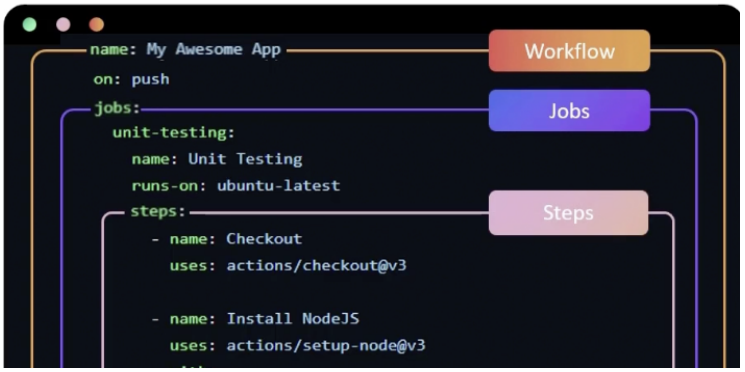
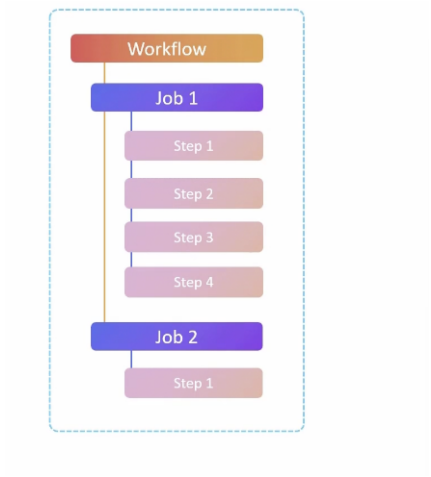
Core Components

We have 3 core components, which are Workflow, Jobs and Steps, OJShiwi.



A workflow is an automated process capable of executing one or more jobs. It is widely used in software development process from building and testing code to deploy to various environments. These workflows are defined using YAML files and are located within your repository. A repository can have multiple workflows, each of which runs in response to a specific event occurring in your repository. To uniquely identify workflows, each workflow can also have an optional name keyword. This name will be visible in the actions tab of the GitHub repository. In this example, the workflow is triggered for a push event,

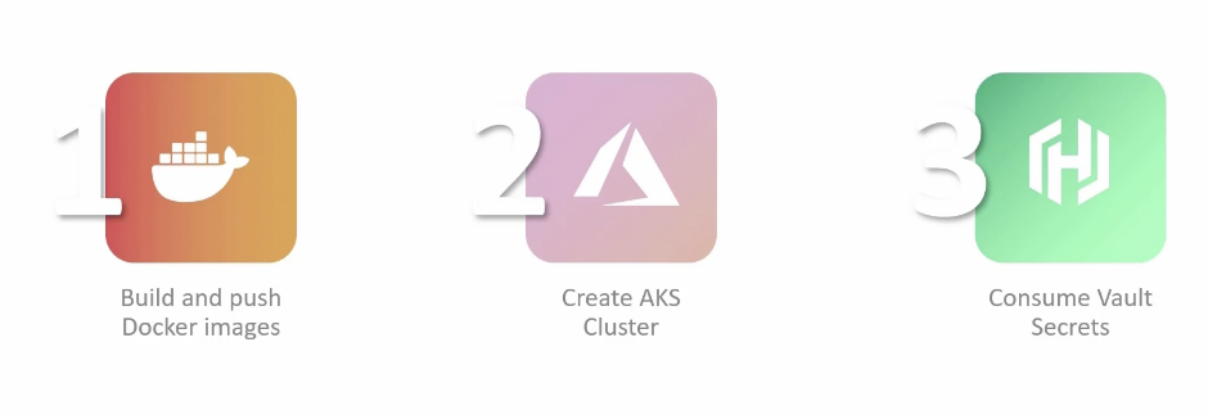
The next core component is a job. So jobs are the building blocks of a workflow, and you can have one or more jobs within a single workflow. Each job is, again, associated with a runner. It could be a GitHub hosted or a self-hosted runner. So we use the runs on attribute to specify, you know, uh, the runner configurations, that we only have one job and it'll, you, it'll run on the up into latest, you know, run.



The next component is steps. So steps in GitHub actions, or in GitHub workflows are individual tasks or actions that make up a job. So these are executed sequentially within a jobs runner environment. Steps can include commands, actions, or scripts allowing you to automate specific actions like building, testing, or deploying code as part of your CICD process.
So in this example, we have four steps. The first step uses an action named checkout of TAG V three

The second step also uses a action to install node is on the runner, which provides node and NPM commands in the run apart. And the third step uses a run keyword to execute shell commands on the runner. In this case, it is using NPM to install dependencies. And the last step, again, uses a run keyword to execute the NPM test command. And all the steps are executed sequentially.

So, actions within GitHub workflows are prebuilt reusable automation components designed for a specific task. These actions can be created by you or by members of the community, making it easy to share and reuse automation logic across the repositories. Actions also facilitate integration with third party tools and services, therefore enhancing your projects automation capabilities. So a few examples of official GitHub actions include, you know, the build and push docker images. So this is a GitHub action, which is used for, you know,



Create and Run First Workflow

https://github.dev/NishantGautam023/trying-github-actions

Install github-actions extension,

```yaml
name: My first workflow

on: push

jobs:
  first_job:
    runs-on: ubuntu-latest
    steps:
      - name: welcome message
        run: echo "My first Github Actions Job"

      - name: List files
        run: ls

      - name: Read file
        run: cat README.md
```

This YAML file is a configuration for a GitHub Actions workflow. GitHub Actions is a feature on GitHub that allows you to automate tasks within your software development workflow.

name: My first workflow: This is the name of the workflow. It's a label you use to identify this specific workflow among others in your repository.

on: push: This specifies the event that will trigger the workflow. In this case, the workflow is triggered on a 'push' event, meaning anytime you or someone else pushes commits to any branch in your repository, this workflow will run.

jobs: Jobs are a set of steps that execute on the same runner. Within a workflow, you can define multiple jobs. Jobs run in parallel by default, but you can also configure them to run sequentially.

first_job: This is the identifier for the first (and in this case, the only) job in this workflow. You can name it anything you want.

runs-on: ubuntu-latest: This specifies the type of runner that the job will execute on. A runner is a server with a specific operating system. Here, it's using the latest version of Ubuntu Linux.

steps: Steps are individual tasks that run commands within a job. A job can have one or more steps.

- name: welcome message: This is the first step. Each step can have a name, which makes it easier to identify what the step is doing.

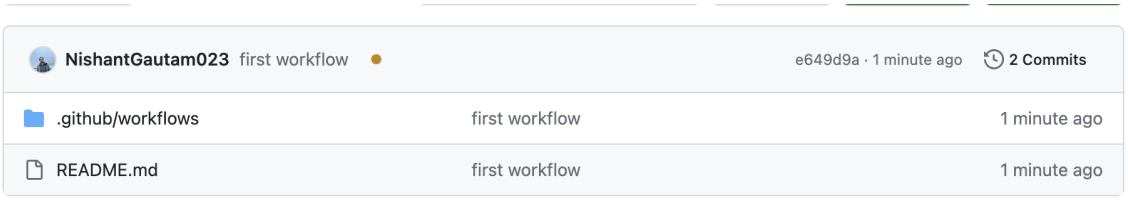
run: echo "My first Github Actions Job": This is the command that the step will execute. In this case, it's printing "My first Github Actions Job" to the console.

- name: List files: This is the second step.

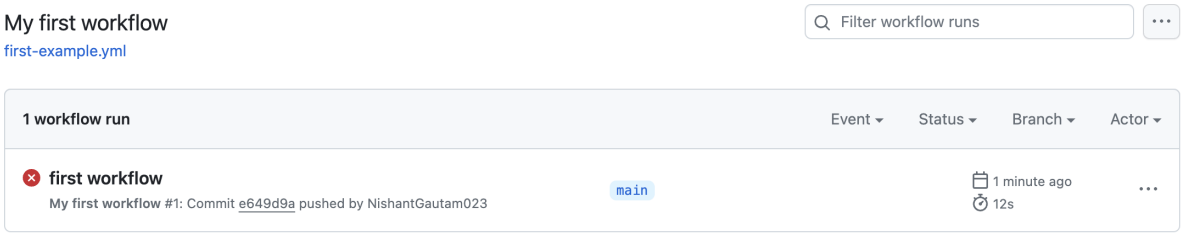
run: ls: This command lists all the files in the current directory where the workflow is running.

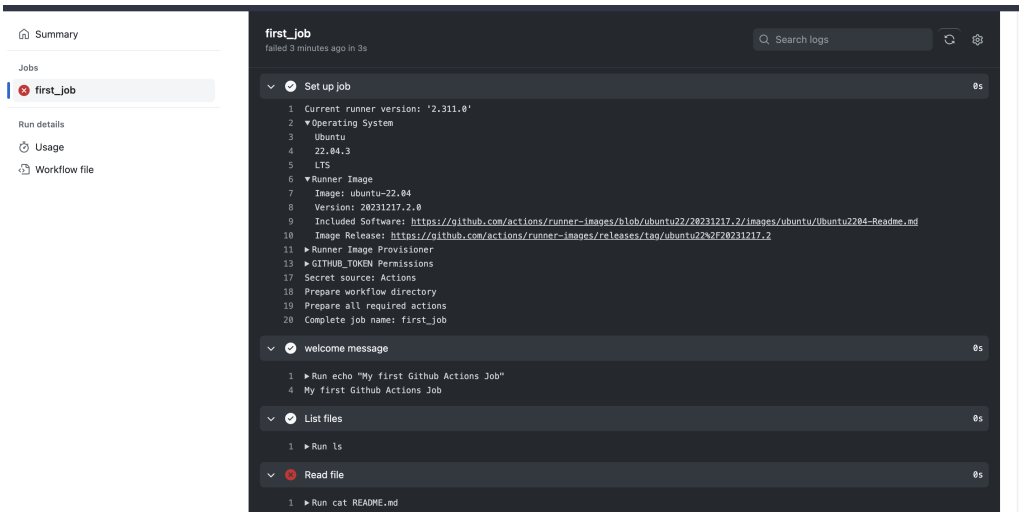
- name: Read file: This is the third step.

run: cat README.md: This command reads the contents of the file 'README.md' and prints it to the console.
In summary, when you push changes to your repository, this workflow will start. It will run on an Ubuntu server and execute three tasks: print a welcome message, list the files in the directory, and display the contents of your
README.md file. GitHub Actions can be very powerful for automating tests, deployments, and many other tasks in your development workflow.



