Devops

GITHUB API INTEGRATION WITH SHELL SCRIPTING

REAL-WORLD DEVOPS SCENARIO

Problem: You're working as a DevOps engineer in an organization where you have a Git repository. You want to keep checking the collaborators having access to this Git repo regularly.

Manual Process (Inefficient):

1. Log into GitHub
2. Go to repository
3. Click Settings
4. Click Collaborators
5. Take screenshot
6. Send to higher authority
7. Repeat daily ❌

Problem with manual process: Doing this on a day-to-day basis is a very complex task. Every time logging into the GitHub repository manually is tedious.

SOLUTION: AUTOMATE WITH SHELL SCRIPT

What we can do: Use shell script with GitHub integration to automate this task.

Requirements: Shell script will require GitHub integration. If we want to talk to an application, we can do it with the help of:

1. **API** (Application Programming Interface)
2. **CLI** (Command Line Interface)

Application can be anything - in our case, it is **GitHub**.

WHY USE API FOR GITHUB?

For GitHub, **API is more simple to use**.

Benefits:

- Write scripts and directly talk to GitHub
- Get information programmatically
- No need to use UI

WHAT IS API?

API = Application Programming Interface

Definition: API is where we get information from the application **programmatically** and **not via UI**.

Important Note: DevOps engineers will **NOT write** this API, but they will **consume** this API.

API REFERENCE DOCUMENTATION

Why needed? For every API, there is an **API reference documentation**. Without this documentation, DevOps engineers do not know how to make requests to this API.

What documentation provides:

- URL for the API
- Request format
- Response format
- Authentication methods
- Available endpoints

Where to find: GitHub Docs → API Documentation

Example: If you want to find pull requests:

1. Go to GitHub
2. Go to API documentation
3. Search for "Pull Request"
4. You will get the URL and command format

DEVOPS ENGINEER'S RESPONSIBILITIES

As a DevOps engineer:

1. Support multiple teams
2. Maintain lots of Git repos
3. Each team has one repo
4. For each repo, make sure proper access is granted
5. Write CI/CD pipelines
6. Monitor and maintain infrastructure

OUR TASK

Goal: Learn how to list people who have access to a repository using shell scripting.

Steps:

1. Check if wrong person has access
2. Revoke access if needed
3. Automate this with shell script

STEP 1: CREATE NEW EC2 INSTANCE

1. Go to AWS Console
2. Go to EC2 Dashboard
3. Click "Launch Instance"
4. Choose Ubuntu AMI
5. Choose t2.micro (free tier)
6. Create/select key pair
7. Launch instance

STEP 2: CONNECT TO EC2 INSTANCE

Using MobaXterm:

1. Open MobaXterm
2. Click "Session"
3. Select SSH

4. Paste EC2 public IP
5. Advanced SSH settings → Use private key (.pem file)
6. Click OK
7. Connection established ✅

STEP 3: CLONE THE GITHUB REPOSITORY

Command:

bash

```
git clone https://github.com/iam-veeramalla/shell-scripting-projects
```

Explanation:

- **git clone:** Command to clone/download repository
- **URL:** GitHub repository URL
- This downloads the entire repository to your local machine

Verify:

bash

```
ls
```

Output: You will see shell-scripting-projects directory

STEP 4: NAVIGATE TO THE PROJECT

Commands:

bash

```
cd shell-scripting-projects

ls

cd github-api

ls
```

Result: You will see list-users.sh file

STEP 5: SET UP GITHUB USERNAME

Command:

```bash
export username="Sushmita-Hubli"
```

Explanation:

- **export:** Creates environment variable

- **username:** Variable name

- **"Sushmita-Hubli":** Your GitHub username

Why export?

Makes the variable available to child processes (the script will use it)

STEP 6: CREATE GITHUB PERSONAL ACCESS TOKEN

What is a token?

When you log into GitHub, you provide username and password. But for APIs, we do not have password - we have something called **API token**.

Why token instead of password?

- More secure

- Can be revoked anytime

- Can have specific permissions

- Can set expiration

- Multiple tokens for different purposes

HOW TO CREATE PERSONAL ACCESS TOKEN

Steps:

1. **Go to GitHub**
   - Log into your GitHub account

2. **Go to Settings**
   - Click on your profile picture (top right)
   - Click "Settings"

3. **Developer Settings**
   - Scroll down in left sidebar
   - Click "Developer settings"

4. **Personal Access Tokens**
   - Click "Personal access tokens"
   - Click "Tokens (classic)"

5. **Generate New Token**
   - Click "Generate new token"
   - Click "Generate new token (classic)"

6. **Configure Token**
   - **Note:** Give a name (e.g., "test" or "devops-automation")
   - **Expiration:** Choose expiration period (30 days, 60 days, 90 days, or No expiration)

  - **Select scopes:** Check the permissions you want

Recommended scopes for this task:

- ✅ **repo** (Full control of private repositories)
  - repo:status
  - repo_deployment
  - public_repo
  - repo:invite
  - security_events
- ✅ **read:org** (Read org and team membership)
- ✅ **admin:repo_hook** (if needed)

7. **Generate Token**
   - Click "Generate token" button at bottom
   - **IMPORTANT:** Copy the token immediately
   - It will be shown only once!

