**API Overview: REST API vs. GraphQL**

APIs (Application Programming Interfaces) allow different software applications to communicate with each other. Two popular types of APIs are **REST API** and **GraphQL**. Let's dive into what they are, how they work, and use real-world examples to explain them.

**1. REST API (Representational State Transfer)**

**REST** is an architectural style for designing networked applications. It uses HTTP requests to perform CRUD operations (Create, Read, Update, Delete). REST APIs are based on a client-server model where the client requests data from the server, and the server responds with the requested data.

**Key Features of REST:**

* **HTTP Methods**: Uses standard HTTP methods like GET, POST, PUT, DELETE, etc.
* **Resources**: The data is represented as resources (e.g., users, products, etc.).
* **Stateless**: Each API request from the client contains all the information the server needs, so no session is stored on the server.
* **Cacheable**: Responses can be cached for efficiency.

**Real-world Example of REST API:**

Imagine you're using an online store website that shows product details. Here's how REST would work:

* **GET** /products: Fetches a list of all products.
* **GET** /products/123: Fetches details for the product with ID 123.
* **POST** /products: Creates a new product.
* **PUT** /products/123: Updates the details of product 123.
* **DELETE** /products/123: Deletes product 123.

Each URL (like /products or /products/123) represents a "resource," and the HTTP methods represent the actions you can perform on them.

**2. GraphQL API**

**GraphQL** is a query language for APIs developed by Facebook. It allows clients to request exactly the data they need, and nothing more. GraphQL APIs are more flexible and efficient compared to REST APIs.

**Key Features of GraphQL:**

* **Single Endpoint**: Unlike REST, which has multiple endpoints for different resources, GraphQL uses a single endpoint (e.g., /graphql).
* **Client-Specified Queries**: Clients can specify exactly what data they need in a query. This avoids over-fetching or under-fetching data.
* **Real-Time Updates**: With GraphQL, you can also subscribe to real-time updates on data changes (using subscriptions).

**Real-world Example of GraphQL:**

Imagine you're using a mobile app that shows information about a user, their posts, and comments on those posts. In a REST API, you might need multiple requests to get this data:

* **GET** /users/123: Fetches user details.
* **GET** /users/123/posts: Fetches posts made by user 123.
* **GET** /posts/456/comments: Fetches comments for post 456.

In GraphQL, you can send a single request to get all the information in one go:

graphql

CopyEdit

{

user(id: 123) {

name

email

posts {

title

comments {

text

}

}

}

}

This query requests:

* The name and email of the user with ID 123.
* The title of each of their posts.
* The text of each comment on those posts.

In a REST API, you would need separate requests for each of those pieces of data. With GraphQL, you can specify exactly what you need in one query.

**Key Differences Between REST and GraphQL:**

| **Feature** | **REST API** | **GraphQL API** |
| --- | --- | --- |
| **Data Fetching** | Fetches predefined data from fixed endpoints. | Fetches custom data based on queries. |
| **Multiple Endpoints** | Multiple endpoints for different resources. | Single endpoint for all operations. |
| **Over-fetching** | Can fetch unnecessary data (e.g., extra fields). | Avoids over-fetching by fetching only required data. |
| **Flexibility** | Less flexible, needs multiple requests. | More flexible, allows specifying exact data needed. |
| **Performance** | Can be inefficient (multiple requests for related data). | More efficient (single request for related data). |
| **Real-time Updates** | Not built-in, can be done via other protocols (e.g., WebSockets). | Can handle real-time updates with subscriptions. |

**When to Use REST API vs. GraphQL?**

* **Use REST** when:
  + You have fixed, predefined resources.
  + Your API clients don't need complex queries.
  + Simplicity and standardization are more important than flexibility.
* **Use GraphQL** when:
  + You need more flexible data retrieval.
  + Clients need to request specific data to avoid over-fetching.
  + You have complex relationships between data (e.g., a user, their posts, and comments).