The orange Dot is the workflow which is on the actions.



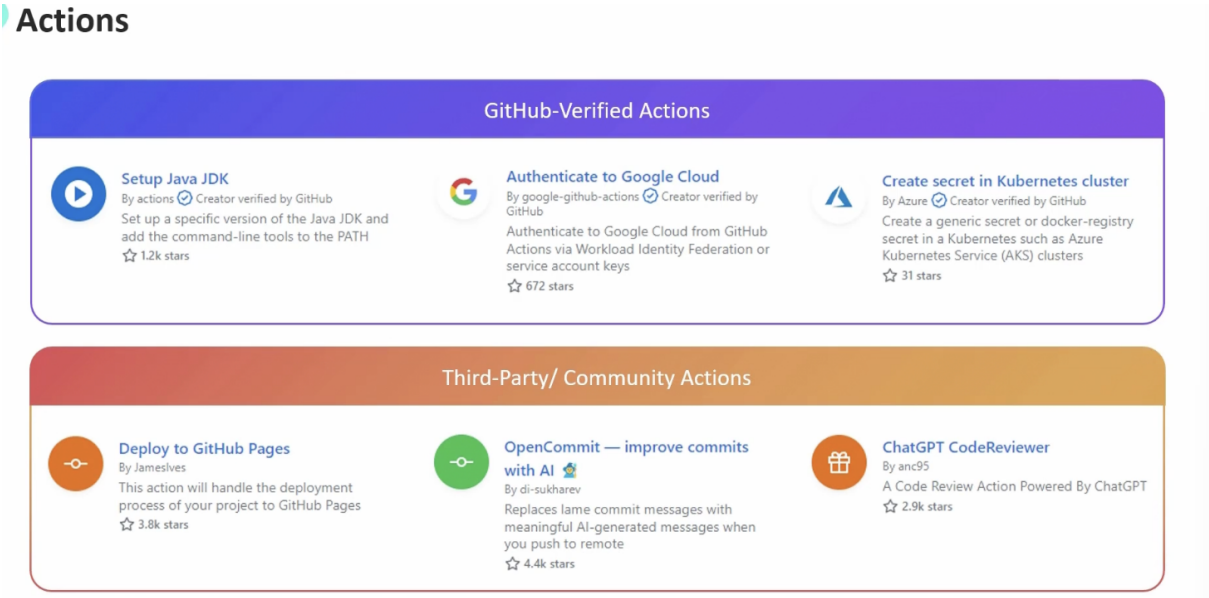


If we go on more details, then we will found out OS, Runner Image, github tokens etc.

EXIT code 1 means the Job is a failure

We need to explicitly allow the workflow to download the reposiotry and have access to the file. That is the reason it wasn't getting the README file.

Actions

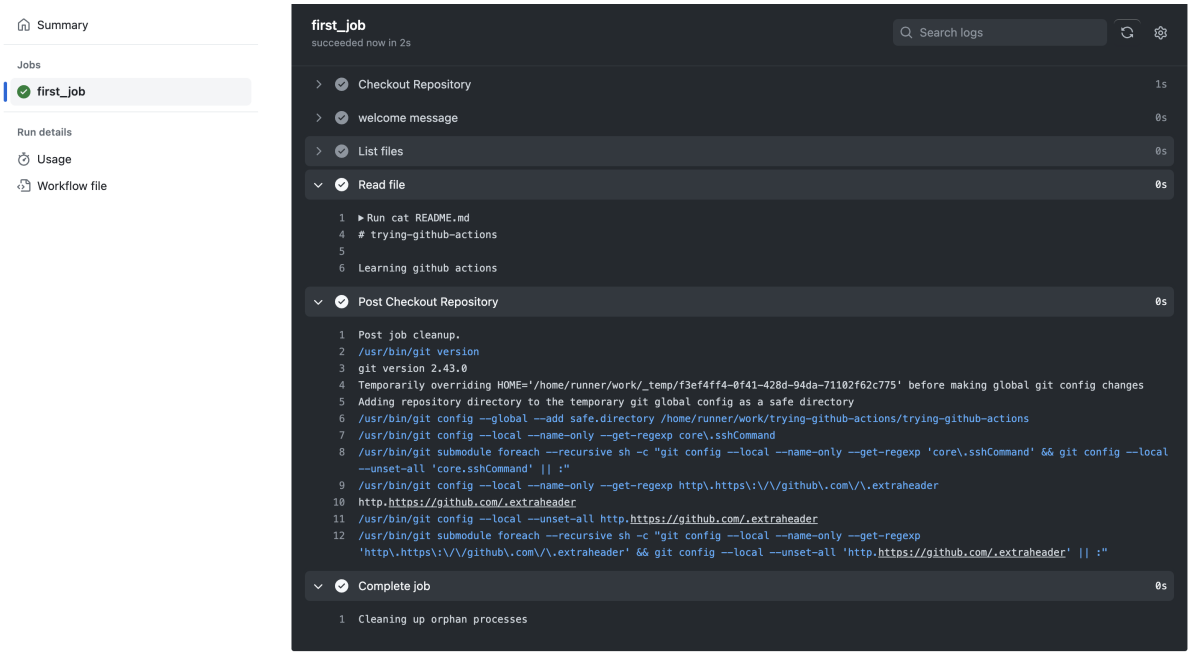


With tick are verified by github, and for third party, we need to ensure that the secrect are not shared with unintended hosts.

Sha == SHAH, sha are immutable and more reliable compared to tags and branches.

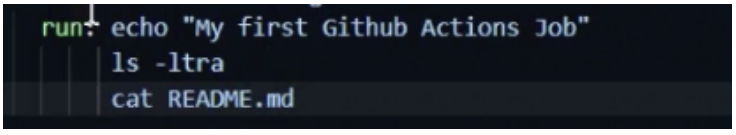
---

https://github.com/actions/checkout

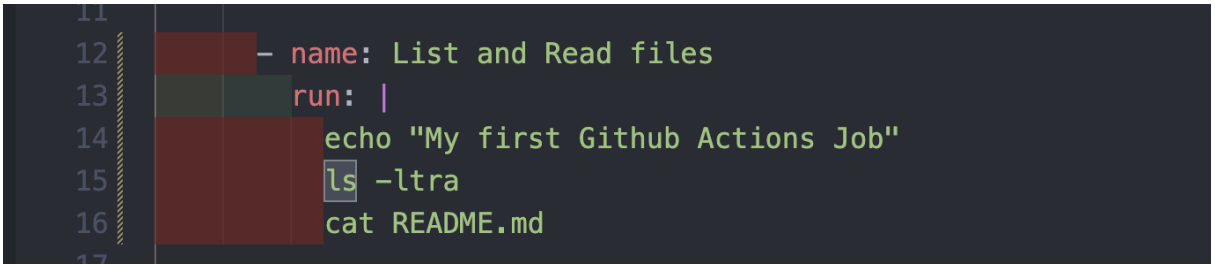



```
name: My first workflow

on: push

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

```
    steps:
      - name: Checkout Repository
        uses: actions/checkout@v4

      - name: welcome message
        run: echo "My first Github Actions Job"

      - name: List files
        run: ls

      - name: Read file
        run: cat README.md
```



If we write this code, it might not work, as this is a yaml specification, we need to use a pipe | which will execute the command in series, # is used for commenting.



1. `ls`: This is the basic command to list files and directories.
2. `-l`: This option tells `ls` to display the listing in a long format. This means it will include detailed information such as file permissions, number of links, owner name, group name, file size, time of last modification, and the file/directory name.
3. `-t`: This option sorts the list by modification time, showing the newest files first.
4. `-r`: This reverses the order of the sort. Combined with `-t`, it will show the oldest files first.
5. `-a`: This includes all files, including those that are hidden (files starting with a dot `.` are considered hidden in Unix/Linux systems).

So, when you run `ls -ltra`, it displays a detailed list of all files and directories, including hidden ones, sorted by the modification time with the oldest files shown first. This command is particularly useful for checking the details of files in a directory and seeing the oldest files at the top of the list.

```
- name: Generate ASCII Artwork using third party library
  run: cowsay -f dragon "I am the dragaon, go get Silver or Ice" >> dragon.txt
```

```
/home/runner/work/_temp/7436f434-dc89-4780-a1fe-0e6d560c4196.sh: line 1: cowsay: command not found
Error: Process completed with exit code 127.
```

The reason it failed is that it could find the command. The CLI cowsay is not preinstalled in the ubuntu machine and hence it could not run this. Any third party that is not installed in the VM, we need to add additional step manually to run it.

```
name: Generate ASCII Artwork

on: push

jobs:
  ascii_job:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout Repository
        uses: actions/checkout@v4

      - name: Install Cowsay Program
        run: sudo apt-get install cowsay -y

      - name: Execute Cowsay CMD
        run: cowsay -f dragon "Here comes the Dragon, Quick, find a Dr

      - name: Test file exists
```

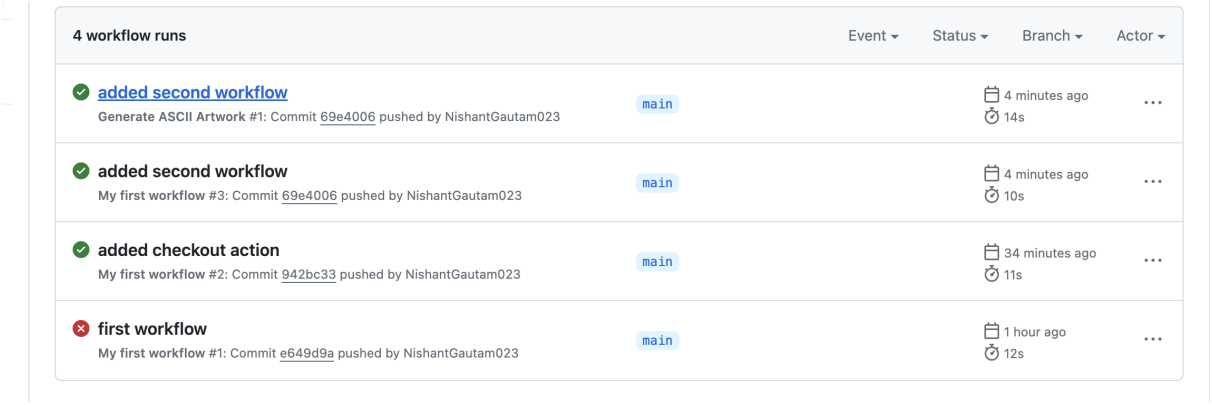
```
        run: grep -i "dragon" dragon.txt

