Table of contents

GitHub Actions

What is GitHub Actions

GitHub Actions is a workflow automation service offered by GitHub, to automate all kinds of repository related processes and actions.

Repository

Is a bucket that contains the code.

Use Cases for GitHub Actions

Code Deployment (CI/CD)

- Automate code testing, building & deployment.
- Combined, they group methods for automating app development and deployment.
- Code changes are automatically built, tested and merged with existing code (CI)
- After integration, new app or package versions are published automatically.

Code & Repository Management

- Automate Code Reviews, issues management

GitHub Actions Key Element

There are three main key elements in GitHub:

- Workflows
- Jobs
- Steps

Workflows

- Workflows are attached to a GitHub repository
- They Contain one or more jobs
- They are triggered upon Events

Jobs

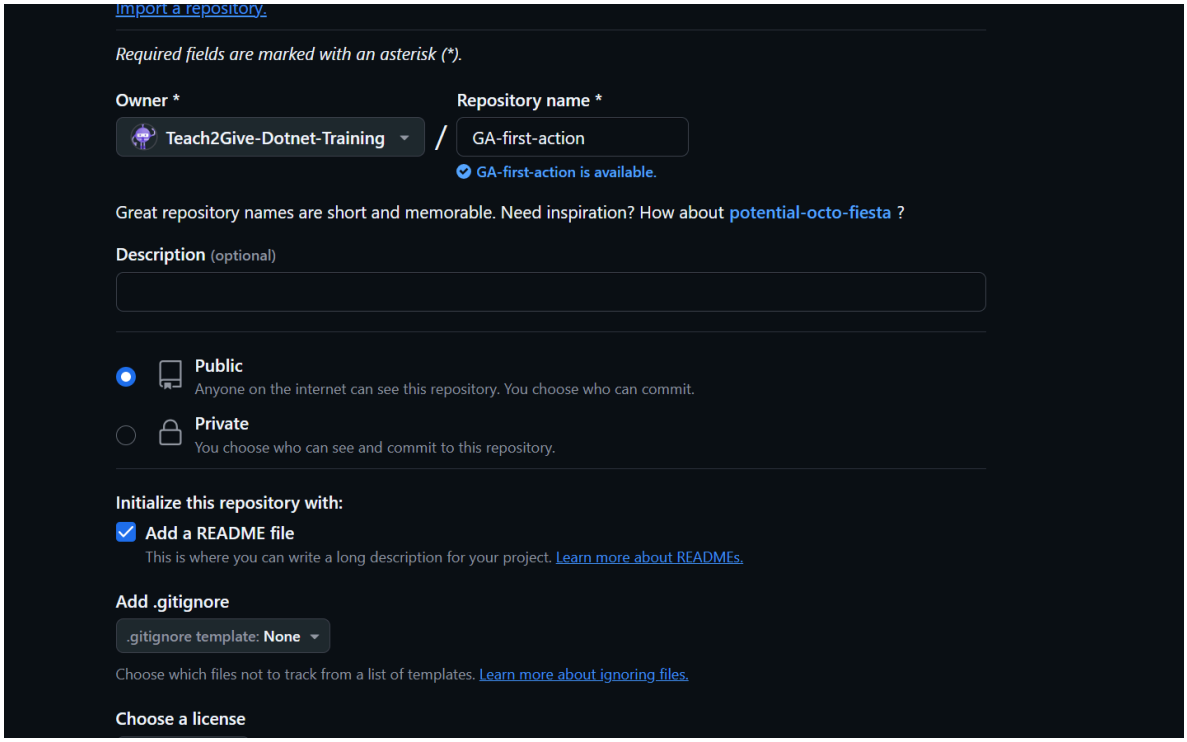
- Jobs define a runner(execution environment)
- Contain one or more steps
- Run in parallel(default) or sequential.
- Can be conditional

Steps

- Execute a shell script or an action.
- Can use custom or third party actions
- Steps are executed in order
- Can be conditional

