CI-CD Pipeline

Real world applications of GitHub Actions

GitHub Actions can be incredibly useful for automating various tasks in software development workflows. For TodoAPI application, here are some real-world applications:

- **Continuous Integration (CI):** Automatically run tests whenever changes are pushed to the repository. This ensures that new code doesn't break existing functionality.
- **Code Quality Checks:** Automatically check code formatting, run static code analysis, and enforce coding standards using tools like SonarQube or Checkstyle.
- **Automated Deployment:** Automatically deploy your application to staging or production environments after successful builds, ensuring that the latest changes are always available.
- **Issue and Pull Request Management:** Automatically assign labels, notify team members, or perform other actions based on the creation or modification of issues or pull requests.
- **Scheduled Tasks:** Schedule periodic tasks such as database backups, data synchronization, or generating reports.
- **Dependency Updates:** Automatically monitor for updates to project dependencies and create pull requests to update them.

Building a complete CI pipeline for a sample Java-based application

Below is a basic example of a CI pipeline for a Java-based application using GitHub Actions

```
name: CI CD Pipeline for TodoApp

on:
  push:
    branches:
      - master

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
    - name: Checkout repository
      uses: actions/checkout@v2
```

```
    - name: Set up JDK
      uses: actions/setup-java@v2
      with:
        java-version: '11'
        distribution: 'adopt'
        cache: maven

    - name: Build with Maven
      run: mvn clean install

    - name: Run tests
      run: mvn test

    - name: Upload jar to folder
      uses: actions/upload-artifact@v4
      with:
        name: my-artifact
        path: target/
```

This GitHub Actions workflow is designed to run continuous integration (CI) tasks for a Java project. Here's a breakdown of each part of the workflow:

- This line sets the name for your GitHub Actions workflow. It helps identify the purpose of the workflow when viewing it in the GitHub Actions dashboard.

```
name: CI CD Pipeline for TodoApp
```

- This section specifies the events that trigger the workflow. In this case, the workflow will be triggered whenever a push event occurs on the master branch.

```
on:
  push:
    branches:
      - master
```

- Here, you define a job named build. This job runs on an Ubuntu environment (ubuntu-latest).

```
jobs:
  build:
    runs-on: ubuntu-latest
```

- This step checks out the source code of the repository into the runner environment.

```
    steps:
    - name: Checkout repository
```

```
      uses: actions/checkout@v2
```

- This step sets up Java Development Kit (JDK) version 11 in the runner environment using the actions/setup-java action. It also configures Maven caching for dependencies.

```
    - name: Set up JDK
      uses: actions/setup-java@v2
      with:
        java-version: '11'
        distribution: 'adopt'
        cache: maven
```

- This step builds the project using Maven. It first cleans the project (mvn clean) and then builds it (mvn install).

```
    - name: Build with Maven
      run: mvn clean install
```

- This step runs the tests for the project using Maven.

```
    - name: Run tests
      run: mvn test
```

- This step uploads the built artifact (jar file) to the GitHub Actions workspace. The name parameter specifies the name of the artifact (in this case, my-artifact), and the path parameter specifies the directory containing the artifact (target/).

```
    - name: Upload jar to folder
      uses: actions/upload-artifact@v4
      with:
        name: my-artifact
        path: target/
```

Add a new self-hosted runner to GitHub Actions

Enable WSL Integration with Ubuntu on Docker Desktop:

- Open Docker Desktop.
- Go to Settings.
- Navigate to Resources -> WSL Integration.
- Enable integration with additional distros and select Ubuntu.

Download and Extract the GitHub Actions Runner:

- Open Windows Subsystem for Linux 2 (WSL2)
- Open a terminal or WSL shell.
- Go to the Settings tab of your repository on GitHub.
- Navigate to the "Actions" section.
- Select "Add runner" to initiate the process of adding a self-hosted runner.
- Choose the appropriate options:
    - Operating System: Select Linux.
    - Architecture: Select x64 (assuming your system architecture is 64-bit).
- After selecting Linux, follow the steps to set up the self-hosted runner.
- After configuration, the self-hosted runner should be operational and ready to execute workflows for your repository. You can use it in your workflow YAML files by specifying `runs-on: self-hosted`.

CD pipeline for a Dockerized application using self-managed runner

Here's an example CD pipeline for deploying a Dockerized application to Docker Hub using a self-managed runner:

```
deploy:
    runs-on: self-hosted
    needs: build

    steps:
    - name: Checkout repository
      uses: actions/checkout@v2

    - name: Download web-app content
      uses: actions/download-artifact@v4
      with:
        name: my-artifact
        path: target/

    - name: Build Docker image
      run: docker build -t my-user/todoapp .

    - name: Log in to Docker Hub
      run: echo "${{ secrets.DOCKER_PASSWORD }}" | docker login -u "${{
secrets.DOCKER_USERNAME }}" --password-stdin

    - name: Push image to Docker Hub
      run: docker push my-user/todoapp
```

- This section defines a deployment job named deploy. It runs on a self-hosted environment and depends on the completion of the build job before execution.

```
deploy:
  runs-on: self-hosted
  needs: build
```

- This step checks out the source code of the repository into the runner environment.

```
    steps:
    - name: Checkout repository
      uses: actions/checkout@v2
```

- This step downloads the artifact named my-artifact from the GitHub Actions workspace. The artifact is typically a file or directory generated during a previous step of the workflow. In this case, it seems to be the output of a build process, possibly containing the built web application files.

```
    - name: Download web-app content
      uses: actions/download-artifact@v4
      with:
        name: my-artifact
        path: target/
```

- This step builds a Docker image named todoapi using the Dockerfile in the current directory.

```
    - name: Build Docker image
      run: docker build -t my-user/todoapp .
```

- This step logs in to Docker Hub using Docker Hub credentials provided as secrets.

```
    - name: Log in to Docker Hub
      run: echo "${{ secrets.DOCKER_PASSWORD }}" | docker login -u "${{
secrets.DOCKER_USERNAME }}" --password-stdin
```

- This step pushes the Docker image tagged as todoapi to the Docker Hub repository. Make sure to replace my-user/todoapp with your actual Docker Hub username and repository name.

```
    - name: Push image to Docker Hub
      run: docker push my-user/todoapp
```

> NOTE: Adding Environment Variables

> To add environment variables to this workflow, you can use GitHub Secrets, which allow you to securely store sensitive information.
>
> - Navigate to your GitHub repository.
> - Go to the "Settings" tab.
> - In the left sidebar, click on "Secrets".
> - Click on the "New repository secret" button.
> - Enter the name and value of your environment variable.
> - Click on "Add secret" to save it.
>
> In the provided workflow, environment variables are used for Docker Hub authentication. The `DOCKER_USERNAME` and `DOCKER_PASSWORD` secrets are used to log in to Docker Hub.