      - name: Read File
        run: cat dragon.txt

      - name: checkout Repo files
        run: ls
```

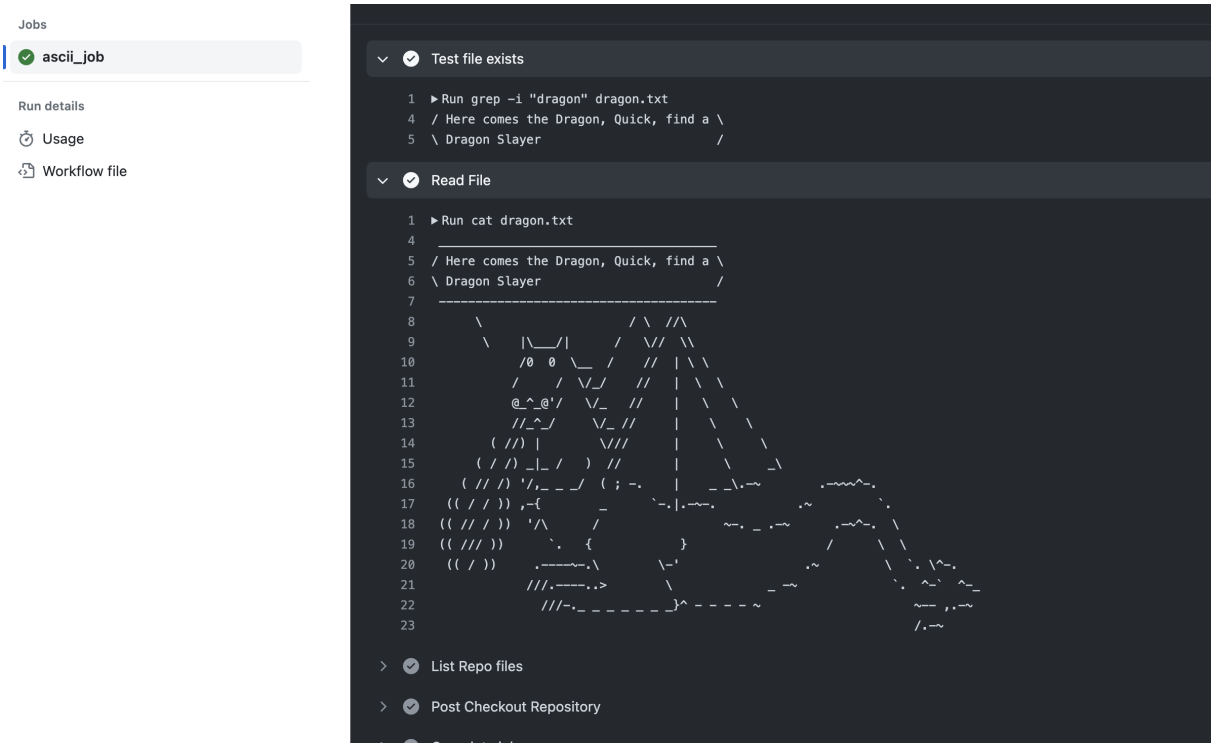
The name of the workflow is the Generate ASCII Artwork. It's going to execute whenever I push to this repository. It has a single job named ASCII and a Square job. It runs on open two latest virtual machine, and it has few steps. The first step it's going to check out the repository and in the second step via installing the CalSAE program. If you remember earlier when I tried to execute the Cal A CLI command, it returned an error message saying that the command could not be found. So for any library or program which you want to use in this ubuntu virtual machine, you need to install them. If it is not available as part of the pre-installed, you know, uh, software list. Okay? So I'm just using a command to install the cowsay CLI. Then I'm using a step to execute the command and generate some text and save the text in a file called as Dragon txt. And in the next step, I'm just testing if the file exists using the grep command, and then I'm going to road the fila and I'm just going to checkout the file.

The command `grep -i "dragon" dragon.txt` in Unix or Linux is used to search for the string "dragon" within the file `dragon.txt`. Here's a breakdown of each part of the command:

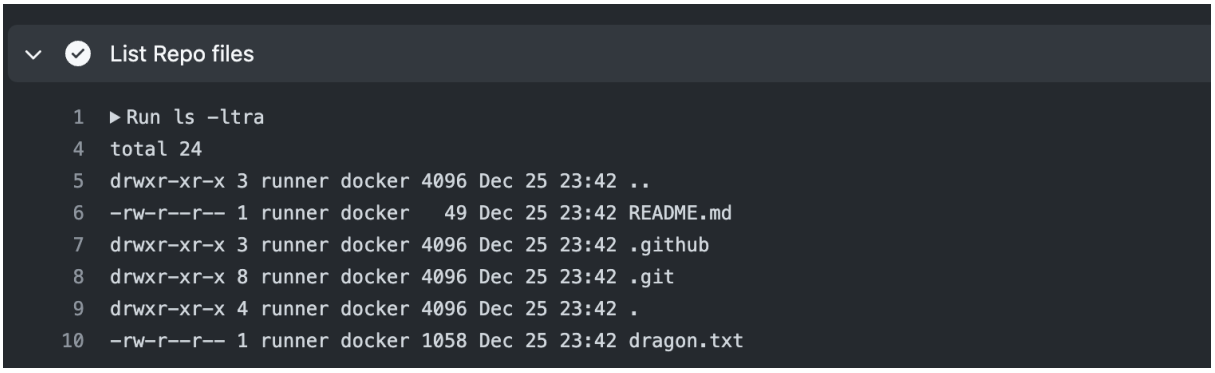
1. `grep`: This is the command used for searching text. `grep` stands for "Global Regular Expression Print." It scans through each line of the given file(s) and looks for matches to the pattern specified.
2. `-i`: This is an option that makes the search case-insensitive. Without `-i`, `grep` would only find matches that exactly match the case of the string. With `-i`, it will find 'dragon', 'Dragon', 'DRAGON', and any other case variations.
3. `"dragon"`: This is the search pattern. `grep` will search for this string in the specified file. The quotes around the word are used to ensure that the entire string is treated as a single argument by the shell.
4. `dragon.txt`: This is the name of the file in which the search will be conducted. `grep` will scan each line of `dragon.txt` looking for the pattern.

When this command is executed, `grep` will output every line in `dragon.txt` that contains the string "dragon", regardless of its case (uppercase or lowercase). If there are no matches, `grep` will not output anything. This command is particularly useful for searching through large text files or logs where you are looking for specific case-insensitive occurrences of a word or phrase.

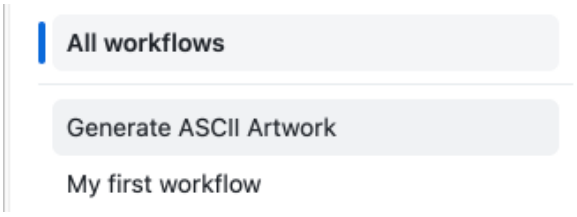




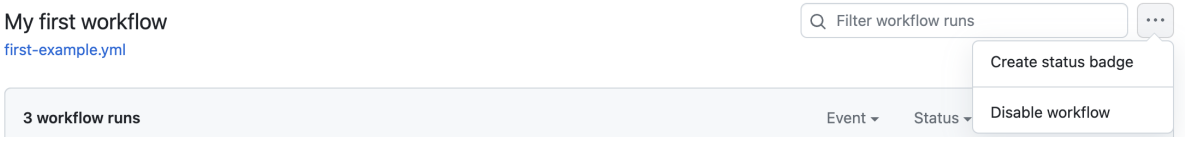
Here our both workflows are on push they are being executed.

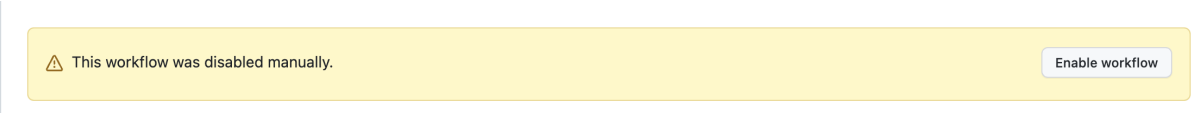


As the file is being generated in the ubuntu machine, once the virtual machine completes all the steps and the virtual machine closes, the file is also deleted, any file that are created or generated during the workflow execution, are automatically deleted when the VM closes.



Sometimes we wan't to disable old workflows, like we have 10 workflow and all have push, disabling old work which are not being used currently. CLick on the workflow that we want to disable, and click on disable to manually disable, thus during push event that wont generate.





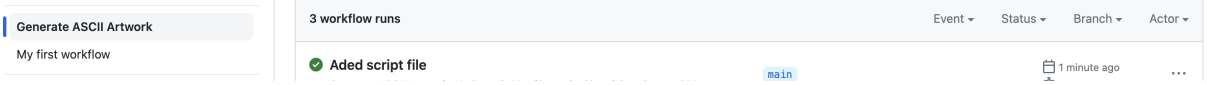
```
#!/usr/bin/env bash

cowsay_cli() {
  echo "The program has started running"

  # Install cowsay if it's not already installed
  if ! command -v cowsay >/dev/null; then
    sudo apt-get install cowsay -y
  fi

  # Create or overwrite dragon.txt with the output of cowsay
  cowsay -f dragon "Run for cover, I am a dragon.. RWAR" > dragon.txt

  # Check if the word "dragon" is in the file
  if grep -qi "dragon" dragon.txt; then
    echo "The word 'dragon' was found in dragon.txt"
  else
    echo "The word 'dragon' was not found in dragon.txt"
```

