**Topics**

1. **Introduction to Continuous Integration (CI) with GitHub Actions**
   * CI with GitHub Actions is a powerful and simple way to automate your workflows for code testing, building, and deploying directly from GitHub.
   * **GitHub Actions** allows you to create workflows defined in .yml files within the .github/workflows/ directory.
   * These workflows are triggered by GitHub events such as push, pull request, and more.
2. **Setting up a CI Pipeline with GitHub Actions**
   * Understanding the basic components of GitHub Actions:
     + **Workflow**: A YAML file defining the CI/CD pipeline.
     + **Job**: A set of steps that run on a specific runner (e.g., Ubuntu, Windows).
     + **Step**: An individual task within a job, such as installing dependencies or running tests.
     + **Event**: Triggers that start workflows, like push, pull\_request, etc.
3. **Automating CI Tasks with Python**
   * Using Python to trigger GitHub Actions workflows, pass variables, and monitor job status.
4. **Triggering CI Jobs from Python**
   * Using the GitHub API to trigger workflows programmatically from Python.

**Practical Exercises**

**1. Setup a Basic GitHub Actions CI Pipeline**

* **Create a Simple CI Pipeline** in GitHub Actions:
  1. In your repository, create a .github/workflows/ci.yml file with the following content:

yaml

Copy code

name: CI Pipeline

on:

push:

branches:

- main

pull\_request:

branches:

- main

jobs:

build:

runs-on: ubuntu-latest

steps:

- name: Checkout code

uses: actions/checkout@v2

- name: Set up Python

uses: actions/setup-python@v2

with:

python-version: '3.8'

- name: Install dependencies

run: |

pip install -r requirements.txt

- name: Run tests

run: |

pytest tests/

- name: Deploy application

run: |

./deploy.sh

* 1. **Explanation**: This workflow runs whenever there is a push or pull request to the main branch. It checks out the code, sets up Python, installs dependencies, runs tests with pytest, and then runs the deployment script.

**2. Triggering GitHub Actions Workflow with Python**

* **Using GitHub API to Trigger CI Workflow**: You can use the GitHub API to trigger GitHub Actions workflows manually or programmatically.

To use the GitHub API:

* 1. **Create a Personal Access Token** with the repo and workflow permissions from GitHub.
  2. Install requests if you don’t have it already:

bash

Copy code

pip install requests

* 1. **Python Script to Trigger Workflow**: Here’s how to trigger the GitHub Actions workflow manually from Python:

python

Copy code

import requests

# GitHub Repository details

repo\_owner = "your\_username"

repo\_name = "your\_repository"

workflow\_file = "ci.yml" # The name of your GitHub Actions workflow file

# Personal Access Token (use a secure method to handle this)

token = "your\_github\_personal\_access\_token"

# API endpoint for triggering a workflow

api\_url = f"https://api.github.com/repos/{repo\_owner}/{repo\_name}/actions/workflows/{workflow\_file}/dispatches"

# Data to pass to the API, specifying the branch (e.g., main)

data = {

"ref": "main" # You can specify another branch here

}

# Headers with the Authorization token

headers = {

"Authorization": f"Bearer {token}",

"Accept": "application/vnd.github.v3+json"

}

# Make the API request to trigger the workflow

response = requests.post(api\_url, json=data, headers=headers)

if response.status\_code == 201:

print("Workflow triggered successfully!")

else:

print(f"Failed to trigger workflow: {response.text}")

* 1. **Explanation**: This script triggers a GitHub Actions workflow by sending a POST request to the GitHub API. It uses the dispatches endpoint to manually trigger a workflow on a specific branch (main in this case).

**3. Automating Python Tests in CI Pipeline**

* **Set up a test script (test\_sample.py)** to run in the CI pipeline using pytest:

python

Copy code

def test\_addition():

assert 1 + 1 == 2

def test\_subtraction():

assert 2 - 1 == 1

* **Run tests automatically in the GitHub Actions workflow**: In the ci.yml file above, the step to run tests is defined as:

yaml

Copy code

- name: Run tests

run: |

pytest tests/

* The pytest command will automatically find and run any tests in the tests/ directory, and you’ll see the results in the GitHub Actions logs.

**4. Automating Deployment with Python**

* **Create a simple deployment script (deploy.sh)**:

bash

Copy code

#!/bin/bash

echo "Deploying application..."

