Github Actions

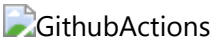


Table of Contents

What is Github Actions

GitHub Actions is a powerful automation tool provided by GitHub, a web-based hosting service for version control using Git. It allows you to define custom workflows and automate various tasks within your software development lifecycle. With GitHub Actions, you can build, test, and deploy your code directly from your GitHub repositories.

Overview of Github Actions

GitHub Actions is a continuous integration and continuous delivery (CI/CD) platform that allows you to automate your build, test, and deployment pipeline. You can create workflows that build and test every pull request to your repository, or deploy merged pull requests to production.

GitHub Actions goes beyond just DevOps and lets you run workflows when other events happen in your repository.

For example, you can run a workflow to automatically add the appropriate labels whenever someone creates a new issue in your repository.

GitHub provides Linux, Windows, and macOS virtual machines to run your workflows, or you can host your own self-hosted runners in your own data center or cloud infrastructure.

Components of Github Actions

GitHub Actions consists of several key components that work together to enable automation and workflow execution. These components include:

- **Workflows**: Workflows are the core component of GitHub Actions. They are defined using YAML files and represent a set of steps and actions that execute in response to specific events or triggers. Workflows define the automation logic for your software development processes.
- **Events and Triggers**: Workflows are triggered by specific events that occur within a GitHub repository. Events can include actions such as pushing to a branch, creating or closing a pull request, or scheduling a specific time interval. Triggers determine when a workflow should start.
- **Jobs**: Workflows can contain one or more jobs. Jobs represent a set of steps that execute on the same runner, which is the execution environment for the workflow. Jobs can run concurrently or sequentially, depending on the configuration.
- **Steps**: Steps are individual units of work within a job. Each step performs a specific action or task, such as running a shell command, invoking an action, or interacting with external services. Steps define the order in which actions are executed.
- **Actions**: Actions are standalone units of work that can be reused across different workflows. They are defined in YAML files or Docker containers and encapsulate a specific task or set of operations. Actions can be created by the community or provided by third-party vendors, and they can be used directly within workflows.
- **Runners**: Runners are the compute resources on which jobs and steps in workflows are executed. GitHub provides hosted runners that run workflows on virtual machines in the GitHub Actions infrastructure. You can also set up your own self-hosted runners on your own infrastructure.
- **Environment Variables**: GitHub Actions allows you to define and use environment variables within your workflows. These variables can store information such as access tokens, API keys, or custom configuration values. They can be set at the workflow, job, or step level.

- **Secrets**: Secrets are encrypted variables that can be used to store sensitive information, such as credentials or private keys. Secrets are securely stored and can be accessed by workflows during execution. They are typically used to authenticate with external services or protect sensitive data.

 components

How GitHub Actions work

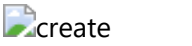
- Github Actions is fully integrated into the Github.
- As described in the above concepts, it mainly consists of workflows that are stored in the git repository (.github/workflows/).
- Workflows are typically triggered by events such as PR, Issues, Commits in the repository or it can also be triggered from the external events using repository webhooks.
- When the workflow is triggered, the runner picks up the job and executes one by one. The job consists of steps (with an action) that perform a unit of work in the workflow.

Create a Workflow in the Github Repository

- To get started using GitHub Actions, we need to add a folder to the root of the GitHub repository.

.github/workflows , Manually or you can follow the steps to create using action button.

- We can create a workflow using Actions button in the Github.(If you dont have an Github account create one by clicking on the following link. https://github.com/)
- Create an new repository in the account by clicking the **NEW** button in the left-side menu bar

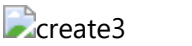
 create

- Give the Repository name and keep the repo public and click on create repository.

 create1

- Click on the **Actions** in the repository.As a next page appears click on **set up a workflow yourself**

 create2

 create3

- You can give the name of the workflow what ever you want but the file should be in the format of .yml/yaml.

 create4

Example Workflow

- 1.Lets run a simple hello-world workflow and see the results.