```
  fi

  # Display the contents of dragon.txt
  cat dragon.txt

  # List files in the current directory
  ls -ltra
}

# Call the function
cowsay_clicluttering the script's output with additional information.
```

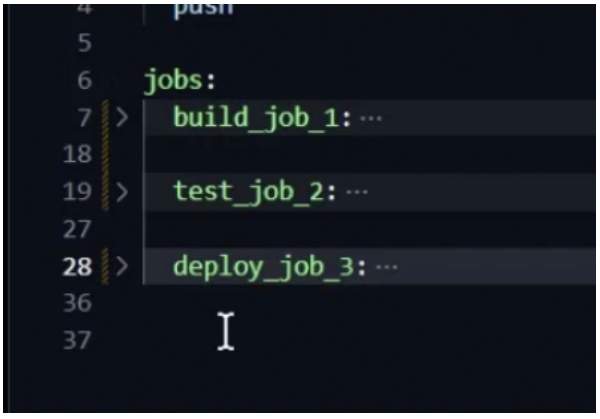
The `command -v` followed by a command name is a POSIX-compliant way to check if a command exists on your system, and `>/dev/null` is used to redirect any output of that command to a special file that discards all the data written to it. Let's break down these components:

1. `command -v`:
   - The `command` utility is used to run a command with a specified name, bypassing any shell function named after that command. The `-v` option makes `command` return a description of how the shell would invoke the named command, but it doesn't actually run the command.
   - In the context of `command -v cowsay`, it's used to check if the `cowsay` command exists and is executable in your system's PATH. It's a common way to check for the existence of a command in shell scripts.
   - If `cowsay` is found, `command -v cowsay` will output the path to the `cowsay` executable. If not, it will produce no output and return a non-zero exit status.
2. `>/dev/null`:
   - `>/dev/null` is a form of redirection in Unix and Linux systems. Anything that is redirected to `/dev/null` is essentially discarded.
   - `/dev/null` is a special file that automatically discards all data written to it. Think of it as a black hole for data.
   - When combined, `command -v cowsay >/dev/null` checks if `cowsay` exists and is executable, but it doesn't show the path to the `cowsay` executable in your terminal. Instead, it silently discards any output.
   - The primary use of this is to check for the existence of a command without

Together, these are used in scripts to quietly check the existence of commands, ensuring that the script can safely use those commands later without running into command-not-found errors. This technique is preferable in scripts where you want to maintain a clean and user-friendly output.



As of now our workflow is doing only a single job,

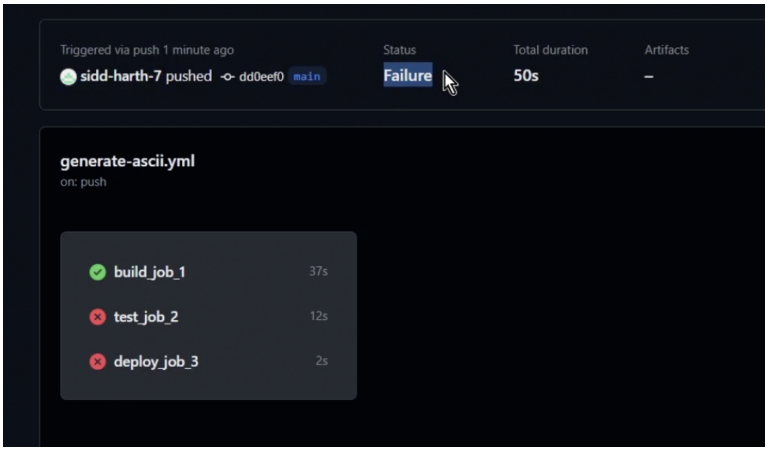


We have added 3 jobs, as generally in the CI CD we have the build phase, test phase and the deploy phase, each of them we need to use as a separate jobs,

```yaml
jobs:
  build_job_1:
    runs-on: ubuntu-latest
    steps:
    - name: Install Cowsay Program
      run: sudo apt-get install cowsay -y

    - name: Execute Cowsay CMD
      run: cowsay -f dragon "Run for cover, I

    - name: Sleep for 30 seconds
      run: sleep 30

  test_job_2:
    runs-on: ubuntu-latest
    steps:
    - name: Sleep for 10 seconds
      run: sleep 10

    - name: Test File Exists
      run: grep -i "dragon" dragon.txt
```

Here each job needs to run on it's own machine, they don't run on the same VM.

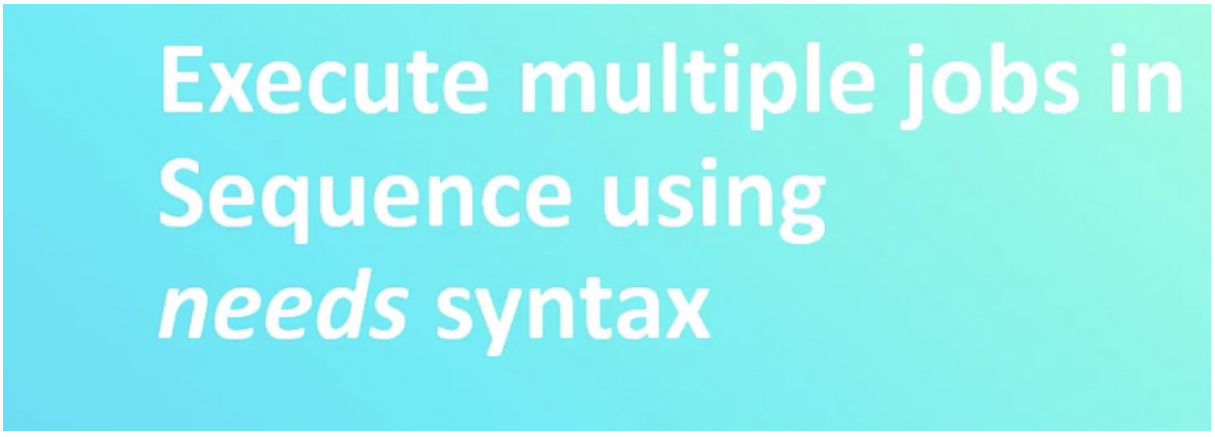
These 3 jobs will be running in parallel by default and they will not wait for any other jobs, But we want the build job first, then test then deploy.



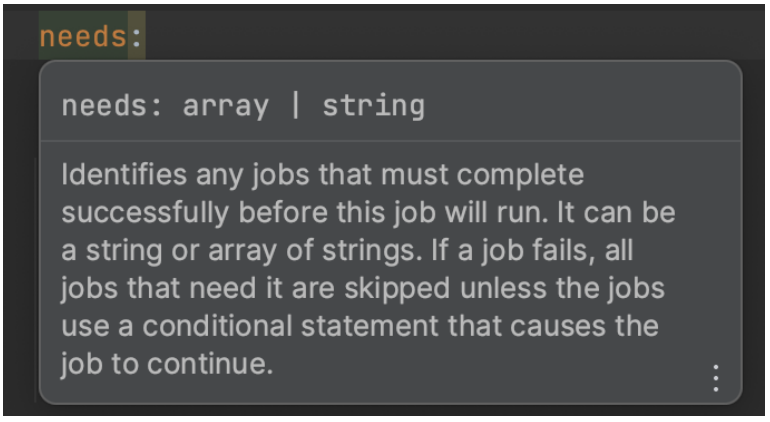
Here as only one job was successfull and the rest of the job failed, so the entire workflow status will be in failure,

---

Problem to solve: the jobs are getting executed in parallel, we need to execute them in sequence one after another, send issue is whatever file is created in one job cannot be accesible in another job.




```
test_job_2:
  needs:
```



```
push

jobs:
  build_job
    needs: build_job_1
```

Value 'build_job_1' is not valid

Job 'build_job_1' depends on job 'build_job_1' which creates a cycle in the dependency graph.

View Problem (Alt+F8) No quick fixes available

This will create an infinite loop or cycle in the dependency graph, it is a kind of validation which occurs before we publish the job. As the test job needs the build job and the build job needs the test job.

```
deploy_job3:
  needs: test_job2
```

if this job depends on multiple jobs we can define an array also. i.e.

needs: [test_job2, test_job1]

Status: Queueed: means waiting for the worker, after that the status changes.

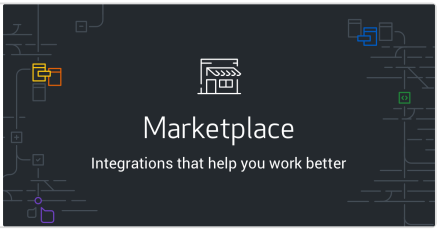
If the previous job that is dependent is failed, then the current job will be skipped.

This is because the files are not accessible as they are running on different machines. How do we share files from one job to another.

GitHub Marketplace: actions to improve your workflow

Find the actions that help your team build better, together.

https://github.com/marketplace?type=actions



Upload-Artifact v3

This uploads artifacts from your workflow allowing you to share data between jobs and store data once a workflow is complete.

GitHub Action

Download a Build Artifact

v2.1.1 Latest version

Download-Artifact v3

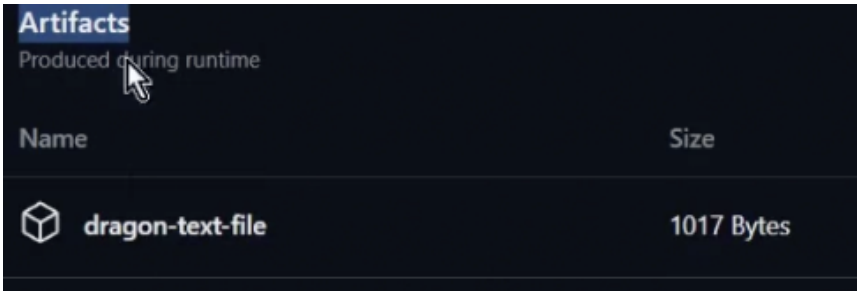
This downloads artifacts from your build

See also upload-artifact.

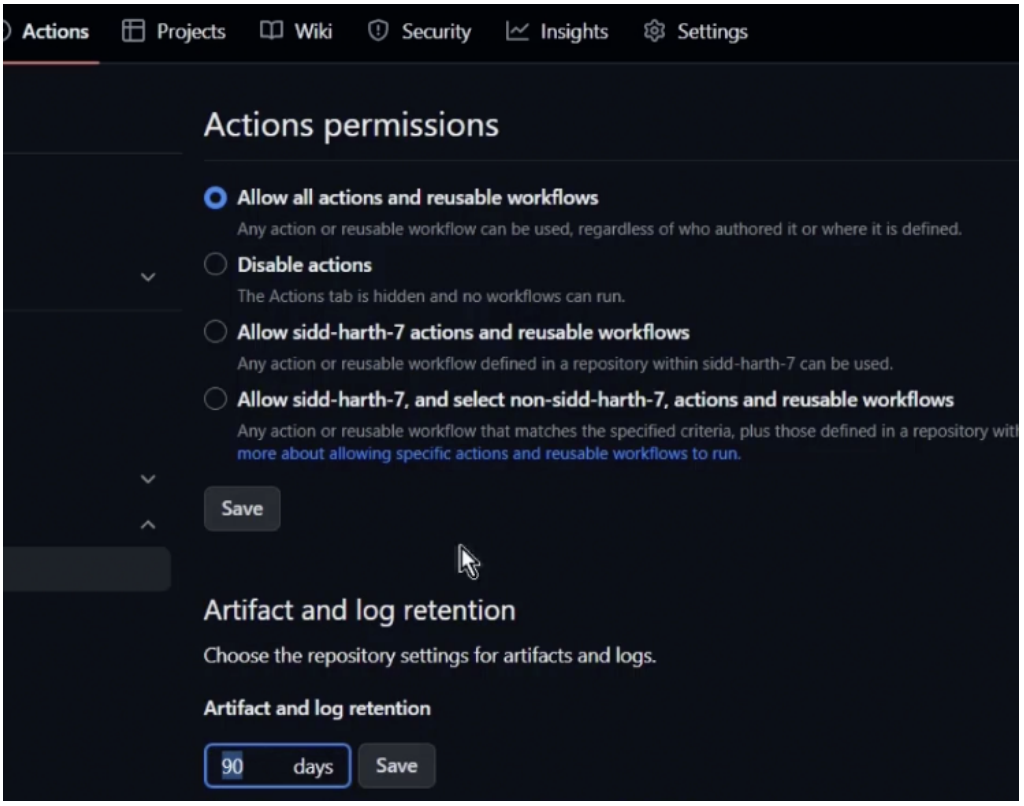
What's new