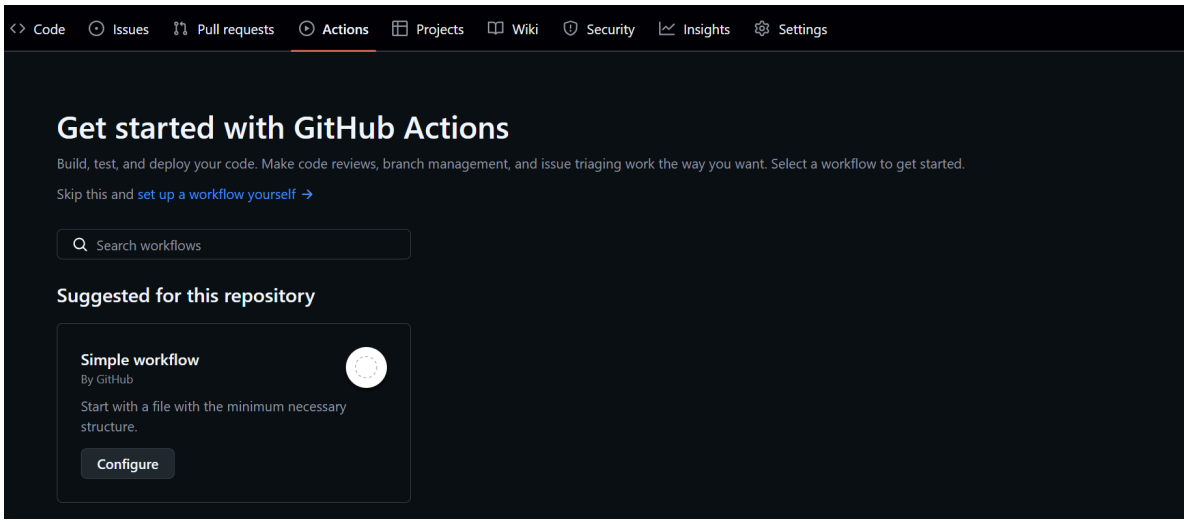
First GitHub Action

To create a GitHub Action, Let's first Create a repository.



Create Repository

Then click on the Actions tab> then configure.



Action

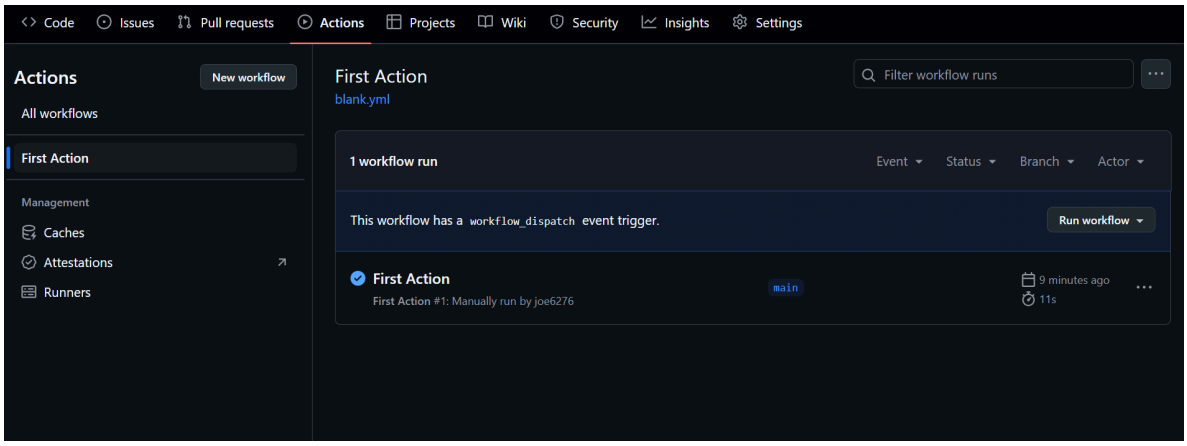
Now delete the steps on the .yml file and paste the following:

```yaml
name: First Action
# Above is the name of the workflow
on: workflow_dispatch
# This states the event that will trigger the workflow
# in this case "workflow_dispatch" means it will be manually triggered.
jobs:
  ## Now list the job to be executed
  first-job:
    # Give a name (depend on you)
    runs-on: ubuntu-latest
    # This is the virtual machine that will run the workflow
    steps:
    # Now the steps that will be taken
      - name: First hello World
        run: echo " Hello World "
        # Here we give a name, to uniquely identify the step
        # Then we use run to run a shell Command
      - name: Say GoodBye
        run: echo " GoodBye!"
        # Another step
```

Now commit the changes.

