'
```

Explanation of the Script

- **Authorization Header:** The script uses the personal access token to authenticate the API request.
- **`jq` Tool:** This command-line tool is used to parse JSON output. Ensure it is installed on your EC2 instance (`sudo apt-get install jq`).
- **Output:** The script lists the usernames of all collaborators with access to the specified repository.

Conclusion

By automating the process of checking and managing repository access using shell scripts and the GitHub API, you can save time and enhance security. This approach allows you to effectively handle common tasks related to user access management as a DevOps engineer.

Next Steps

- **Practice:** Create a test organization and repository to practice these tasks.
- **Enhance the Script:** Consider adding features to the script, such as logging or error handling.
- **Explore More APIs:** Familiarize yourself with other GitHub API endpoints to automate additional tasks related to repository management.

If you have any questions or need further clarification on specific steps or concepts, feel free to ask!

 Share  Rewrite

  ...



Detailed Explanation

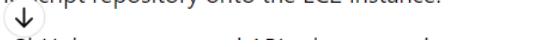
In this guide, we will walk through the process of managing GitHub repository access using a shell script and an AWS EC2 instance. This approach automates the process of checking repository access permissions, which can be tedious and time-consuming if done manually via the GitHub web interface.

Why Automate Repository Access Checks?

1. **Efficiency:** Manually logging into GitHub, navigating to each repository, and checking access permissions is time-consuming, especially if there are many repositories.
2. **Consistency:** Automating this process ensures that the checks are performed consistently, without human error.
3. **Scalability:** As the number of repositories grows, automation can handle the increased workload without additional manual effort.

Steps to Automate with Shell Script and EC2

1. **Launch an EC2 Instance:** This will provide a server environment to run our automation scripts.
2. **Set Up SSH Access:** Ensure you can log into the EC2 instance securely.
3. **Clone the Shell Script:** Use Git to clone the script repository onto the EC2 instance.



3. **Clone the Shell Script:** Use Git to clone the script repository onto the EC2 instance.
4. **Configure Environment Variables:** Set up GitHub username and API token as environment variables.
5. **Execute the Script:** Run the shell script to check repository access permissions.

Step-by-Step Guide

1. Launch an EC2 Instance

- **Go to AWS EC2 Dashboard:** Navigate to the EC2 section in AWS Management Console.
- **Launch Instance:** Choose the "Launch Instance" option.
- **Select AMI:** Choose an Ubuntu AMI (or any other Linux distribution).
- **Instance Type:** Select an appropriate instance type (e.g., t2.micro for small-scale use).
- **Configure Instance:** Configure instance details as required.
- **Add Storage:** Set up storage, typically the default settings are fine.
- **Add Tags:** Optionally, add tags to the instance for easier management.
- **Configure Security Group:** Set up security groups to allow SSH access.
- **Review and Launch:** Review the settings and launch the instance.
↓
- **Key Pair:** Select an existing key pair or create a new one for SSH access.

2. Set Up SSH Access

- **SSH Key:** Ensure you have the `*.pem` file for the key pair.
- **SSH Command:** Use the command below to log into your instance. Replace `ec2-user` with the appropriate user (e.g., `ubuntu` for Ubuntu AMIs) and `instance_ip` with your instance's public IP.

```
sh
```

 Copy code

```
ssh -i /path/to/your-key.pem ec2-user@instance_ip
```

3. Clone the Shell Script

- **Install Git** (if not already installed):

```
sh
```

 Copy code

```
sudo apt-get update  
sudo apt-get install git
```

- **Install Git** (if not already installed):

```
sh
```

 Copy code

```
sudo apt-get update  
sudo apt-get install git
```

- **Clone Repository:** Use the `git clone` command to clone the repository containing the shell script.

```
sh
```

 Copy code

```
git clone <repository_url>  
cd <repository_directory>
```

4. Configure Environment Variables

- **Export GitHub Username:**

```
sh
```

 Copy code

```
export USERNAME=<your_github_username>
```

- Generate API Token:
 - Go to GitHub: Log into your GitHub account.
 - Navigate to Developer Settings: Go to `Settings` > `Developer settings`.
 - Generate Token: In `Personal access tokens`, generate a new token with appropriate permissions.
- Export API Token:

```
sh
```

 Copy code

```
export TOKEN=<your_github_token>
```

5. Execute the Script

- Run the Shell Script: Execute the script that checks repository access.

```
sh
```

 Copy code

```
./list_users.sh
```

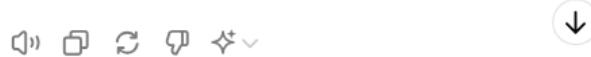
Explanation of Script Execution

- **Cloning the Script:** The script is cloned onto the EC2 instance to ensure it can run in an isolated and controlled environment.
- **Setting Environment Variables:** These variables are used by the script to authenticate with GitHub and perform API calls.
- **Running the Script:** The script uses the GitHub API to fetch information about repository access, simplifying the process of monitoring access permissions.

Improving the Script

- **Error Handling:** Ensure the script handles errors gracefully, such as invalid tokens or network issues.
- **Logging:** Implement logging to keep track of script execution and any issues encountered.
- **Security:** Ensure the API token is stored securely and minimize its permissions to reduce security risks.

This automated approach using a shell script and AWS EC2 instance streamlines the process of managing and monitoring GitHub repository access, making it more efficient and less error-prone.



26:30 mints video:

Answer

Let's break down the process of executing the shell script to list users who have access to a GitHub repository, including the necessary command-line arguments, permissions, and dependencies like `jq`. We'll also discuss the significance of these steps and how to adapt the script for other repositories.

Executing the Shell Script

Prerequisites

Before running the script, you need to ensure that:

1. **Environment Variables:** You have exported your GitHub username and personal access token. This is crucial for authenticating API requests.

```
bash
```

```
export GITHUB_USERNAME="your_github_username"  
export GITHUB_TOKEN="your_personal_access_token"
```



2. **Command-Line Arguments:** The script requires two command-line arguments:

- **Organization Name:** The name of the organization that owns the repository.
- **Repository Name:** The name of the repository you want to check.

For example, if your organization is `devops-by-examples` and the repository is `python`, you would run the script like this:

```
bash
```

```
./list_users.sh devops-by-examples python
```



Setting Script Permissions

If you encounter a "permission denied" error when trying to execute the script, you need to modify the script's permissions to make it executable. The command `chmod 777` grants read, write, and execute permissions to the user, group, and others, but it is generally not recommended for production environments due to security risks. Instead, a more secure permission setting would be `chmod 755`, which allows the owner to read, write, and execute, while the group and others can only read and execute.