```
name: Running Multiple Jobs

on: push

jobs:

  build_job1:
    runs-on: ubuntu-latest
    steps:
      - name: Install Cowsay program
        run: sudo apt-get install cowsay -y

      - name: Execute Cowsay Command
        run: cowsay -f dragon "Run for cover, I am a dragon... RWARRR"

      - name: Upload the Dragon Text file
        uses: actions/upload-artifact@v3
        with:
            name: dragon-text-file
            path: dragon.txt

  test_job2:
    needs: build_job1
    runs-on: ubuntu-latest
    steps:
      - name: Download the Dragon Text file
        uses: actions/download-artifact@v3
```

```
        with:
          name: dragon-text-file


      - name: Test file exists
        run: grep -i "dragon" dragon.txt

  deploy_job3:
    needs: test_job2
    runs-on: ubuntu-latest
    steps:

      - name: Download the  Dragon Text file
        uses: actions/download-artifact@v3
        with:
          name: dragon-text-file


      - name: READ FILE
        run: cat dragon.txt

      - name: Print Hello
        run: echo "Successfully Deployed"
```



The artifact will be stored for default 90 days, we can change the action permissions



This is our free account, and size is only 500 for the free account.

```yaml
name: Exploring Variables and Secrets

on:
    push

jobs:
  docker:
    runs-on
    steps:
    - name:
      run: docker build -t docker.io/dockerUsername/imageName:latest

    - name: Docker Login
      run: docker login --username=dockerUsername --password=s3cUrePaSsw0rd

    - name: Docker Publish
      run: docker push docker.io/dockerUsername/imageName:latest

  deploy:
    needs: docker
    runs-on: ubuntu-latest
    steps:
    - name: Docker Run
      run: docker run -d -p 8080:80 docker.io/dockerUsername/imageName:latest

############### environment variables at STEP level ###############
```

Here 2 jobs docker job and deploy job, where deploy job is dependent on docker job. As in first job, we are logging in the job and using password, with dockerusername, imageName, and container registry name. In workflow we can store environment values for non sensitive data values such as dockerusername, imageName and container registry

```yaml
docker:
  runs-on: ubuntu-latest
  steps:
  - name: Docker Build
    env:
      CONTAINER_REGISTRY: docker.io
      DOCKER_USERNAME: siddharth1
      IMAGE_NAME: github-actions-nginx
    run: echo docker build -t $CONTAINER_REGISTRY/$DOCKER_USERNAME/$IMAGE_NAME:latest

  - name: Docker Login
    env:
      DOCKER_PASSWORD: s3cUrePaSsw0rd
      DOCKER_USERNAME: siddharth1
    run: echo docker login --username=$DOCKER_USERNAME --password=$DOCKER_PASSWORD

  - name: Docker Publish
    env:
      CONTAINER_REGISTRY: docker.io
      DOCKER_USERNAME: siddharth1
      IMAGE_NAME: github-actions-nginx
    run: echo docker push $CONTAINER_REGISTRY/$DOCKER_USERNAME/$IMAGE_NAME:latest
```

TO access that env variale, we can use $name or ${{ env.CONtainervalue }}

Varaibles such as API keys, passwords, tokens are sensitive in nature and should not be hardcoded we should make use of secrects.

We have defined the env variables on the step levels

As we are defining the env varible in the step level and we are repeating itself, we can avoid it by defining at the JOB level.

```yaml
  docker:
    env:
        CONTAINER_REGISTRY: docker.io
        DOCKER_USERNAME: siddharth1
        IMAGE_NAME: github-actions-nginx
    runs-on: ubuntu-latest
    steps:
    - name: Docker Build
      run: echo docker build -t $CONTAINER_REGISTRY/$DOCKER_USERNAME/$IMAGE_NAME:lat

    - name: Docker Login
      env:
        DOCKER_PASSWORD: s3cUrePaSsw0rd
      run: echo docker login --username=$DOCKER_USERNAME --password=$DOCKER_PASSWORD

    - name: Docker Publish
      run: echo docker push $CONTAINER_REGISTRY/$DOCKER_USERNAME/$IMAGE_NAME:latest

  deploy:
    env:
        CONTAINER_REGISTRY: docker.io
        DOCKER_USERNAME: siddharth1
        IMAGE_NAME: github-actions-nginx
    needs: docker
    runs-on: ubuntu-latest
```

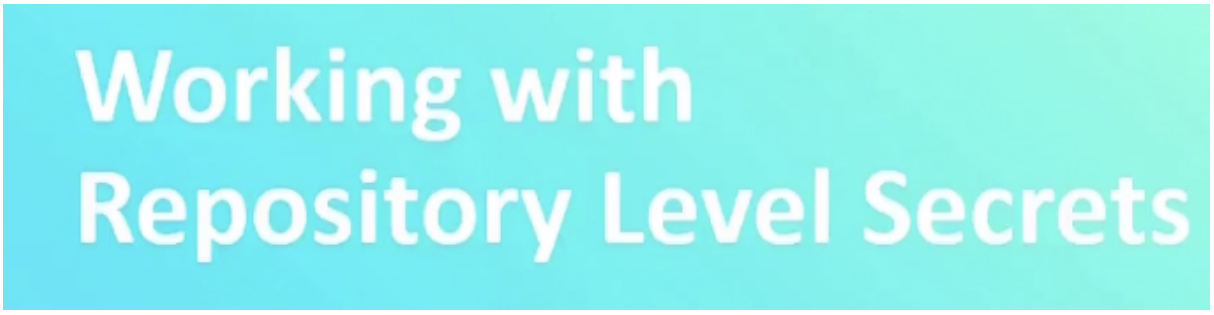
Env defined a the job level will be available ore acessible to all the steps wihin the JOB.

Using env varaible at the workflow level,

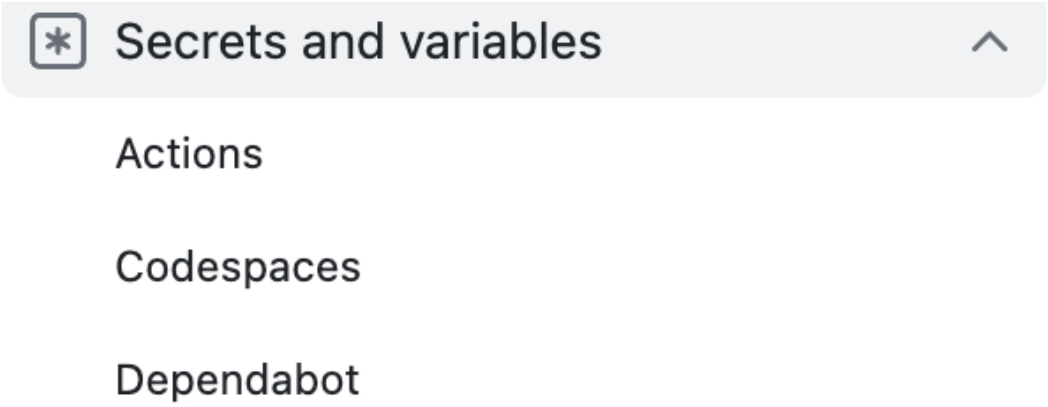
```yaml
name: Exploring Variables and Secrets

on:
    push

env:
    CONTAINER_REGISTRY: docker.io
    DOCKER_USERNAME: siddharth1
    IMAGE_NAME: github-actions-nginx
```

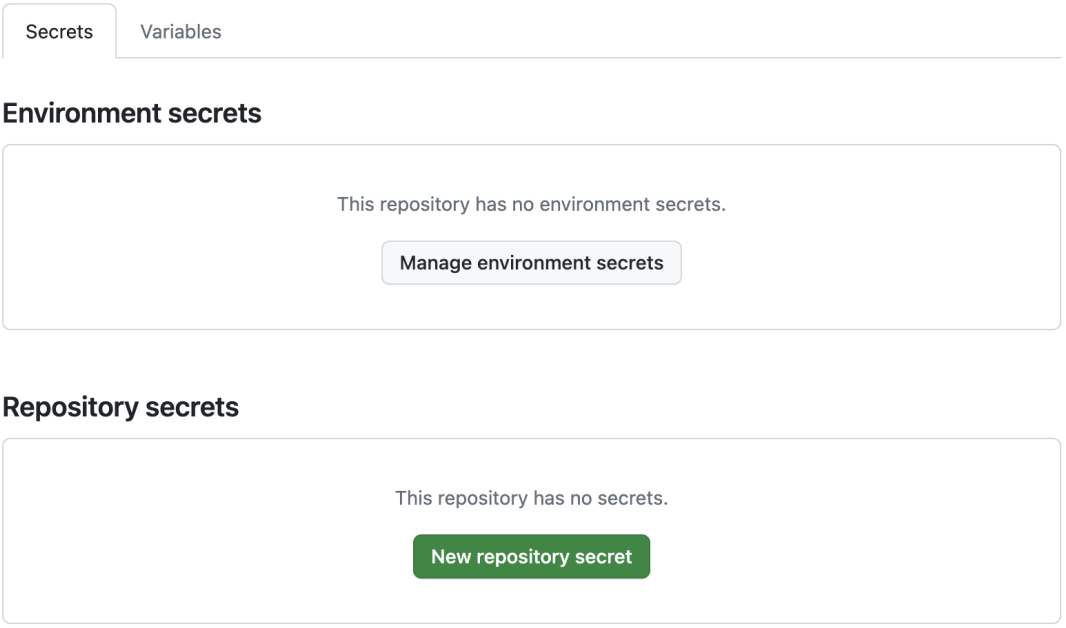
```yaml
jobs:
  docker:
    runs-on: ubuntu-latest
    steps:
    - name: Docker Build
      run: echo docker build -t ${{ env.CONTAINER_REGISTRY }}/$DOCKER_USERNAME/$IMAGE_NAME:latest