Token format:

` ` `

ghp_xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

STEP 7: EXPORT THE TOKEN

Command:

bash

```
export token="ghp_xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
```

Replace with your actual token!

Explanation:

- **export token:** Creates environment variable named "token"

- **"your_token":** Your GitHub personal access token

Verify variables are set:

bash

```
echo $username
```

```
echo $token
```

STEP 8: GRANT EXECUTE PERMISSIONS TO SCRIPT

Command:

bash

```
chmod 777 list-users.sh
```

Or more secure:

bash

```
chmod +x list-users.sh
```

Explanation:

- **chmod:** Change file permissions
- **+x:** Add execute permission
- **777:** Full permissions (read, write, execute for everyone)

STEP 9: EXECUTE THE SCRIPT

Command format:

bash

```
sh list-users.sh <ORGANIZATION_NAME> <REPOSITORY_NAME>
```

OR

bash

```
./list-users.sh <ORGANIZATION_NAME> <REPOSITORY_NAME>
```

Example 1:

bash

```
sh list-users.sh devops-by-examples Python
```

Breakdown:

- **sh list-users.sh:** Execute the script
- **devops-by-examples:** Organization name
- **Python:** Repository name

Example 2:

bash

./list-users.sh kubernetes kubernetes

` ` `

Breakdown:

- **kubernetes:** Organization name (first argument)

- **kubernetes:** Repository name (second argument)

EXPECTED OUTPUT

If you have access to the repo:

` ` `

Listing users with read access to devops-by-examples/Python...

Users with read access to devops-by-examples/Python:

john-doe

jane-smith

contributor123

` ` `

If you don't have access:

` ` `

No users with read access found for devops-by-examples/Python.

` ` `

OR

` ` `

API rate limit exceeded

` ` `

OR

` ` `

Not Found

WHY NO OUTPUT IN SOME CASES?

Reason: You will not see any list of users because you don't have access to this repository.

You need to have access to the repos in order to see the output.

In real organization: If you are working in an organization, you will have access to the repo and you can see the output for the command:

bash

./list-users.sh your-organization-name your-repo-name

THE SCRIPT EXPLANATION

Full Script: list-users.sh

bash

#!/bin/bash

# GitHub API URL

API_URL="https://api.github.com"

```
# GitHub username and personal access token
USERNAME=$username
TOKEN=$token

# User and Repository information
REPO_OWNER=$1
REPO_NAME=$2

# Function to make a GET request to the GitHub API
function github_api_get {
  local endpoint="$1"
  local url="${API_URL}/${endpoint}"

  # Send a GET request to the GitHub API with authentication
  curl -s -u "${USERNAME}:${TOKEN}" "$url"
}

# Function to list users with read access to the repository
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) | .login')"

  # 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
}

# Main script
echo "Listing users with read access to ${REPO_OWNER}/${REPO_NAME}..."
list_users_with_read_access
```

DETAILED LINE-BY-LINE EXPLANATION

Section 1: Shebang and API URL

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

Explanation: Shebang - tells system to use bash to execute this script

```bash
# GitHub API URL
API_URL="https://api.github.com"
```

Explanation:

- **API_URL:** Variable storing GitHub API base URL
- **"https://api.github.com":** GitHub's REST API endpoint
- All API calls will be made to this base URL

GitHub API Documentation: According to GitHub Docs, the base URL for all API requests is: https://api.github.com

---

Section 2: Authentication Variables

```bash
# GitHub username and personal access token
```

```
USERNAME=$username

TOKEN=$token
```

Explanation:

- **USERNAME=$username:** Gets value from environment variable we exported earlier
- **TOKEN=$token:** Gets token from environment variable we exported earlier
- These are used for authentication with GitHub API

Why needed? GitHub API requires authentication to access repository information.

Section 3: Command Line Arguments

bash

```
# User and Repository information

REPO_OWNER=$1

REPO_NAME=$2
```

Explanation:

- **$1:** First command line argument (Organization/Owner name)
- **$2:** Second command line argument (Repository name)
- When you run: ./list-users.sh devops-by-examples Python
    - $1 = "devops-by-examples"
    - $2 = "Python"

Section 4: Function to Make API Request

bash

```
# Function to make a GET request to the GitHub API

function github_api_get {

  local endpoint="$1"

  local url="${API_URL}/${endpoint}"
```

```
    # Send a GET request to the GitHub API with authentication
    curl -s -u "${USERNAME}:${TOKEN}" "$url"
}
```

Line-by-line breakdown:

function github_api_get {

- Defines a reusable function named `github_api_get`

local endpoint="$1"

- **local:** Variable only exists within this function
- **endpoint:** The API endpoint we want to call
- **$1:** First argument passed to this function

local url="${API_URL}/${endpoint}"

- **url:** Complete URL for API request
- **${API_URL}:** Base URL (https://api.github.com)
- **/${endpoint}:** Specific endpoint path
- **Example:** https://api.github.com/repos/owner/repo/collaborators

curl -s -u "${USERNAME}:${TOKEN}" "$url"

- **curl:** Command line tool to make HTTP requests
- **-s:** Silent mode (no progress bar)
- **-u:** Authentication flag
- **"${USERNAME}:${TOKEN}":** Username and token for basic auth
- **"$url":** The URL to request

Comparison with GitHub API Documentation:

From GitHub Docs:

` ` `

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

` ` `

Authorization header required:

` ` `

Authorization: token YOUR_TOKEN

` ` `

Our script uses **basic authentication** format:

` ` `

-u username:token

Both methods work for GitHub API authentication.

