GitHub Actions: Basics

Raju Ahmed Shetu Follow
Jun 1, 2020 · 4 min read ★

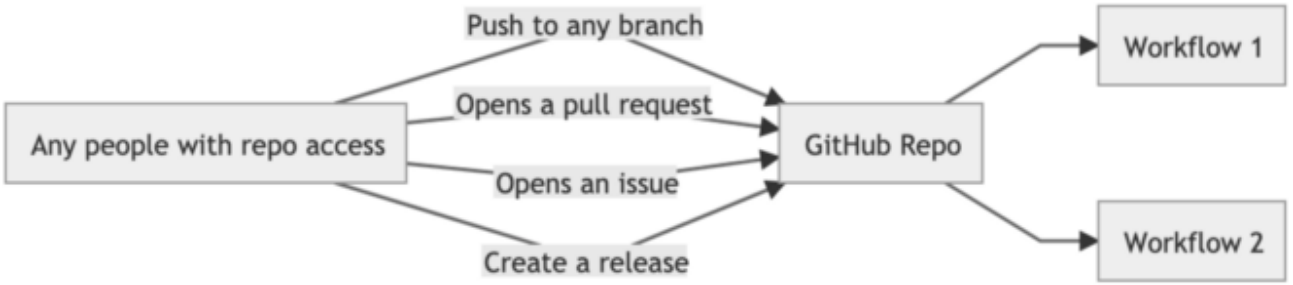
In modern days developing softwares without a CI/CD pipeline is unthinkable. And if it can be tied up with your version control system then that is the power you can harness for good. It would save you a lot of times as those CI/CD platforms are managed and also literally have no setting up steps.

I have already used bit of Jenkins. At work, we are using GitLab CI heavily right now. So I really wanted to try out GitHub Actions now. In the following discussion I won't be comparing any of the platforms rather I will go through the concepts of GitHub Actions. I will try to explain with keywords and building blocks of GitHub Actions with use case example and also try to run a simple workflow at the end of this blog.

From this point to the next I will simply write a git repository as **repo** for less typing. Sorry, if find it inconvenient.

What happens in GitHub Actions?

Let's get graphical.

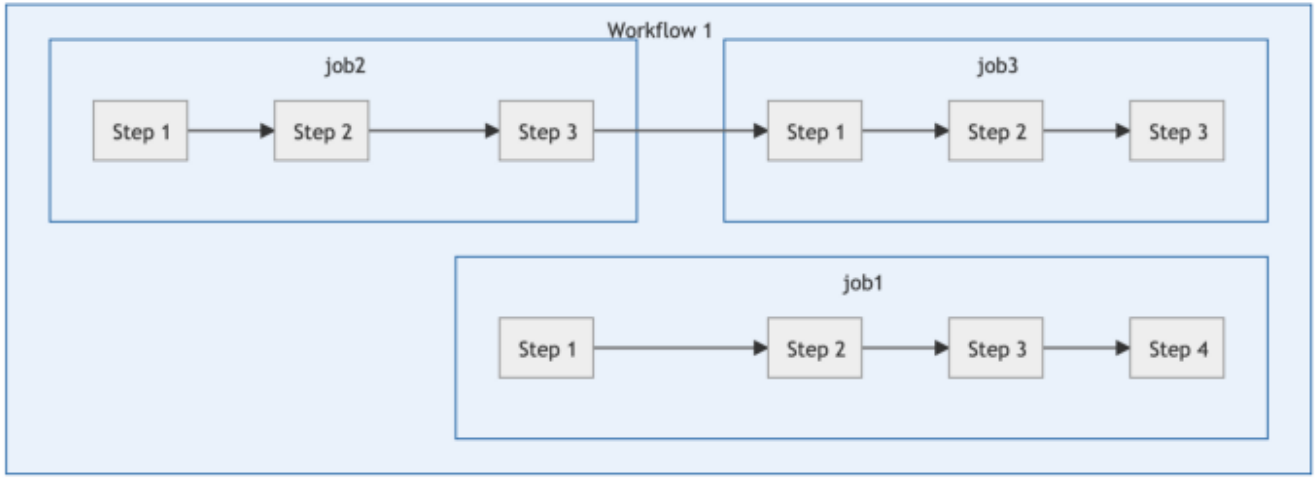


GitHub Workflow Trigger Diagram

- Let's say you have a GitHub repo. Any user with access to it can push, open a pull request or an issue, create a release and also many other things.
- Most of actions that is initiated by a user can be used as trigger to start a workflow.

What is workflow?

Simply a workflow is a collection of job definition that will be executed concurrently as well as sequentially. A job is consists of several steps which will be executed sequentially. Let's get graphical again



Typical Workflow Diagram

- Let's say we a workflow named `Workflow 1` in our repo. We want to run it when any push event happens.
- `Workflow 1` has three jobs `job1` , `job2` , `job3` Each job can have arbitrary number of steps.