Paste the above yaml syntax in the demo.yml

```
name: hello-world
on: push
jobs:
  my-job:
    runs-on: ubuntu-latest
    steps:
      - name: my-step
        run: echo "Hello World!"
```

- **name**: hello-world: This line sets the name of the workflow as "hello-world". It is used to identify the workflow in the GitHub Actions dashboard.
- **on**: push: This line specifies that the workflow will be triggered when a push event occurs in the repository. In other words, whenever code changes are pushed to any branch of the repository, this workflow will be executed.
- **jobs**: This keyword starts the definition of the jobs section, which contains one or more jobs that will be executed as part of the workflow.
- **my-job**: This line defines the name of the job as "my-job". A job is a unit of work that can consist of multiple steps.
- **runs-on**: ubuntu-latest: This line specifies that the job will run on a GitHub-hosted runner with the latest version of the Ubuntu operating system. The runner provides the execution environment for the job.
- **steps**:: This keyword starts the definition of the steps section within the job. Steps define the individual tasks or actions to be performed.
- **-name**: my-step: This line defines the name of the step as "my-step". It helps identify the step in the workflow execution log.
- **run**: echo "Hello World!": This line specifies the command to be executed in the step. In this case, the echo command is used to print the message "Hello World!" to the console.

So, when a push event occurs in the repository, this workflow will be triggered. It will run a single job named "my-job" on an Ubuntu runner. Within the job, there is a single step named "my-step" that executes the command echo "Hello World!". As a result, the workflow will print the "Hello World!" message in the workflow execution log

 work

- 2.Once u have pasted the code click on the commit changes and give the Description.

 work

- This Process has started Github Action workflow

Check Results

- One can check the results by clicking the Action tab.

 result

- In this page you can see the workflow and runs of the workflow.
- You can check the logs and result of workflow by clicking the workflow run from right side bar(Create demo.yml) .

 result

 result

CI-CD Pipeline using Git Actions

Continuous Integration (CI):

Continuous Integration is a software development practice where developers frequently integrate their code changes into a shared repository. After each integration, an automated build and test process is triggered to verify that the changes haven't introduced any bugs or conflicts. The goal of CI is to detect and fix integration issues early in the development cycle.

Continuous Deployment (CD):

Continuous Deployment is an extension of Continuous Integration where code changes that pass automated tests are automatically and continuously deployed to production or a live environment without manual intervention. The aim of CD is to streamline the release process and ensure that new features and bug fixes are delivered to end-users rapidly and reliably.

CI/CD Pipeline:

A CI/CD Pipeline is an automated sequence of steps that code changes go through, from version control to production deployment. It typically includes building the code, running tests, packaging, and deploying to various environments. The pipeline ensures consistency and efficiency in the software development and deployment process.

Build CI-CD Pipeline Using GitHub Actions

Let's create a CI/CD Pipeline using GitHub Actions and using that workflow how we can automate process of build and push docker image to docker hub.

- Here we build an Project in Springboot and Push the code to Github.
- Using Github Action tab we will create a workflow for CI/CD.
- Using CI/CD actions we Perform Build and Test the code , Create Docker image of my Application and Push the Docker image to DockerHub.

Step 1: Create a new Springboot Starter project with maven and having an java version 8.Add Spring web dependencies.

 java

 java

Step 2: Go to main class and write an endpoint like shown in a image.

 java

Pushing Code to Github Repository

- You can create an new repository and push the code to the Github.
- Go to the terminal (make sure you are in the correct directory) in STS an execute the commands.
- Intialize the github

```
git init
```

 push

- Check the status

```
git status
```

- Add src and pom.xml file

```
git add src

git add pom.xml
```

 push

 push

- Commit the code with some message to branch master and add origin using the repository link. execute the command one by one

```
git commit -m "code"
```

```
git branch -M master
```

```
git remote add origin "Repositorylink"
```

 push

- Finally Push the code to the master

```
git push -u origin master
```

 push

- You can check the code has been pushed to master

 push

Create Workflow

- Now go to Action button and create a workflow,and paste the code in the .yaml file and commit changes.

```
# This workflow will build a Java project with Maven, and cache/restore any
dependencies to improve the workflow execution time
# For more information see: https://help.github.com/actions/language-and-
framework-guides/building-and-testing-java-with-maven

name: project cicd flow

on:
  push:
    branches: [ master ]
  pull_request:
    branches: [ master ]
```

```
jobs:
  build:

    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v2
    - name: Set up JDK 1.8
      uses: actions/setup-java@v1
      with:
        java-version: '1.8'
        distribution: 'adopt'
        cache: maven
    - name: Build with Maven
      run: mvn clean install

    - name: Build & push Docker image
      uses: mr-smithers-excellent/docker-build-push@v5
      with:
        image: sharan7898/github-action
        tags: latest
        registry: docker.io
        dockerfile: Dockerfile
        username: ${{ secrets.DOCKER_USERNAME }}
        password: ${{ secrets.DOCKER_PASSWORD }}
```

Explainantion

This GitHub Actions workflow is a Continuous Integration/Continuous Deployment (CI/CD) pipeline for a Java project that includes building and pushing a Docker image to Docker Hub. Here's a step-by-step explanation of the workflow:

```
name: project cicd flow
```