```
bash
```

```
chmod 755 list_users.sh
```



Installing Dependencies

The script requires the `jq` command-line tool to parse JSON responses from the GitHub API. If you haven't installed `jq`, you can do so with the following command:

```
bash
```

```
sudo apt install jq -y
```



Running the Script

Once the script is executable and dependencies are installed, you can run it with the required arguments:

```
bash
```

```
./list_users.sh devops-by-examples python
```



Understanding the Output

When you execute the script, it will output a list of users who have access to the specified repository. For example, the output might look like this:

```
text
```

```
Collaborators with access to the repository:
```

```
Moit
```



In this case, the script shows that "Moit" is an outside collaborator with access to the repository. The reason other users (like yourself or other admins) are not listed is that the script is designed to show only those who are not owners or admins of the repository.

Why Only Some Users Are Displayed

- **Admin Privileges:** Users who are admins of the repository may not appear in the output. This is intentional, as the script is likely filtering out users with higher privileges, focusing instead on outside collaborators or users with specific roles.
- **Access Levels:** The script is designed to show a clear distinction between different types of access. Owners and admins typically have broader access, while outside collaborators may have limited permissions.

Adapting the Script for Other Repositories

If you want to check access for other repositories, simply provide the appropriate organization and repository names when executing the script. For example:

bash

```
./list_users.sh another-org another-repo
```



Conclusion

By automating the process of checking who has access to a GitHub repository using a shell script, you can save time and enhance security. This approach allows you to efficiently manage user access, especially when dealing with multiple repositories across different teams.

Next Steps

- **Enhance the Script:** Consider adding functionality to revoke access or to list all users, including admins, if needed.
- **Explore API Documentation:** Familiarize yourself with the GitHub API documentation to understand other endpoints that could be useful for your tasks.
- **Testing:** Test the script in a safe environment before deploying it in production to ensure it meets your needs.

If you have any further questions or need clarification on specific aspects of the script or its execution, feel free to ask!

Share Rewrite

...



Detailed Script Execution and Explanation

Now that you have exported the GitHub API token and username, it's time to execute the script and understand its purpose and functionality in detail. The script will provide a list of users who have access to a specific GitHub repository within an organization. This process involves running the script with command line arguments specifying the organization and repository names.

Steps to Execute the Script

1. **Set Up Execution Permissions:** Before running the script, ensure it has the necessary execution permissions.
2. **Install Dependencies:** The script requires `jq` for processing JSON data.
3. **Run the Script:** Execute the script with the appropriate command line arguments.

Example Execution

1. **Provide Execution Permissions:**

```
sh
```

Copy code

```
chmod 777 list_users.sh
```

The `chmod 777` command grants read, write, and execute permissions to the script for the owner, group, and others. This ensures the script can be executed without permission issues.

The `chmod 777` command grants read, write, and execute permissions to the script for the owner, group, and others. This ensures the script can be executed without permission issues.

2. Install jq:

```
sh Copy code
sudo apt-get install jq -y
```

`jq` is a lightweight and flexible command-line JSON processor. The script uses `jq` to parse and format JSON data received from the GitHub API.

3. Execute the Script:

```
sh Copy code
./list_users.sh devopsbyexamples python
```

Here, `devopsbyexamples` is the organization name, and `python` is the repository name.

Understanding the Script

Let's break down the script to understand its functionality:

```
sh
Copy code

#!/bin/bash

# Check if the correct number of arguments are provided
if [ "$#" -ne 2 ]; then
    echo "Usage: $0 <organization> <repository>"
    exit 1
fi

# Assign command line arguments to variables
ORG=$1
REPO=$2

# GitHub API URL
API_URL="https://api.github.com/repos/$ORG/$REPO/collaborators"

# Fetch collaborators from GitHub API
RESPONSE=$(curl -s -u $USERNAME:$TOKEN $API_URL)

# Check if the response contains an error
if echo "$RESPONSE" | jq -e 'has("message")' > /dev/null; then
    echo "Error: $(echo $RESPONSE | jq -r '.message')"
    exit 1
fi

# Parse and display collaborators
echo "Collaborators with access to $ORG/$REPO:"
echo "$RESPONSE" | jq -r '.[] | .login'
```

Script Explanation

1. Argument Check:

```
sh Copy code
if [ "$#" -ne 2 ]; then
    echo "Usage: $0 <organization> <repository>"
    exit 1
fi Explain
```

The script checks if exactly two command line arguments are provided. If not, it prints a usage message and exits.

2. Variable Assignment:

```
sh Copy code
ORG=$1
REPO=$2 Explain
```

The script assigns the first and second command line arguments to `ORG` and `REPO` variables, respectively.

3. GitHub API URL:

```
sh Copy code
API_URL="https://api.github.com/repos/$ORG/$REPO/collaborators" Explain
```

Constructs the GitHub API URL for fetching collaborators of the specified repository.

4. Fetch Collaborators:

```
sh Copy code
RESPONSE=$(curl -s -u $USERNAME:$TOKEN $API_URL) Explain
```

Uses `curl` to send an authenticated request to the GitHub API. The `-u \$USERNAME:\$TOKEN` option provides the username and token for authentication. The response is stored in the `RESPONSE` variable.

5. Error Handling:

```
sh Copy code
if echo "$RESPONSE" | jq -e 'has("message")' > /dev/null; then
    echo "Error: $(echo $RESPONSE | jq -r '.message')"
    exit 1
fi Explain
```

Checks if the API response contains an error message. If an error is found, it prints the error message and exits.

6. Parse and Display Collaborators:

```
sh Copy code
echo "Collaborators with access to $ORG/$REPO:"
echo "$RESPONSE" | jq -r '.[].login'
```

Uses `jq` to parse the JSON response and extract the login names of the collaborators. These names are then displayed.

Explanation of Output

- **User Roles:** The script differentiates between regular collaborators and repository owners. Owners may not be listed if the script is designed to show only collaborators with specific permissions.
- **Outside Collaborators:** Users who are not part of the organization but have been granted access to the repository are shown distinctly.

Use Case in DevOps

As a DevOps engineer, automating the process of checking repository access is crucial for maintaining security and ensuring that only authorized users have access to sensitive code. This script provides a quick and reliable way to audit repository access across an organization, helping to identify and address any potential security concerns.

Improving the Script

1. **Detailed Logging:** Add logging to capture script execution details, which can help in debugging and auditing.
2. **Permission Filtering:** Extend the script to filter collaborators based on their permissions (e.g., read, write, admin).
3. **Security Enhancements:** Store the API token securely, possibly using environment variables or a secrets management service.

By following these steps and understanding the script, you can efficiently manage GitHub repository access and ensure that your organization's code repositories are secure and well-managed.

30: 35:56 mints video:

Answer

Sure, let's break down the concepts in a more detailed and easy-to-understand way, using examples where appropriate.

Changing Organization and Repository Names

Imagine you have a shell script that can list all the people who have access to a specific GitHub repository. To use this script for a different repository, you simply need to modify two things:

1. **Organization Name:** This is the name of the GitHub organization that owns the repository. For example, if the repository is owned by an organization called "MyCompany", you would replace the organization name in the script with "MyCompany".
2. **Repository Name:** This is the name of the specific repository you want to check. For instance, if the repository is called "MyProject", you would replace the repository name in the script with "MyProject".

By changing these two values in the script, you can use it to check access for any repository you have permission to view.

Access Control and Permissions

GitHub has a system that controls who can access repositories and what actions they can perform. This is called access control.

1. **User Interface:** When you log in to GitHub and view a repository, you see the user interface. If you have the necessary permissions, you can see the repository settings and view the list of people who have access.
2. **API:** GitHub also provides an Application Programming Interface (API), which allows you to interact with GitHub programmatically. The API has the same access control rules as the user interface.

For example, if you try to use the API to view the list of collaborators for a repository, but you don't have permission to view the repository settings, the API will not return any information. This is because the API respects the same access control rules as the user interface.

Practical Application

Let's say you work at a company called "MyCompany", and your team uses GitHub to manage your code repositories. As a DevOps engineer, you are responsible for managing access to these repositories.

To use the shell script to list the collaborators for a repository in your company, you would:

1. **Open a terminal** on your computer.
2. **Navigate to the directory** where the shell script is located.
3. **Run the script**, providing the organization name (in this case, "MyCompany") and the repository name (e.g., "MyProject") as arguments.

The script would then use the GitHub API to retrieve the list of people who have access to the specified repository.

Creating a GitHub Organization

If you want to practice using the shell script without affecting your company's repositories, you can create your own GitHub organization. Here's how:

1. **Log in to your GitHub account**.
2. **Click on your profile picture** in the top-right corner and select "Your organizations".
3. **Click on the "New organization" button**.
4. **Follow the prompts** to create a new organization, providing a name and other details as requested.

Once you have your own organization, you can create repositories within it and add collaborators to test the shell script.

Understanding the Shell Script

Let's take a closer look at the structure of the shell script:

1. **Shebang:** The first line of the script, `#!/bin/bash`, tells the system that this script should be run using the Bash shell.
2. **API URL:** The script constructs the GitHub API URL, which is used to make requests to the GitHub API. The base URL is `https://api.github.com`, and specific

2. **API URL:** The script constructs the GitHub API URL, which is used to make requests to the GitHub API. The base URL is `https://api.github.com`, and specific endpoints are added based on the action you want to perform.
3. **Environment Variables:** The script uses environment variables to store sensitive information, such as your GitHub username and a personal access token. This allows you to keep this information separate from the script itself, making it more secure.
4. **Command-Line Arguments:** When you run the script, you provide the organization name and repository name as arguments. The script then uses these values to dynamically form the API endpoint URL.
5. **Functions:** The script is divided into functions, each of which performs a specific task. For example, one function might be responsible for forming the API request, while another function processes the response.
6. **cURL Command:** The script uses the `curl` command to make HTTP requests to the GitHub API. This command is commonly used in shell scripts for interacting with web services.
7. **Response Handling:** After making the API request, the script processes the JSON response using a tool called `jq`. This allows the script to extract relevant information, such as the list of collaborators.

7. **Response Handling:** After making the API request, the script processes the JSON response using a tool called `jq`. This allows the script to extract relevant information, such as the list of collaborators.

By understanding how these components work together, you can modify the script to suit your specific needs and explore other ways to automate tasks related to GitHub repository management.



Let's break down the concepts in your script and process in a detailed, deep, and easy-to-understand manner:

Changing the Organization and Repository

Concept: The script can be used to get a list of users who have access to any repository within an organization by changing the organization name and repository name.

Explanation:

- The script takes two command line arguments: the organization name and the repository name.
- By providing different values for these arguments, you can get the access details for any repository you have permission to view.

Example of Changing the Repository

1. Scenario:

- You want to check who has access to the `kubernetes/kubernetes` repository.
- However, you made a typo in the repository name.

2. Correcting the Typo:

```
sh Copy code
./list_users.sh kubernetes kubernetes
```

3. Error Handling:

- If you don't have access to the repository, the script will fail.
- This is because you cannot perform an action through the API that you cannot perform through the GitHub UI.

Access and Permissions

Concept: The script uses the GitHub API to fetch data, which requires proper access permissions.

Explanation:

- If you cannot see the settings tab for a repository on GitHub, you won't be able to get the access list through the API.
- The script will show an error if you do not have the necessary permissions.

Practicing the Script

Concept: You can practice using the script with your own repositories.

Explanation:

- If you are in an organization, you can run the script without issues as it only lists users.
- To practice independently, you can create a new organization and repositories on GitHub.

Creating an Organization

1. Steps:

- Go to GitHub and navigate to `Settings` > `Organizations`.
- Click on `New Organization`.
- Follow the prompts to create a free organization.
- Provide an organization account name and other details.

2. Use Case:

- Within your new organization, create repositories and add collaborators.
- This simulates an open-source organization or a company's GitHub setup.

Understanding the Script

1. Script Start:

- Always starts with a shebang (`#!/bin/bash`) to specify the shell.

2. Exporting Sensitive Information:

- Username and token are exported to avoid hardcoding sensitive information.
- This ensures security and flexibility.

3. Command Line Arguments:

- The script reads the organization and repository names from the command line arguments.
- These arguments are accessed using  (first argument) and `\\$2` (second argument).

Script Structure

1. API URL Construction:

```
sh Copy code
API_URL="https://api.github.com/repos/$ORG/$REPO/collaborators"
```

- Constructs the API URL dynamically using the provided organization and repository names.

2. Authentication:

```
sh Copy code
RESPONSE=$(curl -s -u $USERNAME:$TOKEN $API_URL)
```

- Uses `curl` to send an authenticated request to the GitHub API.

3. Error Handling:

```
sh Copy code
Explain
if echo "$RESPONSE" | jq -e 'has("message")' > /dev/null; then
  echo "Error: $(echo $RESPONSE | jq -r '.message')"
  exit 1
fi
```

- Checks if the API response contains an error message.

4. Parsing JSON Response:

```
sh
echo "$RESPONSE" | jq -r '.[] | .login'
```

 Copy code

- Uses `jq` to parse the JSON response and extract the `login` field of each collaborator.

Example Script

Here is the script with comments explaining each section:

```
sh
#!/bin/bash

# Check if the correct number of arguments are provided
if [ "$#" -ne 2 ]; then
    echo "Usage: $0 <organization> <repository>"
    exit 1
fi

# Assign command line arguments to variables
ORG=$1
REPO=$2

# GitHub API URL
API_URL="https://api.github.com/repos/$ORG/$REPO/collaborators"
```

 Explain |  Copy



```
# GitHub API URL
API_URL="https://api.github.com/repos/$ORG/$REPO/collaborators"

# Fetch collaborators from GitHub API
RESPONSE=$(curl -s -u $USERNAME:$TOKEN $API_URL)

# Check if the response contains an error
if echo "$RESPONSE" | jq -e 'has("message")' > /dev/null; then
    echo "Error: $(echo $RESPONSE | jq -r '.message')"
    exit 1
fi

# Parse and display collaborators
echo "Collaborators with access to $ORG/$REPO:"
echo "$RESPONSE" | jq -r '.[] | .login'
```

Enhancing the Script

1. Detailed Logging:

- Add logging to capture script execution details for debugging.

```
sh
LOG_FILE="script.log"
echo "$(date): Executing script for $ORG/$REPO" >> $LOG_FILE
```

2. Filter Collaborators by Permissions:

- Modify the script to filter collaborators based on their permissions (e.g., `push` access).

```
sh
```

 Copy code

```
echo "$RESPONSE" | jq -r '.[] | select(.permissions.push == true) | .login'
```

3. Secure Handling of API Token:

- Store the API token securely, possibly using a secrets management service or encrypted environment variables.

By understanding these concepts and steps, you can efficiently manage GitHub repository access and enhance the script for better functionality and security.



36:56 - 41:42

Let's break down the concepts from your explanation regarding the shell script that interacts with the GitHub API to list users who have access to a repository. We will cover the structure of the script, the use of functions, the handling of API requests, and the processing of JSON responses.

Overview of the Shell Script

The shell script is designed to interact with the GitHub API to retrieve a list of users who have access to a specific repository. It does this by forming a cURL command to make an API request, processing the JSON response, and filtering the results to display relevant information.

Key Concepts

1. Functions in Shell Scripts:

- Functions are used to organize code into reusable blocks. In this script, there are at least two functions:
 - One for forming the cURL command.
 - Another for handling input and output.

Example:

```
bash
```

```
❶ Explain | ⌂ function create_curl_command {
    # Code to form the cURL command
}

function print_output {
    # Code to print the output
}
```



2. Forming the cURL Command:

- The cURL command is constructed to make an HTTP request to the GitHub API. This command includes the API endpoint and any necessary headers for authentication.

Example:

```
bash
```

```
curl -H "Authorization: token $GITHUB_TOKEN" -H "Accept:
application/vnd.github.v3+json" "$API_URL"
```



3. API Endpoints:

- The API endpoint specifies the resource you want to access. For example, to list collaborators for a repository, the endpoint might look like this:

text

```
https://api.github.com/repos/{owner}/{repo}/collaborators
```



4. Input and Output Handling:

- The script takes command-line arguments (such as the organization name and repository name) and uses them to form the API request.
- The output of the script is printed to the terminal, showing the users who have access to the repository.

JSON Responses

1. Understanding JSON:

- JSON (JavaScript Object Notation) is a lightweight data interchange format that is easy for humans to read and write and easy for machines to parse and generate. Most APIs, including GitHub's, return data in JSON format.

2. Using `jq` for JSON Processing:

- The `jq` command-line tool is used to parse and filter JSON data. It allows you to extract specific fields from the JSON response.

Example:

bash

```
echo "$response" | jq '.[] | select(.permissions.admin = false) | .login'
```



3. Filtering for Specific Users:

- In the script, the goal is to list users who have read access but are not admins. The script checks the `permissions` field in the JSON response to determine if a user is an admin.

Example:

- If the JSON response includes:

```
json
Explain {  
    "login": "moit",  
    "permissions": {  
        "admin": false,  
        "pull": true  
    }  
}
```

- The script filters out users where `permissions.admin` is `true`.

Error Handling

1. Handling API Errors:

- If the script fails to retrieve data (e.g., due to insufficient permissions), it will return an error message. This is important for debugging and understanding what went wrong.

2. Example of Error:

- If you try to access a repository for which you do not have permission, the API will return an error message indicating that the request was unauthorized.

Example of the Complete Process

1. Setting Up the Environment:

- Before running the script, you need to set your GitHub username and personal access token as environment variables.

bash

```
export GITHUB_USERNAME="your_github_username"
export GITHUB_TOKEN="your_personal_access_token"
```



2. Running the Script:

- Execute the script with the organization and repository name as arguments.

bash

```
./list_users.sh devops-by-examples python
```



3. Output:

- The script will print the usernames of collaborators who have read access to the repository, excluding those who have admin access.

Conclusion

This shell script is a powerful tool for managing access to GitHub repositories. By understanding how to form API requests, process JSON responses, and filter for specific user permissions, you can effectively automate the task of listing users with access to your repositories.

Next Steps

- **Experiment with the Script:** Try modifying the script to include additional functionality, such as listing all users regardless of their permissions.
- **Learn More About APIs:** Familiarize yourself with other GitHub API endpoints to expand your automation capabilities.
- **Practice Using `jq`:** Explore the `jq` tool further to become proficient in parsing and manipulating JSON data.

If you have any further questions or need clarification on specific aspects, feel free to ask!

 Share  Rewrite





Let's break down and explain each concept in detail and easy to understand:

1. Function Breakdown

Forming the `curl` Command

- **Concept:** One function is responsible for forming the `curl` command to make an API request.
- **Explanation:**
 - **Purpose:** This function constructs the complete `curl` command needed to interact with the GitHub API.
 - **Details:**
 - It builds the API URL and includes any necessary parameters.
 - The URL typically includes the endpoint to fetch data (e.g., repository collaborators).

Executing the `curl` Command

- **Concept:** Another function handles executing the `curl` command and processing its output.
- **Explanation:**
 - **Purpose:** This function runs the `curl` command and captures the JSON response from GitHub.



Executing the `curl` Command

- **Concept:** Another function handles executing the `curl` command and processing its output.
- **Explanation:**
 - **Purpose:** This function runs the `curl` command and captures the JSON response from GitHub.
 - **Details:**
 - Executes the command that was formed by the first function.
 - GitHub APIs usually return data in JSON format.

2. Input and Output Handling

Handling Inputs

- **Concept:** The script takes input parameters such as organization name and repository name.
- **Explanation:**
 - **Purpose:** These parameters specify which repository's access list to fetch.
 - **Details:**
 - Inputs are passed as command-line arguments to the script.
 - These arguments are used to form the API URL dynamically.

Printing Output

- **Concept:** The final function is responsible for processing and printing the output.
- **Explanation:**
 - **Purpose:** This function prints the results of the `curl` command in a user-friendly format.
 - **Details:**
 - The script uses tools like `jq` to parse the JSON response and extract relevant information.

3. Using `jq` for JSON Processing

What is `jq`?

- **Concept:** `jq` is a command-line tool for parsing and manipulating JSON data.
- **Explanation:**
 - **Purpose:** Extract specific fields from the JSON output.
 - **Details:**
 - Converts the raw JSON into a readable format by filtering out only the necessary data.

Example of JSON Processing

- **Concept:** Filtering JSON data to get specific information, such as usernames.
- **Explanation:**
 - **Purpose:** To display only the usernames of collaborators who have specific permissions.
 - **Details:**
 - The script extracts the `login` field from the JSON response using `jq`.
 - This is useful to filter out unnecessary details, like admin users.
↓

4. Error Handling and Permissions

Handling API Errors

- **Concept:** Checking for errors in the API response.
- **Explanation:**
 - **Purpose:** To handle cases where the API request fails or returns an error.
 - **Details:**
 - The script checks if the response contains an error message.
 - Displays an error message if something goes wrong.

Permissions and Filtering

- **Concept:** Filtering collaborators based on their permissions.
- **Explanation:**
 - **Purpose:** To include or exclude collaborators based on their access level.
 - **Details:**
 - JSON data includes a `permissions` field indicating whether a user is an admin or has read/write access.
↓
- The script filters out users based on these permissions (e.g., showing only users who can read but not administer the repository).

5. Script Example

Here's a simplified example of the script with explanations:

```
sh Copy code
Explain

#!/bin/bash

# Check if correct number of arguments are provided
if [ "$#" -ne 2 ]; then
    echo "Usage: $0 <organization> <repository>"
    exit 1
fi

# Assign arguments to variables
ORG=$1
REPO=$2

# Construct the API URL
API_URL="https://api.github.com/repos/$ORG/$REPO/collaborators"

# Execute the curl command and get JSON response
RESPONSE=$(curl -s -u $USERNAME:$TOKEN $API_URL)

# Check for errors in the response
if echo "$RESPONSE" | jq -e 'has("message")' > /dev/null; then
    echo "Error: $(echo $RESPONSE | jq -r '.message')"
    exit 1
fi

# Extract and print usernames of collaborators who can read the repo
echo "Collaborators with read access to $ORG/$REPO:" ↓
echo "$RESPONSE" | jq -r '.[].select(.permissions.pull == true) | .login'
```

Summary

1. Function Separation:

- One function forms the `curl` command.
- Another function executes the command and processes the output.

2. Handling Inputs and Outputs:

- Inputs are provided as command-line arguments.
- Outputs are processed and printed using `jq`.

3. Using `jq`:

- Parses JSON data to extract specific fields.
- Filters data based on permissions or other criteria.

4. Error Handling:

- Checks if the API response contains errors.
- Provides meaningful error messages to the user.

By understanding these concepts, you can effectively use and modify the script to manage and retrieve data from GitHub repositories based on your needs.



To understand the output of the script, let's assume some example scenarios. I'll simulate the output for different situations: successful execution with valid inputs and a scenario where an API error occurs.

Scenario 1: Successful Execution

Assumptions:

- Organization: `devops-by-examples`
- Repository: `python`
- User: `username` with the API token `token`

Script Execution:

```
sh
```

Copy code

```
./list_users.sh devops-by-examples python
```

Output:

```
plaintext
```

Copy code

```
Collaborators with read access to devops-by-examples/python:
```

```
moit
```

Explanation:

- The script fetches the list of collaborators who have read access to the repository `python` within the organization `devops-by-examples`.
- The output lists only the usernames of collaborators who can read the repository, excluding admins.

Scenario 2: API Error

Assumptions:

- Organization: `kubernetes`
- Repository: `kubernetes`
- User: `username` with the API token `token`

Script Execution:

```
sh Copy code
./list_users.sh kubernetes kubernetes
```

Output:

```
plaintext Copy code
Error: Not Found
```

Explanation:

- The script attempted to fetch data from the GitHub API for the repository `kubernetes` within the organization `kubernetes`.
- If the repository or organization does not exist or the user does not have access to view it, GitHub returns an error message.
- The script captures this error message and prints it.

Explanation of Outputs

1. Successful Execution:

- The script successfully constructs and executes the `curl` command.
- It receives a JSON response from GitHub.
- `jq` processes this JSON to extract and print usernames of users with read permissions.

2. API Error:

- If the API request fails due to reasons like the repository not existing, permission issues, or incorrect details, the script detects this error.
- The error message is printed to indicate the problem.

Example Outputs from `jq`

JSON Response Sample:

Here's a sample JSON response for a successful API call:

```
json
[{"login": "moit",
 "id": 123456,
 "node_id": "MDQ6VXNlcjEyMzQ1Ng==",
 "avatar_url": "https://avatars.githubusercontent.com/u/123456?v=4",
 "gravatar_id": "",
 "url": "https://api.github.com/users/moit",
 "html_url": "https://github.com/moit",
 "followers_url": "https://api.github.com/users/moit/followers",
 "following_url": "https://api.github.com/users/moit/following{/other_user}",
 "gists_url": "https://api.github.com/users/moit/gists{/gist_id}",
 "starred_url": "https://api.github.com/users/moit/starred{/owner}{/repo}",
 "subscriptions_url": "https://api.github.com/users/moit/subscriptions",
 "organizations_url": "https://api.github.com/users/moit/orgs",
 "repos_url": "https://api.github.com/users/moit/repos",
 "events_url": "https://api.github.com/users/moit/events{/privacy}",
 "received_events_url": "https://api.github.com/users/moit/received_events",
 "type": "User",
 "site_admin": false,
 "permissions": {
   "admin": false,
   "push": false,
   "pull": true
 }
}]
```

Using `jq`:

To get only the usernames:

```
sh
```

 Copy code

```
echo "$RESPONSE" | jq -r '.[] | select(.permissions.pull == true) | .login'
```

Output:

```
plaintext
```

 Copy code

```
moit
```

This `jq` command filters the JSON data to show only the usernames of collaborators with read access (``permissions.pull == true``).



explain each attributes, what is the purpose?

Command:

```
sh
```

 Copy code

```
echo "$RESPONSE" | jq -r '.[] | select(.permissions.pull == true) | .login'
```

Answer is:

The `jq` command provided processes the JSON response from the GitHub API to filter and extract specific information. Let's break down each part of the command:

Command:

```
sh
echo "$RESPONSE" | jq -r '.[] | select(.permissions.pull == true) | .login'
```

 Copy code

Components Breakdown:

1. `echo "\$RESPONSE"`:

- This part of the command outputs the content of the `\$RESPONSE` variable, which is expected to contain the JSON data returned from the GitHub API. The JSON data is passed through a pipe (`|`) to the `jq` command for further processing.

2. `jq -r`:

- `jq` is a command-line tool for processing JSON data.
- The `-r` option stands for "raw output". It tells `jq` to output raw strings rather than JSON-encoded strings. This is useful for extracting plain text values, such as usernames.

3. `.[]`:



- This filters the input JSON array by iterating over each element. In this case, it processes

3. `.[]`:

- This filters the input JSON array by iterating over each element. In this case, it processes each object in the array returned by the GitHub API.

4. `select(.permissions.pull == true)`:

- `select` is a `jq` function that filters objects based on a condition.
- `permissions.pull == true` is the condition being tested. It checks if the `pull` permission is set to `true` in the `permissions` object of each JSON object.
- If the condition is true, the object is included in the output; otherwise, it is filtered out.

5. `login`:

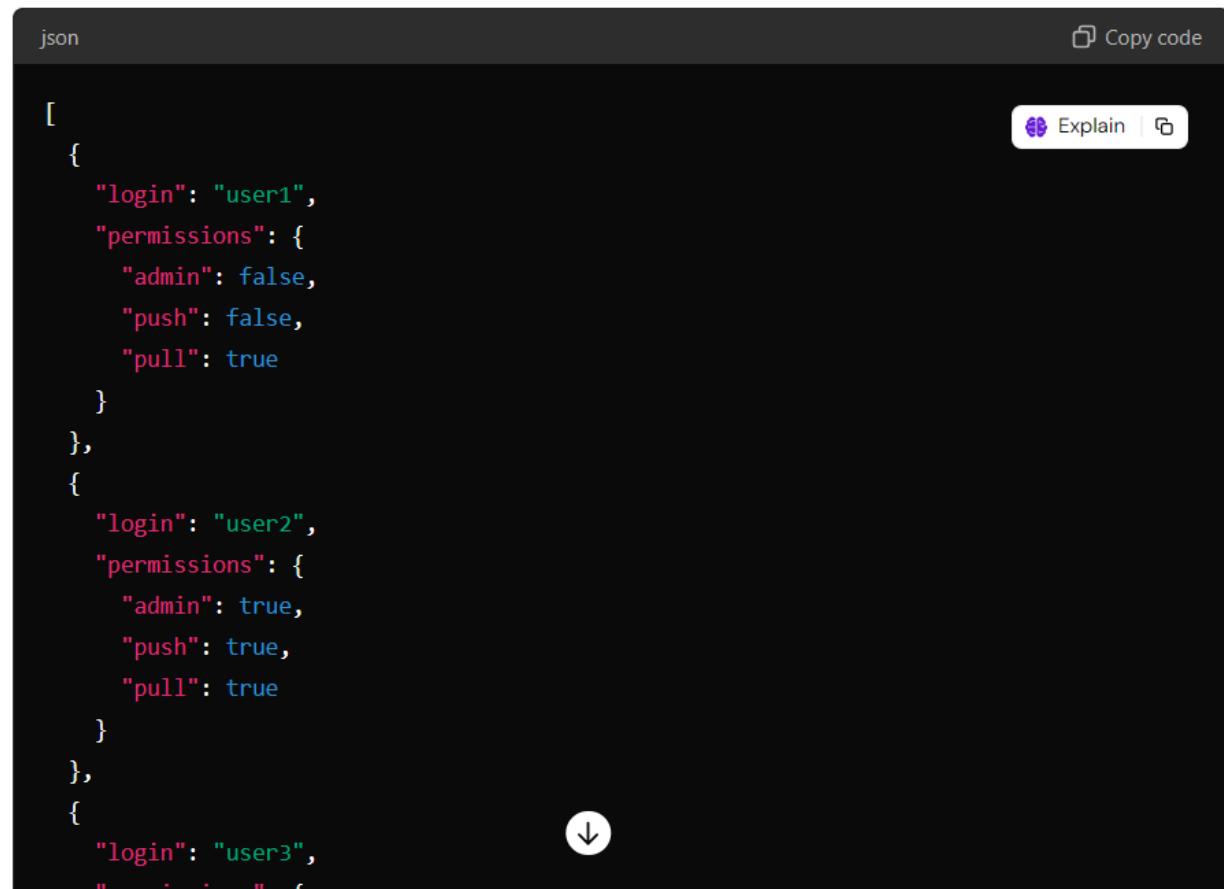
- After filtering the objects, `login` extracts the value of the `login` attribute from each remaining object. The `login` attribute represents the username of the collaborator.

Purpose of Each Attribute:

- `echo "\$RESPONSE"`:
 - Outputs the JSON data so it can be processed by `jq`.
- `jq -r`:
 - Processes the JSON data and outputs raw strings for easier readability and further use.
- `.[[]]`:
 - Iterates over each object in the JSON array to apply filtering.
- `select(.permissions.pull == true)`:
 - Filters out only those objects where the user has read (pull) access to the repository. This helps to focus on users who can access the repository content but may not necessarily have admin or write permissions.
- `.login`:
 - Extracts the username of the users who meet the criteria. This provides a list of usernames with read access to the repository.

Example Scenario:

Assuming the JSON response from the GitHub API includes:



A screenshot of a JSON editor interface. The title bar says "json". In the top right corner, there are "Copy code" and "Explain" buttons. The main area contains the following JSON data:

```
[  
  {  
    "login": "user1",  
    "permissions": {  
      "admin": false,  
      "push": false,  
      "pull": true  
    }  
  },  
  {  
    "login": "user2",  
    "permissions": {  
      "admin": true,  
      "push": true,  
      "pull": true  
    }  
  },  
  {  
    "login": "user3",  
    "permissions": {  
      "admin": true,  
      "push": true,  
      "pull": true  
    }  
  }]
```

The JSON structure represents a list of three users. Each user object contains a "login" key with a value like "user1", and a "permissions" key which is itself an object containing "admin", "push", and "pull" keys with boolean values.

```
        "pull": true
    }
},
{
    "login": "user3",
    "permissions": {
        "admin": false,
        "push": false,
        "pull": false
    }
}
]
```

The `jq` command will output:

 Copy code

```
user1
user2
```

This is because `user1` and `user2` have `pull` permissions set to `true`, whereas `user3` does not.



42:00-49:00

Answer

Let's break down the concepts from your explanation about the shell script that interacts with the GitHub API to list users who have access to a repository. We will cover the functionality of the script, the use of `jq` for JSON processing, the structure of the script, and how to improve it.

Key Concepts in the Shell Script

1. Using `jq` for JSON Processing

`jq` is a powerful command-line tool for parsing JSON data. It allows you to extract specific fields from JSON objects easily.

- **Extracting Fields:** When you retrieve data from the GitHub API, it often comes in JSON format. For instance, if you want to get the value of a specific field, such as `permissions.admin`, you can use `jq` like this:

bash

```
jq '.permissions.admin'
```



- **Example:** If you have a JSON object that looks like this:

```
json
Explain | ⌂ { "login": "moit", "permissions": { "admin": false, "pull": true } }
```

You can extract the `admin` value with:

```
bash
echo '{"login": "moit", "permissions": {"admin": false, "pull": true}}' |
jq '.permissions.admin'
```

2. Filtering Users Based on Permissions

In your script, you want to filter out users based on their permissions. For example, if you only want to list users who have read access but are not admins, you can use the following logic:

- **Condition:** Check if `permissions.pull` is `true` and `permissions.admin` is `false`.
- **Example:**

```
bash
if [ "$(echo $user | jq '.permissions.pull')" = "true" ] && [ "$(echo $user | jq '.permissions.admin')" = "false" ]; then
    echo $user | jq '.login'
fi
```

3. Script Structure

The script is organized into functions, which helps keep the code clean and maintainable.

- **Function for API URL:** One function is responsible for forming the API URL based on the organization and repository names. This helps in keeping the logic modular.
- **Function for Output:** Another function handles the output, printing the usernames of collaborators based on the filtered results.

4. Error Handling and Helper Functions

To improve user experience and error handling, you can implement a helper function that checks if the required command-line arguments are provided.

- **Example of a Helper Function:**

```
bash
Explain ⌂      function check_arguments {
    if [ "$#" -ne 2 ]; then
        echo "Usage: $0 <organization> <repository>"
        exit 1
    fi
}
```

- **Calling the Helper Function:** At the beginning of your script, you can call this helper function to ensure the user provides the necessary arguments before proceeding:

```
bash
check_arguments "$@"
```

5. Comment Section

Adding comments at the beginning of your script is a good practice. It helps other users (or your future self) understand what the script does, what inputs are required, and how to use it.

- Example Comment Section:

```
bash

⠄ Explain | ⌂      #!/bin/bash
# Script to list users who have access to a specified
GitHub repository.
# Usage: ./list_users.sh <organization> <repository>
# Expects the following environment variables:
# GITHUB_USERNAME - Your GitHub username
# GITHUB_TOKEN - Your GitHub personal access token
```

6. Executing the Script

Once the script is set up, you can execute it with the organization and repository names as arguments:

```
bash
```

```
./list_users.sh devops-by-examples python
```



7. Iterating Over Multiple Repositories

If you have multiple repositories and want to list users for all of them, you can use a loop to iterate through a list of repositories:

```
bash
```

```
for repo in repo1 repo2 repo3; do
    ./list_users.sh devops-by-examples $repo
done
```



This allows you to automate the process of checking access for many repositories at once.

Conclusion

By understanding how to use `jq` for JSON processing, structuring your script with functions, implementing error handling, and documenting your code, you can create an efficient and user-friendly shell script to manage access to GitHub repositories. This not only saves time but also enhances security by ensuring that only the appropriate users have access.

Next Steps

- **Practice:** Create your own GitHub organization and repositories to test the script.
- **Enhance the Script:** Consider adding features such as logging or the ability to revoke access directly from the script.
- **Explore More APIs:** Familiarize yourself with other GitHub API endpoints to expand your automation capabilities.

If you have any further questions or need clarification on specific aspects, feel free to ask!

Let's break down the explanation and the concepts mentioned into detailed and easy-to-understand sections:

1. Purpose of JQ in Shell Scripts

JQ is a command-line tool used to parse and filter JSON data. Here's a detailed explanation of how JQ is used in shell scripting:

- **Filtering JSON Data:** JQ allows you to extract specific pieces of information from a JSON response. For instance, if you have a JSON array of user objects and you want to filter users based on their permissions, you can use JQ to achieve that.
- **Selecting Specific Fields:** You can choose which fields to include in the output by specifying the field name after a dot (`.`). For example, `.`login` extracts the login name of a user from a JSON object.

2. JQ Command Breakdown

The provided command is:

```
sh
```

 Copy code

```
echo "$RESPONSE" | jq -r '.[[]] | select(.permissions.pull == true) | .login'
```

Components:

- `echo "\$RESPONSE"`:
 - Outputs the content of the `"\$RESPONSE"` variable, which contains JSON data. This data is passed to `jq` for processing.
- `jq -r`:
 - `jq` processes JSON data.
 - `--raw-output` (raw output) option prints raw strings instead of JSON-encoded strings, making the output easier to use in shell scripts.
- `.[[]]`:
 - Iterates over each item in the JSON array. Each item is processed individually.
- `select(.permissions.pull == true)`:
 - Filters items where the `pull` permission is set to `true`. This means only users who have read access are selected.
- `login`:
 - Extracts the `login` field from each selected item. This field typically represents the username of the user.

3. Working with JSON Fields

In JSON, data is organized into key-value pairs. Here's how you interact with fields:

- **Accessing Nested Fields:** Use `.` to access fields. For example, `permissions.admin` accesses the `admin` field inside the `permissions` object.
- **Filtering Based on Conditions:** You can use conditions to filter data. For example, `select(.permissions.pull == true)` filters objects based on the `pull` permission.

4. Practical Use Case

Imagine you have a JSON response with a list of users and their permissions, like this:

```
json Copy code
[Explain ☰]
{
  {
    "login": "moit",
    "permissions": {
      "admin": false,
      "push": false,
      "pull": true
    }
  },
  {
    "login": "bavva".
  }
}
```

4. Practical Use Case

Imagine you have a JSON response with a list of users and their permissions, like this:

```
json
Copy code
[{"login": "moit", "permissions": {"admin": false, "push": false, "pull": true}}, {"login": "bavya", "permissions": {"admin": true, "push": true, "pull": true}}]
```

Explain | ⚙️

- Output Explanation:
 - `moit` has `pull` permission, so it appears in the output.
 - `bavya` also has `pull` permission but is an admin as well.

5. Adding Helper Functions and Error Handling

In shell scripts, helper functions and error handling improve usability and robustness:

- **Helper Function:** A function that provides assistance, such as validating input arguments or displaying usage instructions.