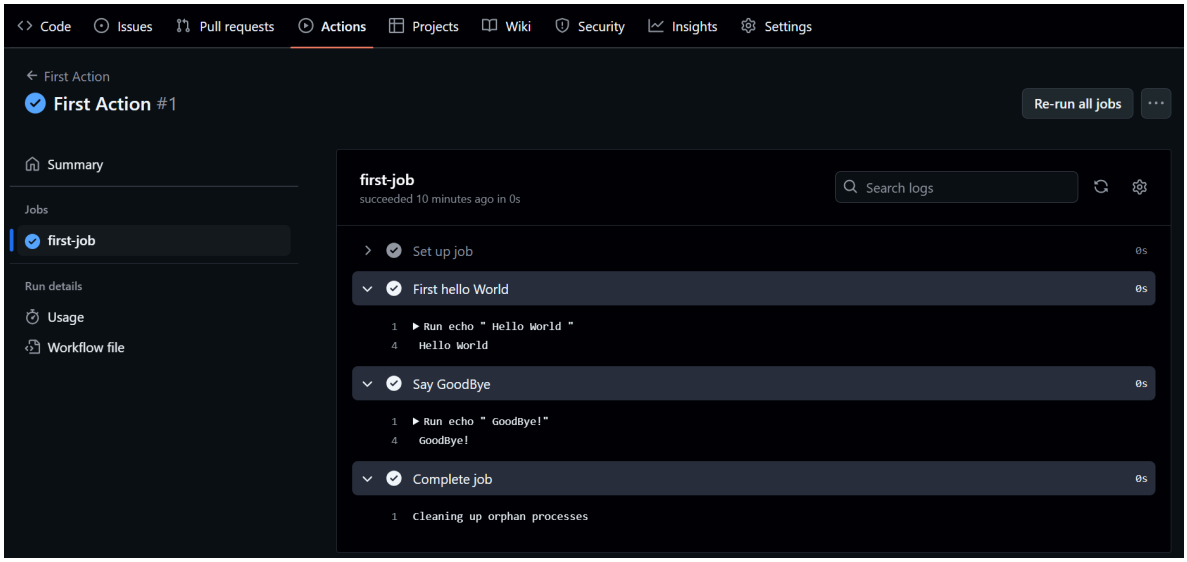
Run the workflow

Under actions > All Workflows You will find the 'First Action' work flow:



First Workflow

Click on run workflow.



Run Workflow

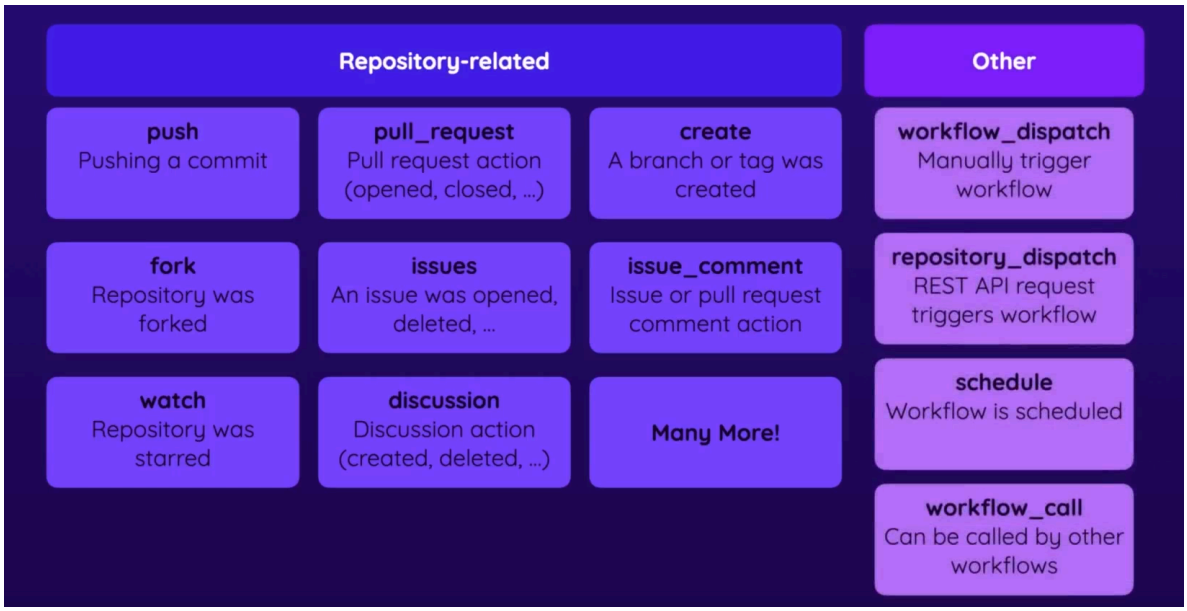
GitHub Action Example Two

Find the project Here: https://github.com/Teach2Give-Dotnet-Training/GA_second_Action This is a simple React App with some test files, the goal will be to write an action that can run the test files.

```yaml
name: Example two
on: push
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - name: Get the Code from the repository
        uses: actions/checkout@v4

      - name: Install NodeJS
        uses: actions/setup-node@v4
        with:
          node-version: '20'

      - name: Install Dependencies
        run: npm ci

      - name: Run Tests
        run: npm run test
```

We will give it a name. Then run the program whenever we push to the repository.