- The name of the GitHub Actions workflow is "project cicd flow."

```
on:
  push:
    branches: [ master ]
  pull_request:
    branches: [ master ]
```

- The workflow is triggered on both pushes and pull requests to the master branch. This means that whenever code changes are pushed or a pull request is made to the master branch, the workflow will run.

```
jobs:
  build:
```

- The workflow defines one job named build. This job is responsible for the entire CI/CD process.

```
runs-on: ubuntu-latest
```

The job will run on a virtual machine with the latest version of Ubuntu.

```
steps:
- uses: actions/checkout@v2
- name: Set up JDK 1.8
  uses: actions/setup-java@v1
  with:
    java-version: '1.8'
    distribution: 'adopt'
    cache: maven
- name: Build with Maven
  run: mvn clean install
- name: Build & push Docker image
  uses: mr-smithers-excellent/docker-build-push@v5
  with:
    image: sharan7898/github-action
    tags: latest
    registry: docker.io
    dockerfile: Dockerfile
    username: ${{ secrets.DOCKER_USERNAME }}
    password: ${{ secrets.DOCKER_PASSWORD }}
```

- The steps section contains the individual steps that will be executed as part of the job.

- The first step uses the actions/checkout action to clone the repository and check out the latest code from the default branch (in this case, master).
- The second step sets up JDK 1.8 using the actions/setup-java action. It configures the Java version to 1.8 and uses the AdoptOpenJDK distribution. Additionally, it caches Maven dependencies to speed up subsequent builds.
- The third step runs the mvn clean install command, which uses Maven to build the Java project. The clean goal removes any previous build artifacts, and the install goal compiles the code, runs tests, and packages the application.
- The last step uses the mr-smithers-excellent/docker-build-push action to build the Docker image and push it to Docker Hub. It specifies the Docker image name (sharan7898/github-action) and the latest tag. The image is pushed to Docker Hub (docker.io) using the provided Dockerfile. The Docker Hub username and password are read from GitHub Secrets, allowing secure authentication during the push process.

Image in the Docker

- Pull the workflow in STS using

```
git pull origin master
```

The **.github/workflows/maven.yaml** folder will be created.

- Open the maven.yml and edit the file for your Docker hub credentials.
- Login to Dockerhub if u dont have an account you can create using this link https://www.docker.com/
- Create a repository in the Docker Hub.
- Change the image name and repository name in the in the maven.yml file(workflow file).
- Create a new file in the Project, here i have given the name Dockerfile for the file.
- Paste the above code inside the Dockerfile

```
FROM openjdk:8
EXPOSE 8080
ADD target/github-action.jar github-action.jar
ENTRYPOINT ["java","-jar","/github-action.jar"]
```

 docker

The Dockerfile describes the steps to build a Docker image for a Java application using OpenJDK 8.

FROM openjdk:8: This line sets the base image for the Docker image. In this case, the image is based on OpenJDK 8, which means the Java application will be run using Java 8.

EXPOSE 8080: This line exposes port 8080 in the Docker image. It allows the containerized Java application to accept incoming connections on port 8080.

ADD target/github-action.jar github-action.jar: This line copies the github-action.jar file from the target directory of your project (assuming it's a Maven project) to the root of the Docker image. The ADD command is used to copy files from the host machine (in this case, your project directory) to the Docker image.

ENTRYPOINT ["java","-jar","/github-action.jar"]: This line sets the entry point for the Docker container. It specifies the command that will be executed when the container starts. In this case, it tells Docker to run the Java application using the java -jar command, with the github-action.jar file as the argument.

Note: Specify the jar name in pom.xml (jar name must be same to image name because we are expecting as an image name)

 docker

Adding Secrets

- Go to Github, click on the settings and click on secrets and variables and go to Actions

 secrets

- Click on the New repository secret

 secrets

- Add Name , Secret of our Docker credentials and then click Add secret.

 secrets

- Again create an another New repository to add an Docker password.

Running Image in Docker

- Once u have added the secret goto STS and commit and push the recent changes (add Docker file and update the pom.xml file)
- Once u have pushed the code goto Actions and you can see the Actions running.
- The Action name with the commit message build is executed all the steps in the docker file.

 build

- Now goto DockerHub repository and refresh it and you can see that the Image has been created.

 build

Running Image Locally

- Run the Docker in your local Machine.