```
sh

helper_function() {
    if [ "$#" -ne 2 ]; then
        echo "Please execute the script with the required arguments: <owner> <repo>"
        exit 1
    fi
}
```

- **Usage:**
 - Check if the required number of arguments is provided.
 - If not, print an error message and exit.
- **Invoking Helper Function:**
 - Call the helper function at the beginning of your script to ensure the script is executed with the correct arguments.

```
sh

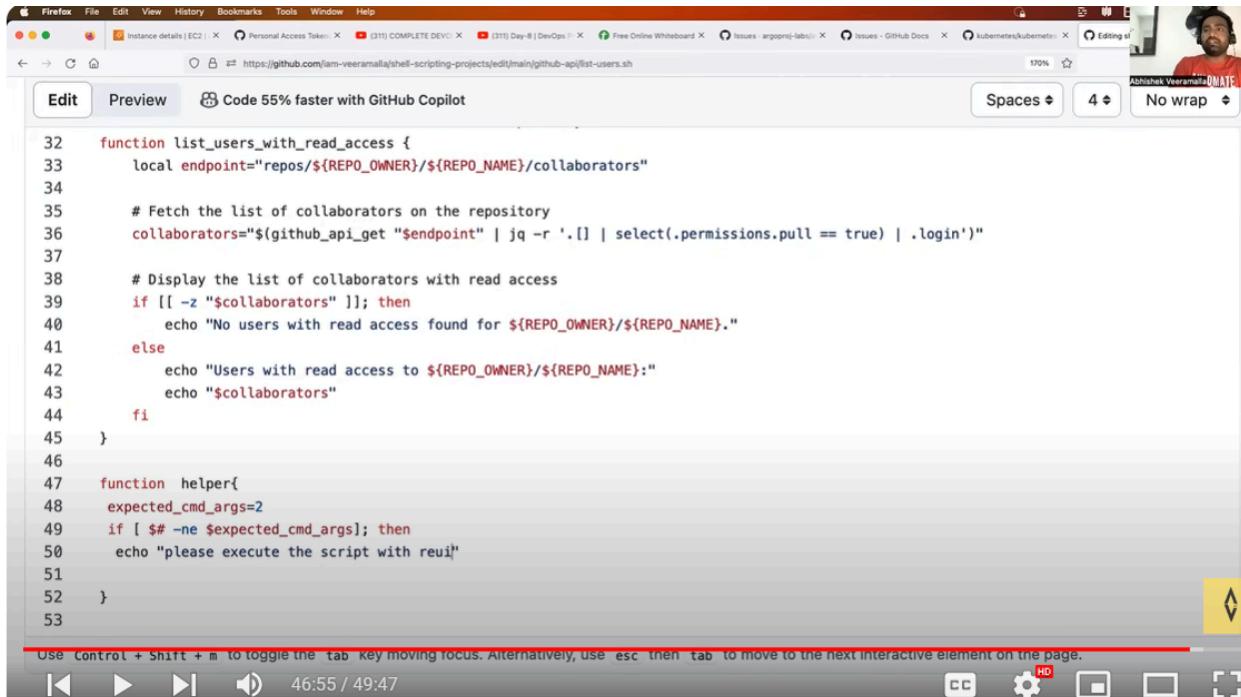
helper_function "$@"
↓
```

6. Enhancing the Script

- **Comments and Documentation:** Add comments to explain what the script does, the required parameters, and how to use it. This helps users understand and maintain the script.
- **Error Handling:** Implement checks to handle missing or incorrect arguments gracefully.
- **Automation:** If you have multiple repositories, use a loop to process each repository and extract relevant data.

```
sh
for repo in repo1 repo2 repo3; do
    ./your_script.sh owner "$repo"
done
```

By following these guidelines, you can make your shell scripts more user-friendly, maintainable, and effective.



The screenshot shows a GitHub code editor interface in a Firefox browser window. The URL is https://github.com/iam-veeramalla/shell-scripting-projects/edit/main/github-api/list-users.sh. The code editor has tabs for 'Edit' and 'Preview'. A GitHub Copilot icon is visible in the preview tab. The code itself is a shell script:

```
32 function list_users_with_read_access {
33     local endpoint="repos/${REPO_OWNER}/${REPO_NAME}/collaborators"
34
35     # Fetch the list of collaborators on the repository
36     collaborators=$(github_api_get "$endpoint" | jq -r '.[] | select(.permissions.pull == true) | .login')"
37
38     # Display the list of collaborators with read access
39     if [[ -z "$collaborators" ]]; then
40         echo "No users with read access found for ${REPO_OWNER}/${REPO_NAME}."
41     else
42         echo "Users with read access to ${REPO_OWNER}/${REPO_NAME}:"
43         echo "$collaborators"
44     fi
45 }
46
47 function helper{
48     expected_cmd_args=
49     if [ $# -ne $expected_cmd_args ]; then
50         echo "please execute the script with reui"
51     fi
52 }
```

At the bottom of the browser window, there are navigation controls (back, forward, search, etc.) and a status bar indicating '46:55 / 49:47'.

Purpose of the Script

This shell script is designed to list collaborators on a GitHub repository who have read access. It uses the GitHub API and `jq` for JSON processing. Additionally, it includes a helper function to check if the correct number of command-line arguments are provided.

Detailed Breakdown

1. Function: `list_users_with_read_access`

```
sh                                     Copy code

function list_users_with_read_access {
    local endpoint="repos/${REPO_OWNER}/${REPO_NAME}/collaborators"

    # Fetch the list of collaborators on the repository
    collaborators=$(github_api_get "$endpoint" | jq -r '.[] | select(.permissions.pull == true)')

    # Display the list of collaborators with read access
    if [[ -z "$collaborators" ]]; then
        echo "No users with read access found for ${REPO_OWNER}/${REPO_NAME}."
    else
        echo "Users with read access to ${REPO_OWNER}/${REPO_NAME}:"
        echo "$collaborators"
    fi
}
```

Explanation:

- `local endpoint="repos/\${REPO_OWNER}/\${REPO_NAME}/collaborators"`: Defines the API endpoint to get the list of collaborators for a specific repository.
- `collaborators="\$(github_api_get "\$endpoint" | jq -r '.[] | select(.permissions.pull == true) | .login')"`:
 - `github_api_get "\$endpoint"`: Calls a function (presumably defined elsewhere) to make a GET request to the GitHub API endpoint.
 - `jq -r '.[] | select(.permissions.pull == true) | .login'`: Uses `jq` to filter the JSON response, selecting only the collaborators who have pull (read) access and extracting their login names.
- `if [[-z "\$collaborators"]]; then ... else ... fi`: Checks if the `collaborators` variable is empty.
 - If empty, it prints a message indicating no users with read access.
 - If not empty, it prints the list of users with read access.

2. Function: `helper`