Section 5: Function to List Users with Read Access

bash

```
# Function to list users with read access to the repository
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) | .login')"

  # 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
}
```

Line-by-line breakdown:

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

- **endpoint:** API endpoint to get collaborators
- **repos/${REPO_OWNER}/${REPO_NAME}/collaborators:** GitHub API path
- **Example:** repos/devops-by-examples/Python/collaborators

According to GitHub API Documentation:

` ` `

GET /repos/{owner}/{repo}/collaborators

Description: Lists collaborators for the specified repository.

Response format:

json

```
[
 {
  "login": "username",
  "permissions": {
   "admin": false,
   "maintain": false,
    "push": false,
    "triage": false,
    "pull": true
  }
 }
]
```

Understanding the curl + jq pipeline:

bash

```
collaborators="$(github_api_get "$endpoint" | jq -r '.[] | select(.permissions.pull == true)
| .login')"
```

Breaking it down:

1. github_api_get "$endpoint"

- Calls our function to make API request
- Returns JSON response with all collaborators

2. | (pipe)

- Sends output to next command

3. jq -r

- **jq:** JSON parser and filter
- **-r:** Raw output (no quotes)

4. '.[]'

- Iterate through each element in JSON array

5. select(.permissions.pull == true)

- **select:** Filter function
- **.permissions.pull:** Access the "pull" permission field
- **== true:** Only keep collaborators with pull (read) access

6. .login

- Extract only the username (login) field

Example JSON from API:

json

```
[
 {
   "login": "john-doe",
   "permissions": {
     "pull": true,
     "push": false,
     "admin": false
   }
 },
 {
   "login": "jane-smith",
   "permissions": {
     "pull": true,
     "push": true,
     "admin": false
   }
 }
]
```

After jq filtering:

```
john-doe
jane-smith
```

Understanding permissions in GitHub:

According to GitHub Documentation:

Permission levels:

- **pull (read):** Can read and clone the repository
- **push (write):** Can read, clone, and push to the repository
- **admin:** Full access including settings and collaborator management

Our script filters for:

bash

```
.permissions.pull == true
```

This means: **Anyone with at least read access** (which includes read, write, and admin users)

Display logic:

bash

```
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:

[[-z "$collaborators"]]

- **-z:** Test if string is empty
- If no collaborators found, string is empty

if empty:

- Print: "No users with read access found"

else:

- Print header
- Print list of usernames

Section 6: Main Script Execution

```bash
# Main script
echo "Listing users with read access to ${REPO_OWNER}/${REPO_NAME}..."
list_users_with_read_access
```

Explanation:

- **echo:** Print status message

- **list_users_with_read_access:** Call the function we defined

This is the entry point that starts the execution.

COMPLETE WORKFLOW DIAGRAM

```
1. User runs: ./list-users.sh devops-by-examples Python
           ↓
2. Script reads: REPO_OWNER=$1 (devops-by-examples)
          REPO_NAME=$2 (Python)
           ↓
3. Main script calls: list_users_with_read_access()
           ↓
4. Function creates endpoint: repos/devops-by-examples/Python/collaborators
           ↓
5. Calls: github_api_get(endpoint)
```

```
          ↓
6. Makes API request: curl -u username:token https://api.github.com/repos/devops-by-examples/Python/collaborators
          ↓
7. GitHub API returns: JSON with collaborators
          ↓
8. jq filters: Users with pull==true
          ↓
9. Output: List of usernames
```

GITHUB API ENDPOINTS REFERENCE

According to GitHub API Documentation:

1. List collaborators:
```
GET /repos/{owner}/{repo}/collaborators
```

2. Check if user is collaborator:
```
GET /repos/{owner}/{repo}/collaborators/{username}
```

3. Add collaborator:

```
PUT /repos/{owner}/{repo}/collaborators/{username}
```

4. Remove collaborator:

```
DELETE /repos/{owner}/{repo}/collaborators/{username}
```

Our script uses endpoint #1 to list all collaborators.

TESTING WITH DIFFERENT REPOSITORIES

Test 1: Public repository you have access to

bash

```
./list-users.sh your-org your-repo
```

Test 2: Large open-source project

bash

```
./list-users.sh kubernetes kubernetes
```

Test 3: Your personal repository

bash

```
./list-users.sh your-username your-repo-name
```