- So when a user pushes any code to our repo, `job1` and `job2` will be fired concurrently. `job3` will only trigger when `job2` will be passed.
- If any step inside a job fails, the job fails which also make the workflow fails.

Where does this workflow file live in repo?

Your repo must have a folder named `.github/workflows` This folder should contain all the workflow file you can write. Workflow files are written as `yaml` files.

Building Block of a Workflow file

What I want at this point is to get you familiar important keywords of workflow file. I will try to explain each keywords with sample use cases in separate `.yaml` file. These `.yaml` files are not complete workflow file rather it's a snap of each keywords. I have added comments to necessary lines in the snippet to make it more clear. **Please read this files with comments. I believe this will help you immensely understanding the workflow files**. At the end we will combine most of them and make a simple workflow to validate.

name

```yaml
# Each workflow must have a name
# Here the workflow will be named as Sample Workflow
name: Sample workflow
```

action-name.yaml hosted with ❤ by **GitHub** **view raw**

on

```yaml
# Each workflow is triggered on some events
# will be triggered on push
on: push

# will be triggered on push and create pull_request
on: [push, pull_request]

# You can also have some configuration like this
on:
  push: # will be triggered for any push
  pull_request:
    types: [opened] # will be triggered when a pull request is created
  issues:
    types: [opened] # will be triggered when a issue is created
  issue_comment:
    types: [deleted] # will be triggered when a comment in a issue is deleted

# You can trigger any workflow using schedule too
on:
  schedule: 0 12 * * * # this will be triggered on 12pm (UTC) each day
```

```yaml


# You can filter out branches and tags too
on:
  push:
    tags: # will be triggered when v1.0 or v2.0 is pushed.
      - v1.0
      - v2.0
    branches: # will be triggered when code is pushed to master or any branch starting w
      - master
      - release/*
  pull_request:
    branches: # will be triggered when a pull request is submitted to develop or master
     - develop
     - master


# you can also ignore branches and tags to by simply using branches-ignore instead of br
# same goes for tags-ignore in place of tags
# you can not use branches-ignore and branches together and same of tags


on:
  push:
    branches-ignore:
      - feature/* # will not trigger for any push to feature/ branches
    tags-ignore:
      - v1.* # will not trigger any version like v1.1, v1.5, v1.8




# you can find more in here https://help.github.com/en/actions/reference/events-that-tri

```

action-on.yaml hosted with ❤ by **GitHub** **view raw**

env

```yaml
# You can have environment variables in the workflow
# this will be available in each job and steps inside it.

env:
  PROJECT_ID: sample-project-id
  USERNAME: sample-username

# So now in each steps you can access them by $PROJECT_ID or $USERNAME
# Normally the environment variables will show their value when printed in logs

# You might want to hide sensitive information in logs like password
# For this use,  secrets.
# You can set secrets under Variables in each repository settings

# Repo --> Settings --> Variables
env:
  PROJECT_ID: sample-project-id
  USERNAME: sample-username
  PASSWORD: ${{secrets.PASSWORD}} # you have to set PASSWORD in variables under repo set

# You can have env variables in jobs and steps too.
```

action-env.yaml hosted with ❤ by **GitHub** **view raw**

jobs

```yaml
# there might be multiple jobs definition under jobs.
# this is the job annotation. It can contain multiple job
# in each job definition runs-on and steps are required.
jobs:
  # name of the job to be shown in GitHub
  first-job:
    # os in which the job will run
    # Possible values are ubuntu-latest, windows-latest, ubuntu-16.04
    # there might be many version.
    runs-on: ubuntu-latest # required
    # job specific env variables
    # this env variables won't be available to second-job
    env:
      ENV_VAR: hello-world
      SECOND_VAR: hello-world
    # I will explain the steps separately. Generally steps are list of objects.
    steps: # required
      -
  second-job:
    runs-on: windows-latest
    steps:
      -



# One more interesting thing you can do is to run a job in different os
# For this, you need to strategy under each job config
jobs:
  first-job:
    strategy:
```

```
      matrix:
        os: [ubuntu-latest, macos-latest, windows-latest]
    # This will allow the job to run in 3 different os
    runs-on: ${{ matrix.os }}
    steps:
      -


# Apart from this you can also run all the steps in a docker container

jobs:
  first-job:
    runs-on: ubuntu-latest
    container:
      # this will run all the steps in this job inside a python:3 docker container
      image: "python:3"
    steps:
     -


# By default all the jobs that are defined under jobs keywords
# Sometime we need to start a job after a job is finished.
# for this we need the needs keyword

jobs:
  # as first-job and fourth-job don't have any needs keywords
  # they will start together
  first-job:
  second-job:
    # only run when first-job is successful
    needs: [first-job]
  third-job:
    # only run when first-job and second-job is successful
    needs: [first-job, second-job]
```

```
  fourth-job:


# also sometimes you need docker container to be started as sidecar for a job to run
jobs:
  first-job:
    services:
      postgres:
        image: postgres # docker image name
        ports:
          - "5432:5432" #port exposed
      redis:
        image: redis
```