    - name: Docker Login
      env:
        DOCKER_PASSWORD: s3cUrePaSsw0rd
      run: echo docker login --username=$DOCKER_USERNAME --password=$DOCKER_PASSWORD

    - name: Docker Publish
      run: echo docker push $CONTAINER_REGISTRY/$DOCKER_USERNAME/$IMAGE_NAME:latest
```

At the workflow levels will be accesible to all the jobs.

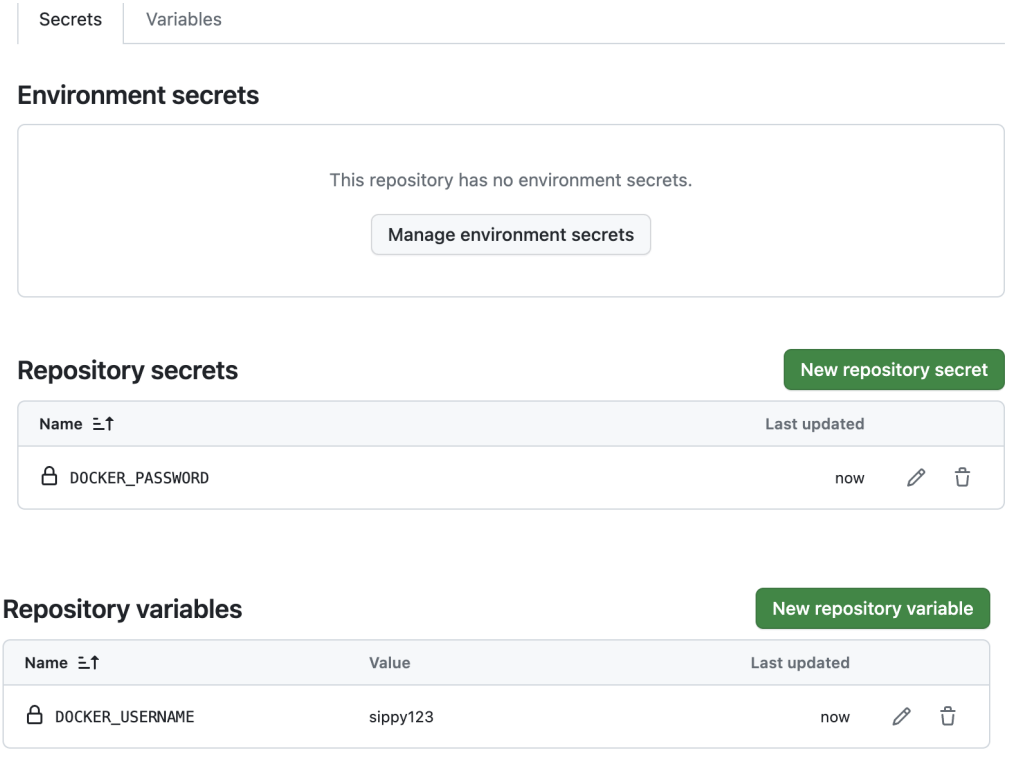


Go to the settings of the projects and click on secrects and variables.





We can use the variables and secrects in the repository level and env. The secrects are encrypted here.



The varibles key and value will be visible but for the secrects it wont be visible.

To refer the secrects.

```
name: Docker Login
run: echo docker login --username=$DOCKER_USERNAME --password=${{ secrets.DOCKER_PASSWORD }}
```

This is how we use defined in security level or repository level.

```
name: Exploring Variables and Secrets

on:
    push

env:
    CONTAINER_REGISTRY: docker.io
    IMAGE_NAME: github-actions-nginx

jobs:
  docker:
    runs-on: ubuntu-latest
    steps:
    - name: Docker Build
      run: echo docker build -t ${{ env.CONTAINER_REGISTRY }}/$DOCKER_USERNAME/$IMAGE_NAME:latest

    - name: Docker Login
      run: echo docker login --username=$DOCKER_USERNAME --password=${{ secrets.DOCKER_PASSWORD }}

    - name: Docker Publish
      run: echo docker push $CONTAINER_REGISTRY/$DOCKER_USERNAME/$IMAGE_NAME:latest
```

```
Docker Login
1  ▼ Run echo docker login --username=$DOCKER_USERNAME --***
2    echo docker login --username=$DOCKER_USERNAME --***
3    shell: /usr/bin/bash -e {0}
4    env:
5      CONTAINER_REGISTRY: docker.io
6      IMAGE_NAME: github-actions-nginx
7  docker login --username= --***
Docker Publish
1  ▶ Run echo docker push $CONTAINER_REGISTRY/$DOCKER_USERNAME/$IMAGE_NAME:latest
7  docker push docker.io/ github-actions-nginx:latest
```

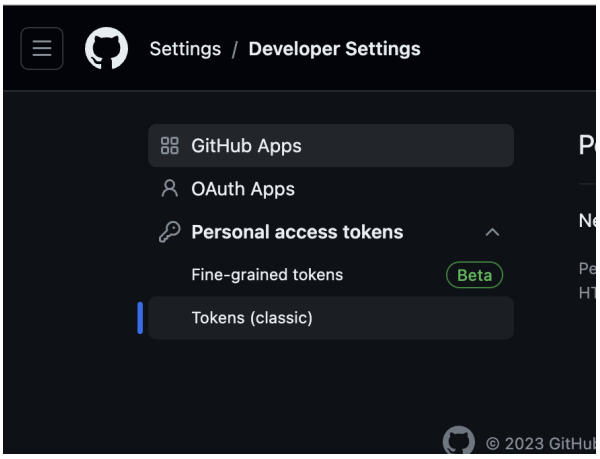
The username is empty, as we defined the repository level within the workflow, to access the repository, organization or env level varaibles we need to use another syntax.

```
name: Docker Publish
run: echo docker push $CONTAINER_REGISTRY/${{ vars.DOCKER_USERNAME }}/$IMAGE_NAME:latest
```

The warning is totally fine as the vscode has no idea where it is getting from.

To master a new technology, you will have to play with it.

– Jordan Peterson



Goto settings and then developer settings.

Managing your personal access tokens - GitHub Enterprise Server 3.6 Docs

You can use a personal access token in place of a password when authenticating to GitHub in the command line or with the API.

https://docs.github.com/en/enterprise-server@3.6/authentication/keeping-your-account-and-data-secure/managing-your-personal-access-tokens

---

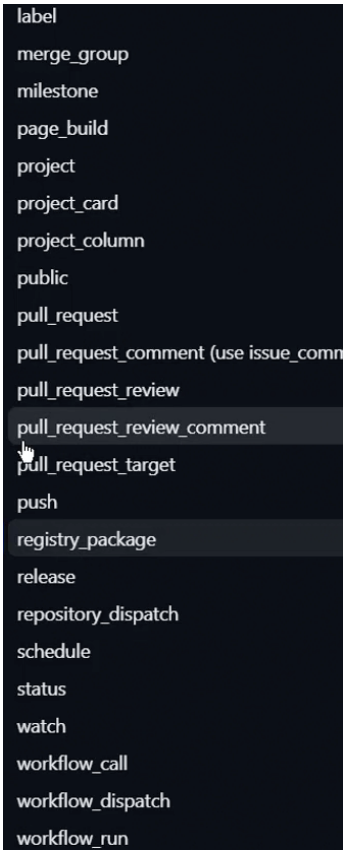
Lab 01

Triggering a workflow - GitHub Docs

How to automatically trigger GitHub Actions workflows

https://docs.github.com/en/actions/using-workflows/triggering-a-workflow

Many different types of events available such as:



ctrl + spacebar, to see all the options available here.

Crontab.guru - The cron schedule expression editor

An easy to use editor for crontab schedules.

https://crontab.guru/

A `cron job` is a scheduled task in Unix-like operating systems, commonly used for automating system maintenance or administration tasks. It is a time-based job scheduler that enables users to execute scripts or commands at specified times and intervals.

Here's a brief overview:

1. **Cron Daemon**: The background service that runs on Unix and Unix-like systems. This daemon is responsible for executing cron jobs at scheduled times.
2. **Crontab (Cron Table)**: This is the file where cron jobs are written and scheduled. Each user on the system can have their own crontab file, and there's also a system-wide crontab for tasks affecting the entire system.
3. **Syntax of a Cron Job**: A cron job entry consists of a series of fields that specify when the job will be run and what command should be executed. The typical structure is:

- Minute (0 - 59)
- Hour (0 - 23)
- Day of the month (1 - 31)
- Month (1 - 12)
- Day of the week (0 - 7, where both 0 and 7 represent Sunday)
- Command to execute

4. **Example of a Cron Job**: If you wanted to schedule a script to run at 5:30 AM every Monday, your cron job entry would look something like this:

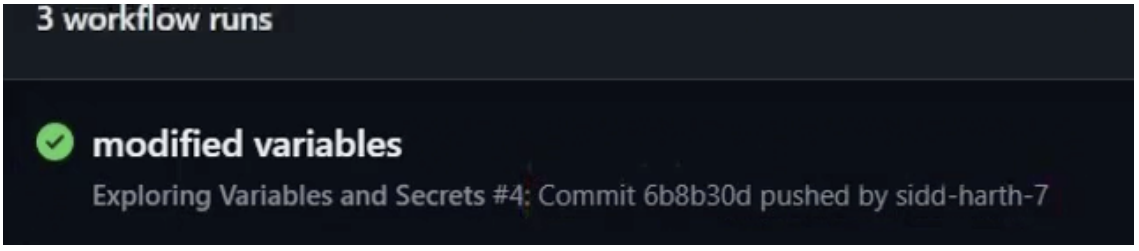
   ```
   bashCopy code
   30 5 * * 1 /path/to/script.sh
   ```

   This tells the system to run `/path/to/script.sh` at 30 minutes past 5 in the morning on every Monday.

5. **Common Uses**: Cron jobs are widely used for automating system backups, monitoring disk space, performing regular cleaning of temporary files, and running maintenance tasks like updating software.

6. **Editing Crontab**: Users typically edit their crontab files using the `crontab -e` command, which opens the crontab file in the default text editor.

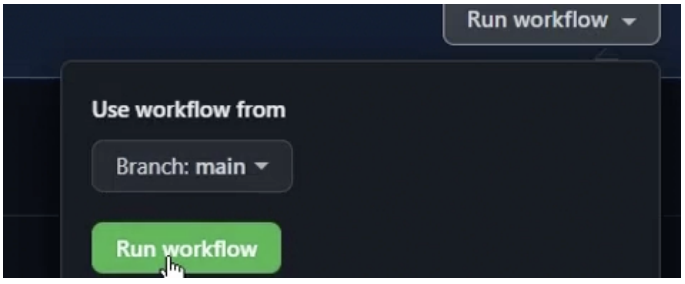
Cron jobs are powerful tools for system administrators and users who need to automate repetitive tasks on their servers or personal machines.



the log which is the pushed event.