**Conclusion**

* **REST** is a traditional, simple approach to APIs, great for straightforward use cases.
* **GraphQL** is a more modern, flexible alternative, suited for complex applications where precise control over the data is needed.

**Lab: API Testing with Postman for GitHub REST API**

This lab will guide you through the steps to perform API testing on GitHub's REST API using **Postman**. By the end of this lab, you'll be able to send requests to GitHub's API, validate responses, and automate API testing using Postman.

**Lab Overview**

GitHub provides a RESTful API that allows users to interact with their repositories, issues, pull requests, commits, and other resources available in GitHub. For this lab, we'll focus on the following:

1. **Fetching user information.**
2. **Listing repositories.**
3. **Creating an issue.**

You'll need a GitHub account and a personal access token to authenticate your requests.

**Prerequisites**

* **Postman Installed**: Ensure Postman is installed on your machine. If not, download it from here.
* **GitHub Account**: You need a GitHub account to generate an access token.
* **Personal Access Token**: GitHub's API requires authentication for most operations, so you will need a personal access token for authentication.

**Step 1: Generate GitHub Personal Access Token**

1. Log in to your **GitHub** account.
2. Go to **Settings** > **Developer settings** > **Personal access tokens**.
3. Click on **Generate new token**.
4. Choose the necessary scopes for the token. For this lab, you can use the following basic permissions:
   * repo (full control of private repositories)
   * user (access user information)
   * public\_repo (access public repositories)
5. Click **Generate token**.
6. **Copy the generated token** as you will use it for authentication in Postman.

**Step 2: Set Up Postman for API Testing**

**2.1 Create a New Request in Postman**

1. **Open Postman** and create a new request:
   * Click on **New** > **Request**.
   * Name the request (e.g., "GitHub User Information").
   * Add it to a collection or create a new collection for organizing your requests.
   * Create a blank collection named “GithubApiRequest” and add the requests inside it whatever u wl create

**2.2 Set Up Authentication**

1. For most requests to GitHub's REST API, you'll need authentication using your GitHub personal access token.
2. Edit the collection “GithubApiRequest”
3. Go to the **Authorization** tab in Postman.
4. Select **Bearer Token** as the type.
5. Paste the GitHub personal access token in the **Token** field.
6. and ur bearer token inside it.
7. Save it
8. and ur bearer token inside it.
9. Save it

**GITHUB API ENDPOINT PAGE :**

<https://docs.github.com/en/rest/repos/repos?apiVersion=2022-11-28#list-repositories-for-a-user>

**Step 3: Test GitHub API Endpoints**

**-- Create a get request under collection “GithubApirequest”**

**-- in the url >> add >>** [**https://api.github.com/users/USERNAME/repos**](https://api.github.com/users/USERNAME/repos)

**Replace username with ramannkhanna2**

* + - * **In authorization keep parent authorization**
  + **Send the api request and get the response**
  + **Test with curl as well :**

**curl -L \ -H "Accept: application/vnd.github+json" \ -H "Authorization: Bearer <YOUR-TOKEN>" \ -H "X-GitHub-Api-Version: 2022-11-28" \ https://api.github.com/users/USERNAME/repos**

**3.1 Get User Information**

GitHub's API allows you to fetch user information. Let's start with that:

* **Endpoint**: https://api.github.com/user
* **Method**: GET
* **Description**: Fetches the authenticated user's details.

1. In Postman, enter the following URL:

arduino

CopyEdit

https://api.github.com/user

1. Select the **GET** method from the dropdown next to the URL.
2. Click **Send**.
3. **Check the response**:
   * You should receive a **200 OK** response with user details (name, username, email, etc.).

**Validate the Response:**

* Status Code: 200 OK
* Check the body of the response for fields like login, id, name, etc.