# Add your deployment commands here (e.g., copying files, restarting servers)

* **Run the deployment script in the GitHub Actions workflow**: The deployment step is defined in the ci.yml file:

yaml

Copy code

- name: Deploy application

run: |

./deploy.sh

* **Explanation**: This step will run your deploy.sh script, which could include commands for deploying to servers, cloud services, etc. You could also use Python in the deployment script to handle more complex tasks like interacting with APIs or remote servers.

**5. Triggering GitHub Actions from a Local Python Script**

You can trigger a GitHub Actions workflow from your local environment, which is useful if you want to automate deployments or tests directly from your local machine.

**Example Python Script** to trigger the workflow from a local machine:

python

Copy code

import requests

# Define the repository and the GitHub API URL

repo\_owner = "your\_username"

repo\_name = "your\_repository"

workflow\_id = "ci.yml" # Name of the GitHub Actions workflow

token = "your\_personal\_access\_token" # Your GitHub token

# Trigger API endpoint for GitHub Actions

api\_url = f"https://api.github.com/repos/{repo\_owner}/{repo\_name}/actions/workflows/{workflow\_id}/dispatches"

headers = {

"Authorization": f"Bearer {token}",

"Accept": "application/vnd.github.v3+json"

}

data = {

"ref": "main" # Specify the branch (e.g., main or any other branch)

}

# Send the request to trigger the workflow

response = requests.post(api\_url, json=data, headers=headers)

if response.status\_code == 201:

print("GitHub Actions workflow triggered successfully.")

else:

print(f"Failed to trigger workflow: {response.text}")

**6. Challenge: Automate Full CI/CD Pipeline with Python**

* **Task**: Create a full CI/CD pipeline that:
  + **Triggers GitHub Actions** whenever code is pushed to the main branch.
  + **Runs tests automatically** using pytest.
  + **Builds and deploys** the application after successful tests.
* You can use the above GitHub Actions workflow to automate the process, and the Python scripts to programmatically trigger these steps as needed.

**Summary**

* **Day 9** focused on integrating **Python** with **GitHub Actions** to automate continuous integration tasks such as running tests, building, and deploying applications.
* You learned how to trigger GitHub Actions workflows using the GitHub API, automate tests with pytest, and deploy applications using Python and shell scripts.
* By automating CI tasks, you streamline the development process and ensure faster, more reliable software delivery.

**Topics**

1. **Introduction to Continuous Integration (CI)**
   * CI is the practice of automatically integrating code changes into a shared repository several times a day to catch errors early and improve code quality.
   * CI pipelines automate the build, test, and deployment process.
2. **Setting up a CI Pipeline using Python**
   * Understanding the components of a CI pipeline:
     + **Version control system (VCS)**: GitHub, GitLab, Bitbucket.
     + **Build system**: Jenkins, GitLab CI, CircleCI.
     + **Testing**: Unit tests, integration tests.
     + **Deployment**: Deploying to staging/production.
3. **Using Python for Automating CI Tasks**
   * Automating code pushes, testing, building, and deployments.
   * Using libraries like pytest for testing, subprocess for interacting with the CI server, and requests for triggering CI jobs via APIs.
4. **Triggering CI Jobs via APIs**
   * Integrating Python with popular CI tools like Jenkins, GitLab CI, and CircleCI via their REST APIs.

**Practical Exercises**

**1. Setup a Simple CI Pipeline in GitLab**

* **GitLab CI Configuration**: Create a .gitlab-ci.yml file to define your CI pipeline.

yaml

Copy code

stages:

- build

- test

- deploy

build:

stage: build

script:

- echo "Building the application"

- python3 setup.py install

test:

stage: test

script:

- echo "Running tests"

- pytest tests/

deploy:

stage: deploy

script:

- echo "Deploying application"

- ./deploy.sh

* This YAML file will instruct GitLab CI to run the **build**, **test**, and **deploy** stages.
* **Python to Trigger GitLab CI Job**: To trigger a pipeline from Python, use the GitLab API. You will need an **API token** to authenticate requests.

Install the requests library:

bash

Copy code

pip install requests

Use Python to trigger a GitLab CI job:

python

Copy code

import requests

gitlab\_url = "https://gitlab.com/api/v4/projects/{project\_id}/trigger/pipeline"

api\_token = "your\_gitlab\_api\_token"

ref = "main" # Branch name to trigger the pipeline

headers = {

"PRIVATE-TOKEN": api\_token

}

data = {

"ref": ref

}

response = requests.post(gitlab\_url, headers=headers, data=data)

if response.status\_code == 201:

print("Pipeline triggered successfully.")

else:

print(f"Failed to trigger pipeline: {response.text}")

* **Explanation**: The script sends an HTTP POST request to GitLab’s API, passing the project ID and triggering a pipeline on the main branch.