action-job.yaml hosted with ❤ by **GitHub** **view raw**

steps

```
# Under each job there is a keyword called steps
# steps contains list of step
# In here will simply explain single step block
# as steps are list, if one fails, the next steps are skipped. There are ways to trigger


# name of the step
name: name of the step
id: sample_id # needed if this step has output to be used in other steps
run: echo 'Hello World' # command we want to run


# why GitHub Actions are so powerful
```

```
# why GitHub Actions are so powerful
# because it has a action marketplace that will let you
# use other's actions and publish yours
name: checkout code at current push
uses: actions/checkout@v1

# here actions is the username, checkout is the repo name
# v1 is the tag
# you can use branch name, sha or tag after @
# but always safe to use tag as it is static


# sometimes some action might take input from outside to act on
# you can pass this using with keywords
name: Running simple steps
uses: actions/hello-world-javascript-action@master
with:
  who-to-greet: 'Mona the Octocat'


## Other than using public actions you can also use your own actions that lives in your
name: Local Action
uses: ./.github/actions/hello
# for this to succeed make sure you have a file named
# ./.github/actions/hello/action.yaml in your repo
# I will go more into it in my subsequent blogs


# You might want to run a docker container in your steps
# simply use this
name: Running docker container
uses: docker://python:3
# this will bootup a python:3 docker container
```

```yaml

# You can have env variables scoped to our steps only
name: Env Variable Steps
run: echo $HELLO_WORLD
env:
  HELLO_WORLD: hello-world # will not be available to other steps


# sometimes you might want to run a step always
name: Run only in case of Failure
run: echo the workflow failed
if: ${{ failed() }} # possible values are failed(), always(), success()



# also you can run a steps based on event type and branches too
name: Run only when push to master
run: echo the code is pushed to master branch
if: ${{ github.event_name == 'push' && github.ref == 'master' }}



# a job might also have output that we can use in subsequent steps too
# let's assume we have a step with id second_step in any step before this step and it ha
name: Print Previous Step Output
run: echo ${{steps.second_step.outputs.time}}

```

action-step.yaml hosted with ❤ by **GitHub** **view raw**

I guess reading this amount of code is overwhelming.

Let's implement a simple workflow that does the following.

- It will be triggered on `push` in `master` branch
- It will have three jobs `job1` , `job2` and `job3` Here `job1` and `job3` will start concurrently while `job2` will only if `job1` is successful.
- I will just simple print some of the line in each job just copying from the above. Nothing meaningful.

I just wanted to show how you can put all of this together.

```
name: Sample Workflow
on:
  push: # enabling push event on master branch to fire the workflow
    branches:
      - master
jobs:
  job1:
    runs-on: ubuntu-latest
    steps:
      - name: checkout out current code using public action
        uses: actions/checkout@v2
      - name: get the list of files in our current directory and its subdirectory
        run: | # this | enables us to multiple commands, good for formatting
          pwd
          ls -alR
  job2:
    runs-on: ubuntu-latest
```

```
18      needs: [job1]
19      steps:
20        - name: print current files # this steps won't be showing any files as we didn't c
21          run: |
22            pwd
23            ls -alR
24
25    job3:
26      runs-on: ubuntu-latest
27      steps:
28        - name: print env variables
29          run: echo $VAR1 $VAR2 $VAR3
30          env:
31            VAR1: Thank you
32            VAR2: very much.
33            VAR3: Enjoy !!!
```

sample-workflow.yaml hosted with ❤ by **GitHub** **view raw**

I pushed it in our master branch and the workflow ran successfully in here. You can find in here. I have also pushed the codes in here.

Why I liked GitHub Actions?

Writing a CI/CD pipeline from scratch is daunting. But in the most of the platform you have to do it yourselves skimming through lots of documents and links. GitHub Actions enable you to use other's actions in your workflow which just make your workflow file small and speed up your

workflow building steps. Also you can write your own actions and publish it to actions marketplace in GitHub that can be used by others.

Thank you for bearing with me. This is just a blog to get you started. In my next blog I will go step by step on how to package and deploy django application in cloud using GitHub actions. Stay tuned !!!

Ci Cd Pipeline Github Actions Automation Workflow Github