**3.3 Create a Repository for the authenticated user :**

**https://docs.github.com/en/rest/repos/repos?apiVersion=2022-11-28#create-a-repository-for-the-authenticated-user**

Now, let's try creating GitHub repository using the API. We’ll need to use the **POST** method to create a new issue.

* **Endpoint**: https://api.github.com/user/repos
* **Method**: POST
* **Description**: Create a new issue in a specific repository.

1. Go to the **Body** tab in Postman and select **raw**. Choose **JSON** from the dropdown.
2. Add the following JSON body:

{

"name": "ramanrepofrompostman",

"description": "This is your first repo!",

"private": false

}

1. Click **Send**.
2. **Check the response**:

**Validate the Response:**

**3.3 Now delete the Repository for the authenticated user :**

**-- https://docs.github.com/en/rest/repos/repos?apiVersion=2022-11-28#delete-a-repository**

<https://api.github.com/repos/OWNER/REPO>

* + - * Replace owner and repo in above url like below :
    - <https://api.github.com/repos/ramannkhanna2/ramanrepofrompostman>

1. You must get a 204 response

**Integrating Tests, Deployment, and Notifications with GitHub Actions and Webhooks**

This unified process involves:

* **Running tests and reporting results** using the GitHub Status API.
* **Deploying the application** automatically with GitHub Actions.
* **Notifying stakeholders** using Slack Webhooks based on the deployment status.

Here’s how all of the above components come together in one cohesive GitHub Actions workflow:

name: CI/CD Pipeline with Deployment and Notifications

on:

push:

branches:

- main

pull\_request:

branches:

- main

jobs:

test:

runs-on: ubuntu-latest

steps:

- name: Checkout the repository

uses: actions/checkout@v2

- name: Set up Python

uses: actions/setup-python@v2

with:

python-version: '3.9'

- name: Install dependencies

run: |

pip install -r requirements.txt

- name: Run tests

run: |

pytest tests/

- name: Report Test Results to GitHub Status API

if: success()

run: |

curl -X POST \

-H "Authorization: token ${{ secrets.GITHUB\_TOKEN }}" \

-d '{"state": "success", "description": "Tests passed successfully."}' \

https://api.github.com/repos/${{ github.repository }}/statuses/${{ github.sha }}

- name: Report Test Failure to GitHub Status API

if: failure()

run: |

curl -X POST \

-H "Authorization: token ${{ secrets.GITHUB\_TOKEN }}" \

-d '{"state": "failure", "description": "Tests failed."}' \

https://api.github.com/repos/${{ github.repository }}/statuses/${{ github.sha }}

deploy:

runs-on: ubuntu-latest

needs: test

steps:

- name: Checkout the repository

uses: actions/checkout@v2

- name: Set up AWS CLI

uses: aws-actions/configure-aws-credentials@v1

with:

aws-access-key-id: ${{ secrets.AWS\_ACCESS\_KEY\_ID }}

aws-secret-access-key: ${{ secrets.AWS\_SECRET\_ACCESS\_KEY }}

aws-region: 'us-west-2'

- name: Deploy to AWS EC2

run: |

scp -i ${{ secrets.EC2\_SSH\_PRIVATE\_KEY }} ./myapp.zip ubuntu@${{ secrets.EC2\_PUBLIC\_IP }}:/home/ubuntu/

ssh -i ${{ secrets.EC2\_SSH\_PRIVATE\_KEY }} ubuntu@${{ secrets.EC2\_PUBLIC\_IP }} 'unzip /home/ubuntu/myapp.zip -d /var/www/myapp'