```
sh                                         Copy code

function helper {
    expected_cmd_args=2
    if [ $# -ne $expected_cmd_args ]; then
        echo "Please execute the script with required CMD args"
        exit 1
    fi
}
```

Explanation:

- `expected_cmd_args=2`: Sets the expected number of command-line arguments to 2.
- `if [\$# -ne \$expected_cmd_args]; then ... fi`: Checks if the number of provided command-line arguments (`\$#`) is not equal to the expected number.
 - If the number of arguments is incorrect, it prints an error message and exits the script with a status code of 1.

Main Script

```
sh
helper # call the helper function to check for correct number of arguments

# The rest of the script would go here
```

 Copy code

Explanation:

- `helper`: Invokes the `helper` function at the beginning of the script to ensure the correct number of command-line arguments are provided before proceeding.

Example Usage

Assuming the script is saved as `list_users.sh`, here is an example of how to use it:

1. Set Required Environment Variables:

```
sh
export REPO_OWNER="your-github-username"
export REPO_NAME="your-repo-name"
```

 Copy code

2. Run the Script:

```
sh
./list_users.sh arg1 arg2
```

 Copy code

- `arg1` and `arg2` represent the required command-line arguments (e.g., GitHub username and token).

Example Output

If there are users with read access:

```
bash                                         ⚒ Copy code

Users with read access to your-github-username/your-repo-name:
user1
user2
user3
```

If there are no users with read access:

```
bash                                         ⚒ Copy code

No users with read access found for your-github-username/your-repo-name.
```

Improving the Script

1. Add a Comment Section:

- Describe the purpose of the script.
- List the input parameters and environment variables required.
- Mention the owner of the script and contact information for issues.

2. Enhance Error Handling:

- Provide more detailed error messages.
- Include instructions on how to properly use the script.

3. Add Logging:

- Log actions and results to a file for troubleshooting and auditing purposes.

Example of Enhanced Script

```
sh Copy code

#!/bin/bash

# About: This script lists collaborators on a GitHub repository with read access.

# Inputs:
#   - REPO_OWNER: The owner of the repository
#   - REPO_NAME: The name of the repository
# Command-line arguments:
#   - arg1: Description of the first argument
#   - arg2: Description of the second argument
# Contact: [Your Contact Information]

function list_users_with_read_access {
    local endpoint="repos/${REPO_OWNER}/${REPO_NAME}/collaborators"

    # Fetch the list of collaborators on the repository
    collaborators=$(github_api_get "$endpoint" | jq -r '.[] | select(.permissions.pull == true)')

    # Display the list of collaborators with read access
    if [[ -z "$collaborators" ]]; then
        echo "No users with read access found for ${REPO_OWNER}/${REPO_NAME}." > /dev/null
    else
        echo "Users with read access to ${REPO_OWNER}/${REPO_NAME}:"
        echo "$collaborators"
    fi
}
```

```
# Display the list of collaborators with read access
if [[ -z "$collaborators" ]]; then
    echo "No users with read access found for ${REPO_OWNER}/${REPO_NAME}."
else
    echo "Users with read access to ${REPO_OWNER}/${REPO_NAME}:"
    echo "$collaborators"
fi
}

function helper {
    expected_cmd_args=2
    if [ $# -ne $expected_cmd_args ]; then
        echo "Please execute the script with required CMD args: arg1, arg2"
        exit 1
    fi
}

# Check if the correct number of command-line arguments are provided
helper "$@"

# Your script logic here
list_users_with_read_access
```

This enhanced script provides clearer guidance, error handling, and is easier to maintain.



Let's break down the working of each command or function:

1) `if [[-z "\$collaborators"]]; then`

- `[[` and `]]`: These are test brackets used in bash scripting to evaluate expressions. They are more powerful and safer compared to the single `[]` brackets.
- `-z`: This is a string comparison operator used to check if a string is empty.
- `"\$collaborators"`: This is the variable being checked. The double quotes around the variable ensure that the condition works correctly even if the variable is empty or contains spaces.

Explanation:

- The `if [[-z "\$collaborators"]]; then` command checks if the `collaborators` variable is empty.
- If the `collaborators` variable is empty (i.e., it contains no data), the script will execute the commands inside the `if` block. If `collaborators` contains data, the `else` block will be executed (if it exists).

Example:

```
sh
collaborators=""
if [[ -z "$collaborators" ]]; then
    echo "No collaborators found."
else
    echo "Collaborators found."
fi
```

Copy code

- Output: `No collaborators found.` (because the `collaborators` variable is empty)

2) `function helper { ... }`

This code defines a helper function that checks if the script is called with the correct number of command-line arguments.

Working of the Function:

```
sh                                     Copy code

function helper {
    expected_cmd_args=2 # Define how many arguments are expected
    if [ $# -ne $expected_cmd_args ]; then # Check if the number of arguments passed to the
        echo "Please execute the script with required CMD args" # Print an error message if the
        exit 1 # Exit the script with a status of 1, indicating an error
    fi
}
```

- `expected_cmd_args=2`: This sets the expected number of command-line arguments to 2.
- `#\$`: This special variable holds the number of command-line arguments passed to the script.
- `-ne`: This is a comparison operator that means "not equal."
- `exit 1`: This command exits the script immediately with a status code of 1, which typically indicates that an error has occurred.

Explanation:

- The function checks whether the number of command-line arguments provided when running the script matches the expected number (``expected_cmd_args``).
- If the number of arguments is not equal to the expected number, it prints an error message and exits the script.

Example:

```
sh Copy code

helper() {
    expected_cmd_args=2
    if [ $# -ne $expected_cmd_args ]; then
        echo "Please execute the script with required CMD args"
        exit 1
    fi
}

# Suppose you run the script with only one argument:
helper arg1
```

- Output: `"**Please execute the script with required CMD args**"` and the script exits with a status of 1.



3) `helper "\$@"`

Explanation:

- `helper "\$@"`: This calls the `helper` function and passes all the command-line arguments (`\$@`) to it.
- `"\$@"`: This special variable represents all the command-line arguments passed to the script, with each argument properly quoted to handle spaces or special characters.

Working:

- When `helper "\$@"` is executed, the `helper` function is invoked, and all the command-line arguments that were passed to the script are also passed to the `helper` function.
- The `helper` function then checks if the number of arguments is correct as per its logic.

Example:

Suppose the script is run with two arguments:

```
sh Copy code
./script.sh arg1 arg2
```

Inside the script:

```
sh Copy code
helper "$@"
↓
```

Suppose the script is run with two arguments:

```
sh
./script.sh arg1 arg2
```

 Copy code

Inside the script:

```
sh
helper "$@"
```

 Copy code

- Here, `\$#` would be 2 because there are two arguments (`arg1` and `arg2`).
- The `helper` function would not print an error and would allow the script to continue running.

Summary:

1. `if [[-z "\$collaborators"]]; then`: Checks if the `collaborators` variable is empty.
2. `function helper { ... }`: Defines a function to check if the correct number of command-line arguments are provided.
3. `helper "\$@"`: Calls the `helper` function, passing all command-line arguments to it, to verify if the correct number of arguments were provided when the script was run.



< 1/2 > ⟲ ⟳ ⌂ ⌃ ⌄ ⌅ ⟶