```yaml
on:
  workflow_dispatch:
  # schedule:
  #   - cron: '*/1 * * * *'
```

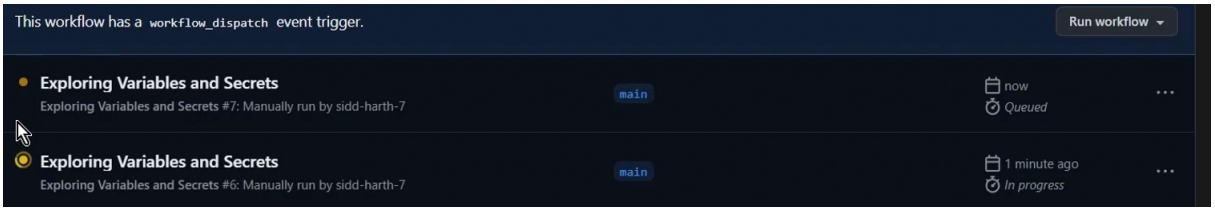
A new buttoon will be availoable tro manually trigger once it is pushed.



As this is manaually triggered, if a job is triggered by a scheduler it will show the schedular, and if the job request is made by the pull request it will show the pull request.

based on the event which are triggered for the workflow it will display the event name.

A job might take 10 min to run, meanwhile other people have pushed something and new trigger might have triggered for the workflow.



Both workflow are exeuting the job, we don't want to have 2 deployments having at the same time. bad Practice. This is where concurrency help us.

Using concurrency

Run a single job at a time.

Overview

You can use `jobs.<job_id>.concurrency` to ensure that only a single job or workflow using the sam concurrency group will run at a time. A concurrency group can be any string or expression. Allowed expression contexts: `github`, `inputs`, `vars`, `needs`, `strategy`, and `matrix`. For more informatior about expressions, see "Expressions."

```
needs: docker
concurrency:
  group: production-deployment
  cancel-in-progress: true
```

docker 1s — deploy 55s

Annotations

2 errors

deploy
Canceling since a higher priority waiting request for 'production-deployment' exists

deploy
The operation was canceled.

When in waiting mode, till previous running workflow completes the job:

```
Waiting: This workflow is waiting on deploy to complete before running.
```

Timeout for Jobs and Steps

```
- name: Docker Run
  run: |
    echo docker run -d -p 8080:80 $CONTAINER_REGISTRY/${{ vars.DOCKER_USERNAME }}/$IMAGE_NAME:latest
    sleep 6000s
```

If we done this by mistake, the workflow will run for 10 hours, by default github actions, will kill the workflow after 6 hours, sometimes due to memeory issue the workflow keeps running itelse, after that it will kill you after 6 hours, it is gonna cost our time and money also.

using timeout to make sure that the job to complete within the specified time else cancel or terminate the job.

```
timeout-minutes: 1
```

This step to complete in 1 min, if it takes more than 1 min, it will terminate the job.

```
deploy:
  needs: docker
  concurrency:
    group: production-deployment
    cancel-in-progress: false
  runs-on: ubuntu-latest
  steps:
  - name: Docker Run
    timeout-minutes: 1
    run: |
      echo docker run -d -p 8080:80 $CONTAINER_REGISTRY/${{ vars.DOCKER_USERNAME }}/$IMAGE_NAME:latest
      sleep 6000s
```

```
Docker Run
1  ▶ Run echo docker run -d -p 8080:80 $CONTAINER_REGISTRY/siddharth67/$IMAGE_NAME:latest
8  docker run -d -p 8080:80 docker.io/siddharth67/github-actions-nginx:latest
9  Error: The action has timed out.

Complete job
```

Using Matrix Strategy

```yaml
name: Matrix Configuration

on:
  push:
  workflow_dispatch:

jobs:
    deploy-on-ubuntu:
      runs-on: ubuntu-latest
      steps:
      - name: Echo Docker Details
        run: docker info

      - name: Run Images
        run: docker run hello-world

    deploy-on-windows:
      runs-on: windows-latest
      steps:
      - name: Echo Docker Details
        run: docker info

      - name: Run Images
        run: docker run hello-world
```

Since we will be repating same line of code for multiple oS and version, to avoid that, we have.

Using a matrix for your jobs

Create a matrix to define variations for each job.

About matrix strategies

A matrix strategy lets you use variables in a single job definition to automatically create multiple job runs that are based on the combinations of the variables. For example, you can use a matrix strategy to test your code in multiple versions of a language or on multiple operating systems.

Using a matrix strategy

Use `jobs.<job_id>.strategy.matrix` to define a matrix of different job configurations. Within your matrix, define one or more variables followed by an array of values. For example, the following matrix has a variable called `version` with the value `[10, 12, 14]` and a variable called `os` with the value `[ubuntu-latest, windows-latest]`:

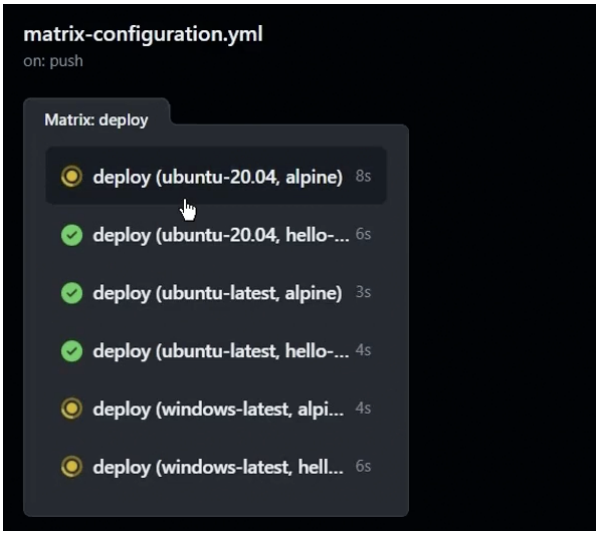
```
jobs:
  example_matrix:
    strategy:
      matrix:
        version: [10, 12, 14]
        os: [ubuntu-latest, windows-latest]
```

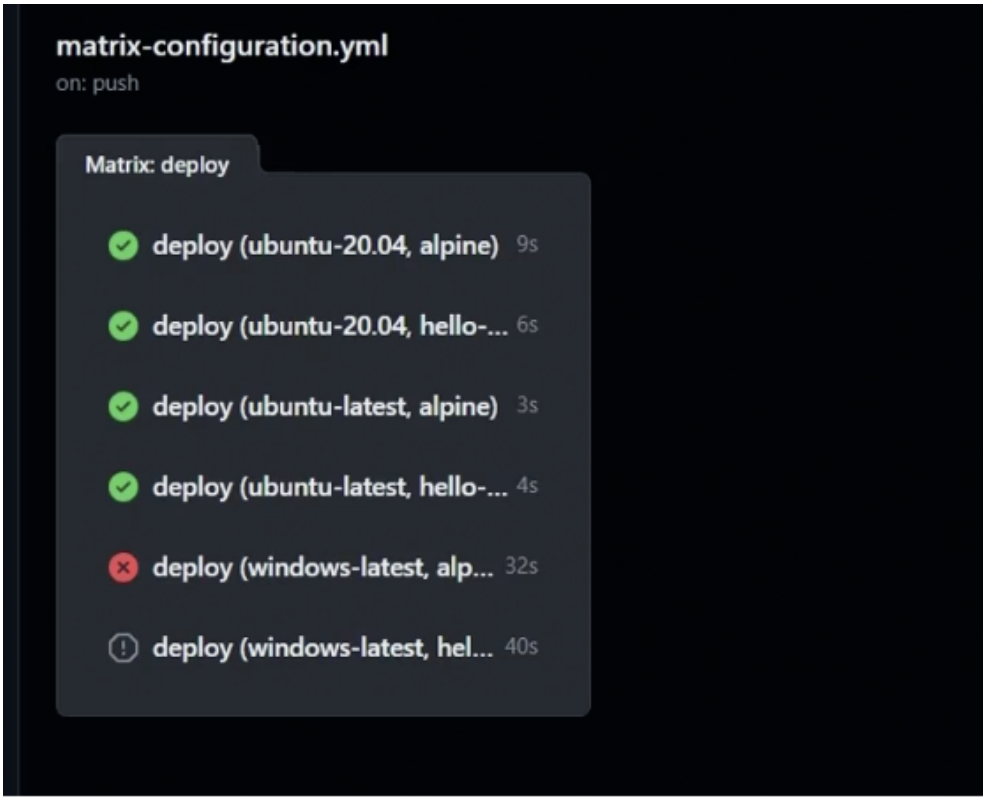
```
  strategy:
    matrix:
      os: [ubuntu-latest, ubuntu-20.04, windows-latest]
      images: [hello-world, alpine]
```

in the runs on we need to reference the value of the matrix.

```
jobs:
    deploy:
      strategy:
        matrix:
          os: [ubuntu-latest, ubuntu-20.04, windows-latest]
          images: [hello-world, alpine]
      runs-on: ${{ matrix.os }}
      steps:
      - name: Echo Docker Details
        run: docker info

      - name: Run Image on ${{ matrix.os }}
        run: docker run ${{ matrix.images }}
```

It will create 6 different jobs 3 *2, 3 os 2 images, so 6 jobs in parallel.





The default behaviour is, any job that are in progress will be canceled, if a job fails, and the entire workflow will be a failure state.

WE HAVE also options for ignoring certain images when using the matrix strategy.

Additional Matrix Strategy

In the previous, our windows machine is failing as it could not load the alpine image.

We have exclude and include .

```
exclude:
  - images: alpine
    os: windows-latest
```

The alpine-image will not run on the windows image, only the hello world. Thus only 5 jobs will run in parallel.