- name: Report Deployment Status to GitHub

run: |

curl -X POST \

-H "Authorization: token ${{ secrets.GITHUB\_TOKEN }}" \

-d '{"state": "success", "description": "Deployment to AWS successful."}' \

https://api.github.com/repos/${{ github.repository }}/statuses/${{ github.sha }}

- name: Notify on Slack (Deployment Success)

run: |

curl -X POST -H 'Content-type: application/json' \

--data '{"text": "Deployment Successful! The tests passed and the app has been deployed to AWS EC2 successfully."}' \

https://hooks.slack.com/services/T00000000/B00000000/XXXXXXXXXXXXXXXXXXXXXXXX

**Workflow Breakdown:**

1. **Test Job**:
   * Runs tests on each push or pull request.
   * Updates GitHub commit status based on test results.
2. **Deploy Job**:
   * Deploys the application only if the tests pass.
   * After deployment, it updates the commit status in GitHub to reflect whether the deployment was successful.
3. **Slack Notification**:
   * Sends a message to Slack with the outcome of the deployment (success or failure).

**Organizations and Teams in GitHub Enterprise Cloud**

In **GitHub Enterprise Cloud (GHEC)**, **organizations** and **teams** are key structures used to manage groups of users, control access to repositories, and apply security policies efficiently.

**1. Organizations in GHEC**

An **organization** in GitHub Enterprise Cloud is a shared account that allows multiple users to collaborate across multiple repositories. Organizations are designed for businesses, open-source projects, or large-scale teams requiring centralized management.

**Key Features of Organizations in GHEC**

* **Centralized Management**: Admins control permissions, security policies, and billing.
* **Security & Compliance**: Supports advanced security features such as **SAML SSO**, **SCIM provisioning**, and **audit logs**.
* **Teams & Roles**: Users are organized into teams with defined roles.
* **Repository Management**: Efficient management of repositories with granular access control.
* **Billing Consolidation**: All organization members share a single billing plan.

**Organization Roles**

* **Owner**: Has full administrative access, can manage settings, billing, security, and repositories.
* **Member**: Has limited permissions based on team roles and repository access.
* **Billing Manager**: Can manage billing without access to repositories or code.
* **Security Manager** (Optional Role): Can manage security alerts, secrets scanning, and vulnerability details without full admin access.

**2. Teams in GHEC**

Teams are groups of organization members that simplify user management and permission control. They are hierarchical and support nested structures to align with organizational workflows.

**Key Features of Teams**

* **Hierarchical Structure**: Supports parent-child relationships (nested teams).
* **Role-based Access Control (RBAC)**: Assign permissions at the team level to streamline repository access.
* **Mentions & Notifications**: Teams can be mentioned directly (@team-name) for faster collaboration.
* **Team Discussions**: Dedicated discussion spaces for better communication.
* **Automatic Access Control**: Grant team-wide permissions to multiple repositories.

**Team Roles**

* **Maintainer**: Full administrative rights over the team (add/remove members, manage discussions, etc.).
* **Member**: Has access based on team permissions but cannot manage the team structure.

**3. Managing Organizations in GHEC**

**a) Creating an Organization**

1. Navigate to [GitHub](https://github.com).
2. In the top-right corner, click your profile photo and select **"Your organizations"**.
3. Click **New Organization**.
4. Select your plan (Enterprise or Free/Team plan).
5. Provide organization details, such as:
   * Organization name
   * Billing email
6. Follow the instructions to configure security settings, SAML SSO (if required), and invite users.

**b) Managing Organization Members**

* Navigate to your organization’s page.
* Go to **People** > **Invite Member**.
* Assign roles (e.g., **Owner**, **Member**, or **Security Manager**) and add the user to specific teams.

**Bulk Invitation:** For large teams, GitHub supports bulk invitations via CSV import.

**4. Managing Teams in GHEC**

**a) Creating a Team**

1. Navigate to your organization's **Teams** tab.
2. Click **New Team**.
3. Enter:
   * **Team Name** (e.g., devops-team)
   * **Description**
   * **Parent Team** (if applicable)
4. Configure repository permissions:
   * **Read** (View access)
   * **Write** (Contribute code)
   * **Admin** (Full control over repository settings)
5. Click **Create Team**.