**2. Jenkins CI Pipeline with Python**

* **Jenkins Setup**: You can configure Jenkins to trigger CI jobs using the Jenkins REST API.
* **Create a Jenkins Job**: Create a new job in Jenkins to build and test your application.
* **Python to Trigger Jenkins Job**: You can use the Jenkins API to trigger jobs using Python.

Example Python script to trigger a Jenkins job:

python

Copy code

import requests

jenkins\_url = "http://your\_jenkins\_url/job/your\_job\_name/build"

user = "your\_username"

api\_token = "your\_jenkins\_api\_token"

response = requests.post(jenkins\_url, auth=(user, api\_token))

if response.status\_code == 201:

print("Job triggered successfully.")

else:

print(f"Failed to trigger job: {response.text}")

* **Explanation**: This script triggers a Jenkins job using a POST request. Jenkins requires authentication, which can be done using your Jenkins username and API token.

**3. Automate Testing with Python (Using pytest)**

* **Setup a Test Script with pytest**: Create a test file test\_sample.py:

python

Copy code

def test\_addition():

assert 1 + 1 == 2

def test\_subtraction():

assert 2 - 1 == 1

* **Run the Tests via Python**: Use Python to run your tests and capture results:

python

Copy code

import subprocess

result = subprocess.run(['pytest', 'test\_sample.py'], capture\_output=True, text=True)

if result.returncode == 0:

print("Tests passed successfully.")

else:

print(f"Tests failed: {result.stderr}")

* **Explanation**: This script runs your tests using pytest and captures the output to print the test results.

**4. Automate Build and Deployment with Python**

* **Automating Build Process**: Use Python to run a build script when code is pushed to the repository. For example, use Python to trigger a build in Jenkins or GitLab CI automatically.

**Build script**:

python

Copy code

import subprocess

def run\_build():

print("Running build process...")

result = subprocess.run(['python3', 'setup.py', 'install'], capture\_output=True, text=True)

if result.returncode == 0:

print("Build succeeded.")

else:

print(f"Build failed: {result.stderr}")

run\_build()

* **Automating Deployment**: Once the build process is complete, use Python to deploy the application. You could invoke a shell script or a deployment tool such as Ansible to handle this.

**Deployment script**:

python

Copy code

import subprocess

def deploy():

print("Deploying application...")

result = subprocess.run(['./deploy.sh'], capture\_output=True, text=True)

if result.returncode == 0:

print("Deployment succeeded.")

else:

print(f"Deployment failed: {result.stderr}")

deploy()

**5. Continuous Deployment with Python (Using Webhooks)**

* **Using Webhooks for Continuous Deployment**: Configure webhooks to trigger Python scripts or external services like Jenkins when new commits are pushed to your version control system (e.g., GitHub, GitLab).
* **Python Webhook Listener**: Listen for webhook events (such as a GitHub push) and trigger your CI pipeline.

python

Copy code

from flask import Flask, request

app = Flask(\_\_name\_\_)

@app.route('/webhook', methods=['POST'])

def webhook():

data = request.json

if data.get('ref') == 'refs/heads/main':

print("Received webhook for push to main branch.")

# Trigger Jenkins or GitLab job

trigger\_ci\_job()

return '', 200

def trigger\_ci\_job():

# Trigger a job on Jenkins/GitLab

print("Triggering CI job...")

# Add your CI trigger code here

if \_\_name\_\_ == '\_\_main\_\_':

app.run(port=5000)

* **Explanation**: This Python script uses Flask to listen for webhook events. When a commit is pushed to the main branch, it triggers a CI job.

**6. Challenge: Create a Full CI Pipeline with Python Integration**

* **Task**: Create a simple CI pipeline that:
  + Pulls code from GitHub.
  + Runs tests using pytest.
  + Builds the application using a build script.
  + Deploys the application using a deployment script.

You can use any CI tool like **GitHub Actions**, **Jenkins**, or **GitLab CI** and automate the steps via Python.

**Summary**

* **Day 9** focused on integrating **Python** with **Continuous Integration (CI)** systems like **GitLab**, **Jenkins**, and **GitHub Actions**.
* You learned how to automate triggering of CI jobs, execute tests, build applications, and deploy them using Python scripts.
* **CI Pipeline automation** is essential in DevOps for efficient software delivery and integration.

Would you like to move forward to **Day 10: Continuous Delivery with Python** or focus more on the CI pipeline?