```
matrix:
  os: [ubuntu-latest, ubuntu-20.04, windows-latest]
  images: [hello-world, alpine]
  exclude:
    - images: alpine
      os: windows-latest
  include:
    - images: amd64/alpine
      os: ubuntu-20.04
```

Setting `fail-fast` to `false` prevents GitHub from canceling all in-progress jobs if any matrix job fails. Default: `true`

Available expression contexts: `github`, `inputs`, `vars`, `needs`

```
fail-fast:
```

The maximum number of jobs that can run simultaneously when using a matrix job strategy. By default, GitHub will maximize the number of jobs run in parallel depending on runner availability.

Available expression contexts: `github`, `inputs`, `vars`, `needs`

```
max-parallel:
```

max-parallel: 2.

Access Workflow Context Information

Contexts are a way to access information about workflow runs, variables, runner environments, jobs, and steps. Each context is an object that contains properties, which can be strings or other objects.

Contexts, objects, and properties will vary significantly under different workflow run conditions. For example, the `matrix` context is only populated for jobs in a matrix.

You can access contexts using the expression syntax. For more information, see "Expressions."

```
${{ <context> }}
```

`runner` context

The `runner` context contains information about the runner that is executing the current job.

Property name	Type	Description
`runner`	`object`	This context changes for each job in a workflow run. This object contains all the properties listed below.
`runner.name`	`string`	The name of the runner executing the job. This name may not be unique in a workflow run as runners at the repository and organization levels could use the same name.
`runner.os`	`string`	The operating system of the runner executing the job. Possible values are `Linux`, `Windows`, or `macOS`.
`runner.arch`	`string`	The architecture of the runner executing the job. Possible values are `X86`, `X64`, `ARM`, or `ARM64`.
`runner.temp`	`string`	The path to a temporary directory on the runner. This directory is emptied at the beginning and end of each job. Note that files will not be removed if the runner's user account does not have permission to delete them.
`runner.tool_cache`	`string`	The path to the directory containing preinstalled tools for GitHub-hosted runners. For more information, see "Using GitHub-hosted runners".
`runner.debug`	`string`	This is set only if debug logging is enabled, and always has the value of `1`. It can be useful as an indicator to enable additional debugging or verbose logging in your own job steps.

Using *if* expression in Jobs

```
1   ▶ Run echo "$GITHUB_CONTEXT"
218 {
219   "token": "***",
220   "job": "dump_contexts_to_log",
221   "ref": "refs/heads/main",
222   "sha": "ab3c0b9ccd32cb80154a5e48e279bad3cf8c6460",
223   "repository": "sidd-harth-7/actions-1",
224   "repository_owner": "sidd-harth-7",
225   "repository_owner_id": "147390322",
226   "repositoryUrl": "git://github.com/sidd-harth-7/actions-1.git",
227   "run_id": "6492400732",
228   "run_number": "1",
229   "retention_days": "90",
230   "run_attempt": "1",
231   "artifact_cache_size_limit": "10",
232   "repository_visibility": "public",
233   "repo-self-hosted-runners-disabled": false,
234   "enterprise-managed-business-id": "",
235   "repository_id": "702448262",
236   "actor_id": "147390322",
237   "actor": "sidd-harth-7",
```

We can make use of the context variables using the expressions,

Expressions

You can evaluate expressions in workflows and actions.

About expressions

You can use expressions to programmatically set environment variables in workflow files and access contexts. An expression can be any combination of literal values, references to a context, or functions. You can combine literals, context references, and functions using operators. For more information about contexts, see "Contexts."

Expressions are commonly used with the conditional `if` keyword in a workflow file to determine whether a step should run. When an `if` conditional is `true`, the step will run.

You need to use specific syntax to tell GitHub to evaluate an expression rather than treat it as a string.

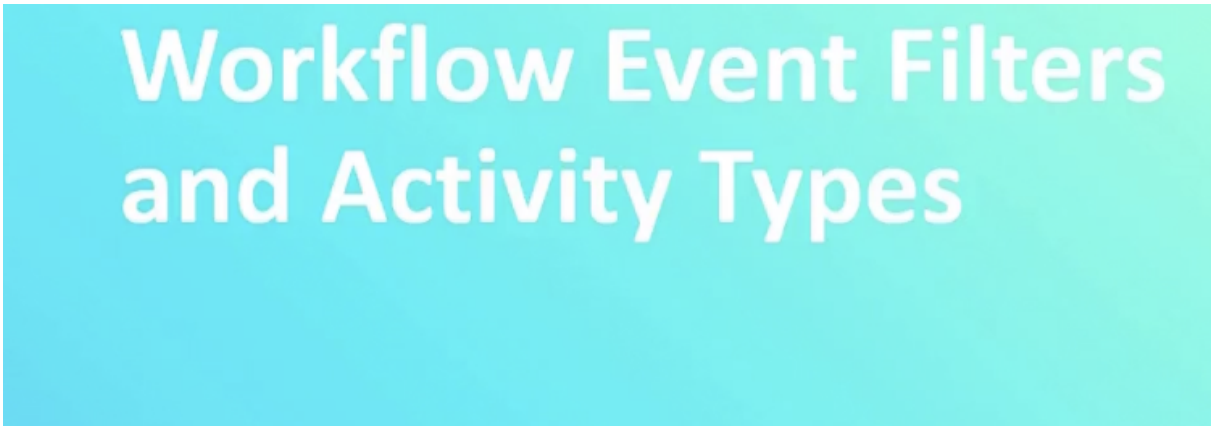
```
${{ <expression> }}
```

SUppose only run a deployment job if the chnages are in the main branch.

1

```yaml
deploy:
  if: github.ref == 'refs/heads/main'
  needs: docker
  concurrency:
    group: production-deployment
```

Now the deploy job will only execute if it is a part of the main branch.




```yaml
push:
  branches-ignore:
    - feature/*     # feature/add-music, feature/updateImages
    - test/**       # test/ui/index, test/checkout/payment/
  branches:
    - main
    - '!feature/*'    # ignoring pushing to any branch name starting with feature using !
pull_request: ···
```

It will ignore and will not trigger the workflow.

```yaml
pull_request:
  types:
    - opened
    - closed
  paths-ignore:       #  workflow will only run when a pull request that includes a change on any file other than README.md
    - README.md
  branches:
      - main          # configures your workflow to only run on pull requests that target specific branches.
```

.github > workflows > workflow-filters.yml

```yaml
name:  Workflow Filters and Activities

on:
  workflow_dispatch:
#   schedule:
#     - cron: "*/59 * * * *"
  push:
    branches:
        - main
        - '!feature/*'    # ignoring pushing to any branch name starting with feature using !
    # branches-ignore:
    #   - feature/*     # feature/add-music, feature/updateImages
    #   - test/**       # test/ui/index, test/checkout/payment/

  pull_request:
    types:
      - opened
      - closed
    paths-ignore:       #  workflow will only run when a pull request that includes a change on any file other than README.
      - README.md
    branches:
        - main          # configures your workflow to only run on pull requests that target specific branches.

jobs:
  hello:
    runs-on: ubuntu-latest
    steps:
    - run: echo this workflow/job/step is executed for event type -  ${{ github.event_name }}
```

github > workflows > workflow-filters.yml

```yaml
name:  Workflow Filters and Activities

on:
  workflow_dispatch:
#   schedule:
#     - cron: "*/59 * * * *"
  push:
    branches:
        - main
        - '!feature/*'    # ignoring pushing to any branch name starting with feature using !
    # branches-ignore:
    #   - feature/*     # feature/add-music, feature/updateImages
    #   - test/**       # test/ui/index, test/checkout/payment/

  pull_request:
    types:
      - opened
      - closed
    paths-ignore:       #  workflow will only run when a pull request that includes a change on any file other than README.md
      - README.md
    branches:
        - main          # configures your workflow to only run on pull requests that target specific branches.

jobs:
  hello:
    runs-on: ubuntu-latest
    steps:
    - run: echo this workflow/job/step is executed for event type -  ${{ github.event_name }}
```

Cancelling and Skipping Workflows

Skipping workflow runs

You can skip workflow runs triggered by the `push` and `pull_request` events by including a command in your commit message.

> **Note:** If a workflow is skipped due to path filtering, branch filtering or a commit message (see below), then checks associated with that workflow will remain in a "Pending" state. A pull request that requires those checks to be successful will be blocked from merging.

Workflows that would otherwise be triggered using `on: push` or `on: pull_request` won't be triggered if you add any of the following strings to the commit message in a push, or the HEAD commit of a pull request:

- `[skip ci]`
- `[ci skip]`
- `[no ci]`
- `[skip actions]`
- `[actions skip]`

Alternatively, you can end the commit message with two empty lines followed by either:

- `skip-checks:true`
- `skip-checks: true`

Another apporach is to just cancel in the UI-Level.

Fork the Repository - https://github.com/kodekloudhub/github-actions-lab2

- Notice the Repository name is - `github-actions-lab2`
- Go through the repository, there is a workflow file named `python-installation.yml`

1. There is one job named - `build`
2. The Job includes one step named Set up Python to install python of version `3.7`
3. Go to Actions Tab click on I understand my workflows, go ahead and enable them

Do the following:

Configure matrix strategy to set up a build matrix for this job/step using the below spec:

1. This job should execute on these operating systems - `ubuntu-latest, macos-latest, windows-latest`
2. Use os as the key to define these operating systems within the matrix.

3. The Set up Python step should install python-version `3.7, 3.8, 3.9`
4. Use `py-version` as the key to define these `python-version` within the matrix.
5. Modify the job and step to make use of these matrix values.
6. Push and trigger the workflow.

```
name: Installing Python
'on':
  - push
jobs:
  build:
    runs-on: '${{ matrix.os }}'
    strategy:
      matrix:
        os:
          - ubuntu-latest
          - macos-latest
          - windows-latest
        py-version:
          - 3.7
          - 3.8
          - 3.9
    steps:
      - name: Set up Python
        uses: actions/setup-python@v4
        with:
          python-version: '${{ matrix.py-version }}'
```

Continue work on the Repository named - github-actions-lab2

Configure matrix strategy to **exclude** below combinations:

- os: `macos-latest`
- py-version: `3.9`

```
name: Installing Python
'on':
  - push
jobs:
  build:
```

```
    runs-on: '${{ matrix.os }}'
    strategy:
      matrix:
        os:
          - ubuntu-latest
          - macos-latest
          - windows-latest
        py-version:
          - 3.7
          - 3.8
          - 3.9
        exclude: #added
          - os: macos-latest #added
            py-version: 3.9 #added
    steps:
      - name: Set up Python
        uses: actions/setup-python@v4
        with:
          python-version: '${{ matrix.py-version }}'
```