**b) Adding Members to a Team**

1. Navigate to the created team.
2. Click **Add Member**.
3. Enter the usernames of the members.
4. Assign roles like **Maintainer** or **Member**.

**c) Managing Team Permissions**

Permissions can be applied at two levels:

* **Directly on Repositories**: Assign granular permissions per repository.
* **Inherited via Parent Teams**: Child teams inherit permissions from their parent teams.

| **Permission Level** | **Description** |
| --- | --- |
| **Read** | View code and discussions. |
| **Triage** | Manage issues and pull requests. |
| **Write** | Push commits and manage branches. |
| **Maintain** | Manage repository settings. |
| **Admin** | Full administrative control. |

**5. Best Practices for Managing Organizations and Teams in GHEC**

1. **Use Nested Teams** for large organizations to simplify role management.
2. **Enforce SAML SSO** for secure authentication.
3. **Enable Audit Logs** to track security events and ensure accountability.
4. **Apply IP Allowlisting** to restrict access to trusted networks only.
5. **Define Repository Rulesets** for branch protection, required reviews, and security protocols.
6. **Utilize CODEOWNERS** files to assign reviewers automatically based on code areas.

**6. Common Use Cases in GHEC**

✅ **Product Teams** — Create a dedicated team for each product line with write access to relevant repositories.  
✅ **Security Teams** — Assign the **Security Manager** role to handle vulnerability alerts and code scanning results.  
✅ **Outsourcing & Consultants** — Use granular repository permissions or deploy keys for limited code access.  
✅ **Compliance Teams** — Enable audit log exports to track policy adherence and user activities.

**Lab : Invite a user to organization and make them a member**

* **Login with the user and check if ur able to see that user In the organization**
* **than go to organizational settings to define organizational role assignment to that user what permission u want to give it (read permissions)**
* **go and check the permission if they are applied or not**
* **than go and give write permission to that user and confirm if now its able to to those operations**

**Step 2: Configuring Organization Settings**

* Navigate to your organization’s page via:
  + **https://github.com/organizations/<your-org>/settings**
* Under **Security** settings:  
  ✅ Enable **Two-Factor Authentication (2FA)** requirement for added security.  
  ✅ Enable **IP Allowlisting** (Optional for demo) to restrict access to specific IP addresses.

**Step 3: Creating Teams in the Organization**

1. Navigate to your organization's page.
2. Click the **Teams** tab → Select **New Team**.
3. Create the following teams:

| **Team Name** | **Description** | **Parent Team (Optional)** |
| --- | --- | --- |
| dev-team | Developers working on core projects | — |
| qa-team | Quality assurance team | — |
| backend-team | Backend developers | dev-team |
| frontend-team | Frontend developers | dev-team |

1. Assign **Team Maintainers** to manage each team.
2. Add members to each team to demonstrate team-wide access control.

**Step 4: Creating Repositories for Teams**

1. Navigate to the **Repositories** tab in your organization.
2. Create the following repositories:

| **Repository Name** | **Description** | **Visibility** |
| --- | --- | --- |
| project-alpha | Main project repository | Public/Private |
| project-beta | Secondary project repository | Private |
| internal-docs | Internal documentation | Private |

1. Assign permissions to teams as follows:

| **Repository** | **Team** | **Permission Level** |
| --- | --- | --- |
| project-alpha | dev-team | **Write** |
| project-alpha | qa-team | **Read** |
| project-beta | backend-team | **Write** |
| project-beta | frontend-team | **Read** |
| internal-docs | qa-team | **Write** |

1. Navigate to each repository's **Settings → Manage Access** section to assign these permissions.

**Managing Dormant Users in GitHub Enterprise**

In **GitHub Enterprise Cloud**, user accounts that are inactive for a certain period are considered **dormant users**. Managing these accounts is essential to maintaining security, license optimization, and ensuring active participation in your organization.