- Go to DockerHub and click on the Public view and copy the Docker Pull Command

 local

 local

- Go to Command Prompt and paste the Docker Pull Command

 local

- Go to Docker desktop and run the container, Once the Application is completed running go to localhost:8080/welcome.

 local

Note: Refer the following video for more information https://www.youtube.com/watch?v=NppkHKvnrqc&t=3s

GitHub Action Environments

- GitHub Actions environments are a feature that allows you to define custom execution environments for your workflows.
- These environments provide a way to specify a set of environment variables, secrets, and other configurations that can be associated with specific branches, tags, or manually triggered workflows.
- When a GitHub Actions workflow deploys to an environment, the environment is displayed on the main page of the repository. For more information about viewing deployments to environments, see "Viewing deployment history."
- When a workflow job references an environment, the job won't start until all of the environment's protection rules pass.
- A job also cannot access secrets that are defined in an environment until all the environment protection rules pass.

Create a Github Action Environments

- Go your repository and click Settings.
- In the left sidebar, click Environments.
- Click New environment.

 envi

- Enter a name for the environment, then click Configure environment.

 envi

- Optionally, specify people or teams that must approve workflow jobs that use this environment. For more information, see "Required reviewers."

1.Select Required reviewers.

2.Enter up to 6 people or teams.

3.Only one of the required reviewers needs to approve the job for it to proceed.

4.Click Save protection rules.

- Optionally, specify the amount of time to wait before allowing workflow jobs that use this environment to proceed. For more information, see "Wait timer."

1.Select Wait timer.

2.Enter the number of minutes to wait.

3.Click Save protection rules.

- Optionally, disallow bypassing configured protection rules. For more information, see "Allow administrators to bypass configured protection rules."

1.Deselect Allow administrators to bypass configured protection rules.

2.Click Save protection rules.

- Optionally, specify what branches can deploy to this environment. For more information, see "Deployment branches."

1.Select the desired option in the Deployment branches dropdown.

2.If you chose Selected branches, enter the branch name patterns that you want to allow.

- Optionally, add environment secrets. These secrets are only available to workflow jobs that use the environment. Additionally, workflow jobs that use this environment can only access these secrets after any configured rules (for example, required reviewers) pass. For more information, see "Environment secrets."

1.Under Environment secrets, click Add Secret.

2.Enter the secret name.

3.Enter the secret value.

4.Click Add secret.

- Optionally, add environment variables. These variables are only available to workflow jobs that use the environment, and are only accessible using the vars context. For more information, see "Environment variables."

1.Under Environment variables, click Add Variable.

2.Enter the variable name.

3.Enter the variable value.

4.Click Add variable.

Using an Environment

Each job in a workflow can reference a single environment. Any protection rules configured for the environment must pass before a job referencing the environment is sent to a runner. The job can access the environment's secrets only after the job is sent to a runner.

When a workflow references an environment, the environment will appear in the repository's deployments. For more information about viewing current and previous deployments, see "Viewing deployment history."

You can specify an environment for each job in your workflow. To do so, add a jobs.<job_id>.environment key followed by the name of the environment.

For example, this workflow will use an environment called production.

```
name: Deployment

on:
  push:
    branches:
      - main

jobs:
  deployment:
    runs-on: ubuntu-latest
    environment: production
    steps:
      - name: deploy
        # ...deployment-specific steps
```

When the above workflow runs, the deployment job will be subject to any rules configured for the production environment. For example, if the environment requires reviewers, the job will pause until one of the reviewers approves the job.

```
name: Deployment

on:
  push:
    branches:
      - main

jobs:
  deployment:
    runs-on: ubuntu-latest
    environment:
      name: production
      url: https://github.com
    steps:
```

```
      - name: deploy
        # ...deployment-specific steps
```

Deployment Protection Rules

Deployment protection rules, also known as deployment approval rules, are a feature in GitHub Actions that allows you to control the deployment process of your applications. These rules add an extra layer of control to your deployment workflows, ensuring that certain conditions are met before a deployment is allowed to proceed.

With deployment protection rules, you can set up specific conditions that must be met before an automated deployment can proceed. These conditions may include requiring manual approvals, checks, or specific conditions related to the state of the code, environment, or other factors.

For example, you might want to enforce the following conditions before deploying a new version of your application:

- **Manual Approval**: Require manual approval from specific users or teams before the deployment proceeds. This ensures that someone reviews the changes and confirms that it's safe to deploy.
- **Status Checks**: Ensure that certain status checks, such as passing tests or code analysis, are successful before deploying. This ensures that the code is in a stable state.
- **Environment Checks**: Verify that the target deployment environment meets specific conditions or requirements before proceeding with the deployment.
- **Branch Protection Rules**: Enforce branch protection rules, ensuring that only specific branches or tags can be deployed.