**What Are Dormant Users?**

A **dormant user** is a member of your GitHub Enterprise Cloud organization who has not performed any activity within a specific timeframe. Dormant users:

✅ Retain their membership in the organization.  
✅ Continue to consume a license.  
✅ Pose potential security risks if not actively managed.

**Identifying Dormant Users**

GHEC offers an **Activity Log** and **API** methods to identify inactive users. A user is considered dormant if:

* They have not logged in.
* They have not pushed to repositories.
* They have not commented on issues, pull requests, or discussions.
* They have not performed any other verifiable action.

**Dormancy Period:** By default, GitHub considers a user dormant if they have been inactive for **90 days**.

**Steps to Manage Dormant Users**

Here’s how you can effectively manage dormant users in GHEC:

**Step 1: Identify Dormant Users via the Web UI**

1. Navigate to your **organization's settings**.
2. In the left sidebar, select **People**.
3. Click the **"Members"** tab.
4. Apply the filter **dormant** to list users who haven’t been active for 90 days or longer.

**Step 2: Identify Dormant Users via API (Recommended for Automation)**

For bulk operations or automated checks, use the following API endpoint:

**API Endpoint:**

bash

CopyEdit

GET /orgs/{org}/members?filter=dormant

**Example:**

bash

CopyEdit

curl -H "Authorization: token YOUR\_GITHUB\_TOKEN" \

-H "Accept: application/vnd.github+json" \

https://api.github.com/orgs/YOUR\_ORG/members?filter=dormant

This returns a list of dormant users in JSON format.

**Step 3: Audit Dormant Users**

Once identified, analyze these users for:

* **Role** in the organization (e.g., Admin, Member, Billing Manager).
* **Access Levels** to repositories, projects, or organizational data.
* **Potential Security Risk**, especially if they have privileged access.

**Step 4: Take Appropriate Actions**

Depending on your organization's policy, you can take the following actions:

✅ **Contact Users:** Send reminders to encourage activity.  
✅ **Remove from the Organization:** Ideal if the user no longer requires access.  
✅ **Demote to Outside Collaborator:** Useful for retaining limited access.  
✅ **Suspend the Account (For SAML SSO):** If your organization uses SAML SSO, suspension may be the best option.

**Step 5: Automate Dormant User Management (Recommended for Large Organizations)**

For enterprises with hundreds of users, automation reduces manual effort. GitHub Actions or scheduled API calls can automate:

* Periodic checks for dormant users.
* Sending email reminders.
* Automatically revoking inactive accounts.

**Example GitHub Action Workflow:**

yaml

CopyEdit

name: Dormant User Audit

on:

schedule:

- cron: "0 0 1 \* \*" # Runs monthly

jobs:

check-dormant-users:

runs-on: ubuntu-latest

steps:

- name: Get Dormant Users

run: |

curl -H "Authorization: token ${{ secrets.GITHUB\_TOKEN }}" \

-H "Accept: application/vnd.github+json" \

https://api.github.com/orgs/YOUR\_ORG/members?filter=dormant

**Step 6: Implement Security Best Practices**

To reduce the risks associated with dormant accounts:

✅ Enable **2FA** for all organization members.  
✅ Set **automatic removal policies** for dormant users.  
✅ Regularly audit membership with tools like **GitHub Audit Log**.  
✅ Use **GitHub Actions** to automate notifications and removal.

**Key Considerations for GHEC**

* GitHub **does not automatically remove dormant users**; manual action or automation is required.
* Dormant users still consume seats in your GitHub Enterprise Cloud **license count**.
* Users marked dormant may still have access to **private repositories**, unless explicitly removed.

**Best Practices for Managing Dormant Users**

✅ Establish a **Dormant User Policy** within your organization’s security guidelines.  
✅ Regularly schedule audits (e.g., quarterly or biannually).  
✅ Automate reminders to ensure critical accounts don’t become inactive unintentionally.  
✅ Use **GitHub Insights** for broader reporting on user activity.