To set up deployment protection rules in GitHub Actions, you can use the if conditionals in your workflow YAML file, or you can use the jobs.<job_id>.steps[step_id].if expression to control the execution of specific steps based on conditions.

Here's a simple example of using an if condition to require a manual approval for deployment:

```
name: Deploy to Production

on:
  push:
    branches:
      - main

jobs:
  deploy:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout code
```

```
        uses: actions/checkout@v2

      # Other build and test steps go here

      - name: Deploy to Production
        run: |
          # Deploy the application to production
           if: github.event_name == 'push' && github.ref == 'refs/heads/main' &&
github.actor == 'approved_user'
```

In this example, the deployment step will only be executed if the push event is targeting the main branch and the actor (user) who triggered the event is the approved_user. This would require the approved user to manually trigger the workflow or have specific permissions to deploy to production.

By using deployment protection rules, you can add safety measures to your CI/CD process, ensuring that deployments meet certain criteria before going live, and reducing the risk of accidental or undesired deployments.

For more about Action Environments refer the following video : https://www.youtube.com/watch?v=5XfgT9A9PHw&ab_channel=MickeyGousset

Action Runners

A GitHub Actions runner is a piece of software responsible for executing workflows in GitHub Actions.

It allows you to run your workflows on your own hardware or infrastructure, providing more control over the environment in which your workflows are executed.

GitHub Actions runners can be either self-hosted or hosted by GitHub. Let's take a closer look at each type:

1.**Self-hosted runner:**

- A self-hosted runner is a runner that you set up and manage on your own infrastructure, such as your local machine, on-premises servers, or cloud virtual machines.
- With self-hosted runners, you have more control over the environment, including the operating system, software, and dependencies installed on the runner.
- Self-hosted runners are useful when you need to perform builds or tests in an environment that is not available on GitHub-hosted runners or when you require access to specific hardware resources or secure networks.

2.**GitHub-hosted runner:**

- GitHub-hosted runners are provided and maintained by GitHub.
- GitHub offers different types of hosted runners that come pre-installed with various software and tools, such as Ubuntu, macOS, and Windows runners with different versions and configurations.

- You can use GitHub-hosted runners for most common use cases since they provide a wide range of operating systems and software environments.
- However, they have some limitations, such as the inability to access resources behind a firewall or specific software configurations not available on the hosted runners.

You can configure your workflows to run on specific runners, either self-hosted or GitHub-hosted, using the runs-on keyword in your workflow YAML file.

Understanding Runners

```
name: My Workflow

on:
  push:
    branches:
      - main

jobs:
  my_job:
    runs-on: ubuntu-latest

    steps:
      # Your workflow steps go here
```

- In the example above, the workflow my_job is set to run on a GitHub-hosted runner with the latest version of Ubuntu.
- If you wanted to use a self-hosted runner, you would change the runs-on value to the label of your self-hosted runner, like runs-on: self-hosted.
- Using GitHub Actions runners, you can automate your software development workflows, run tests, deploy applications, and more in a controlled and customizable environment.

Self-Hosted Runners

- In GitHub Actions, a self-hosted runner is a type of runner that you can set up and manage on your own infrastructure.
- Unlike GitHub-hosted runners, which are provided and maintained by GitHub, self-hosted runners run on your own hardware, virtual machines, or cloud instances.
- Self-hosted runners allow you to have more control over the environment in which your workflows run.
- You can customize the runner's operating system, installed software, dependencies, and configurations to match your specific project requirements.

- This makes self-hosted runners particularly useful for projects with specific toolchain or environment needs, or those that require access to resources behind a firewall or within a private network.

One can see the reference video on how to create a self-hosted runners and connect that with Github Actions : https://www.youtube.com/watch?v=Rb2pUKdmdYo&t=16s&ab_channel=Abhishek.Veeramalla

- Setting up a self-hosted runner involves installing the GitHub Actions Runner application on your infrastructure and registering it with your GitHub repository.
- Once the runner is set up and connected to your repository, GitHub will use it to execute workflows when they are triggered.

References

- https://www.youtube.com/watch?v=Rb2pUKdmdYo&t=16s&ab_channel=Abhishek.Veeramalla
- https://www.youtube.com/watch?v=5XfgT9A9PHw&ab_channel=MickeyGousset
- https://www.youtube.com/watch?v=NppkHKvnrqc&t=3s
- https://help.github.com/actions/language-and-framework-guides/building-and-testing-java-with-maven