More on Events



Events

There are so many events that can trigger a workflow. We have repository related events and other events that don't depend on repositories. Events also have variations. To get to know events and variations more you can check at this link (https://docs.github.com/en/actions/writing-workflows/choosing-when-your-workflow-runs/events-that-trigger-workflows)

Action

- A (custom) application that performs a (typically complex) frequently repeated task
- You can build your own actions, but you can also use official or community actions.

An alternative to an action is Command

Command("run")

- A (typically simple) shell command that you define.

```
steps:
  - name: Get the Code from the repository
    uses: actions/checkout@v4
```

The ubuntu-latest runner by default does not have the code, the above step will make sure that the code is downloaded from the repository to the virtual machine.

```
- name: Install NodeJS
  uses: actions/setup-node@v4
  with:
     node-version: '20'
```

The above step is optional this is because the runner we are using has Node JS installed already. Here is a List of software installed in it: Link (https://github.com/actions/runner-images/blob/main/images/ubuntu/Ubuntu2204-Readme.md)

In case you want to use a different version of Node JS you can now use this action and give a different variation.

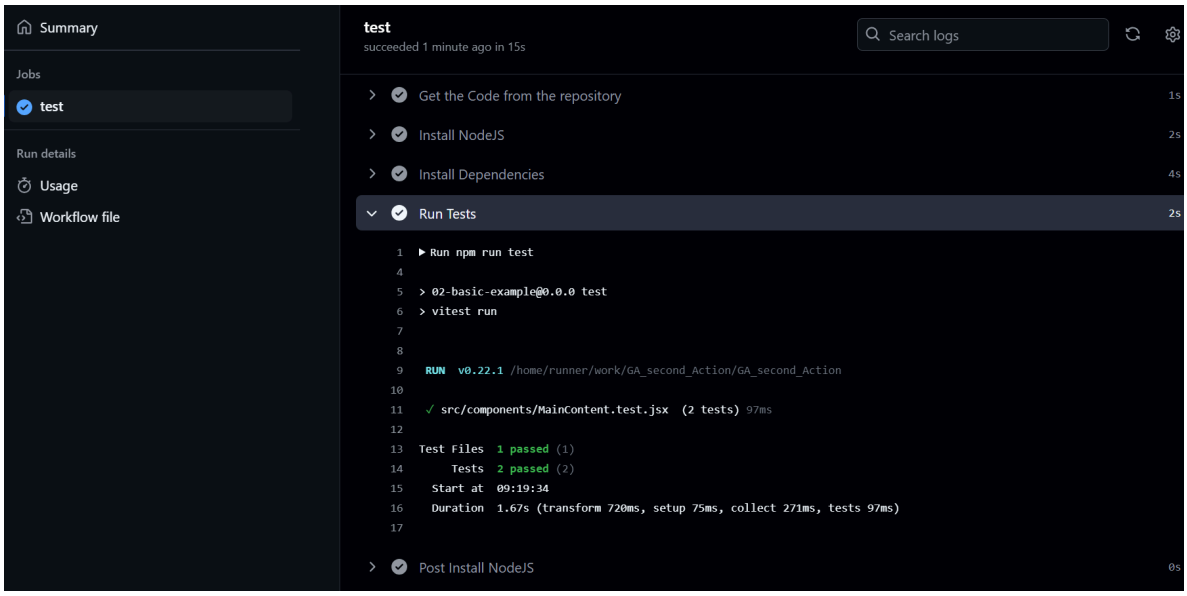
```
- name: Install Dependencies
  run: npm ci
```

This step will make sure that the runner installs the right dependencies needed to run the project.

```
- name: Run Tests
  run: npm run test
```

Now run the command to run the tests. This is also defined in package.json file.

Now push the changes and check your action (Its triggered by a push event)

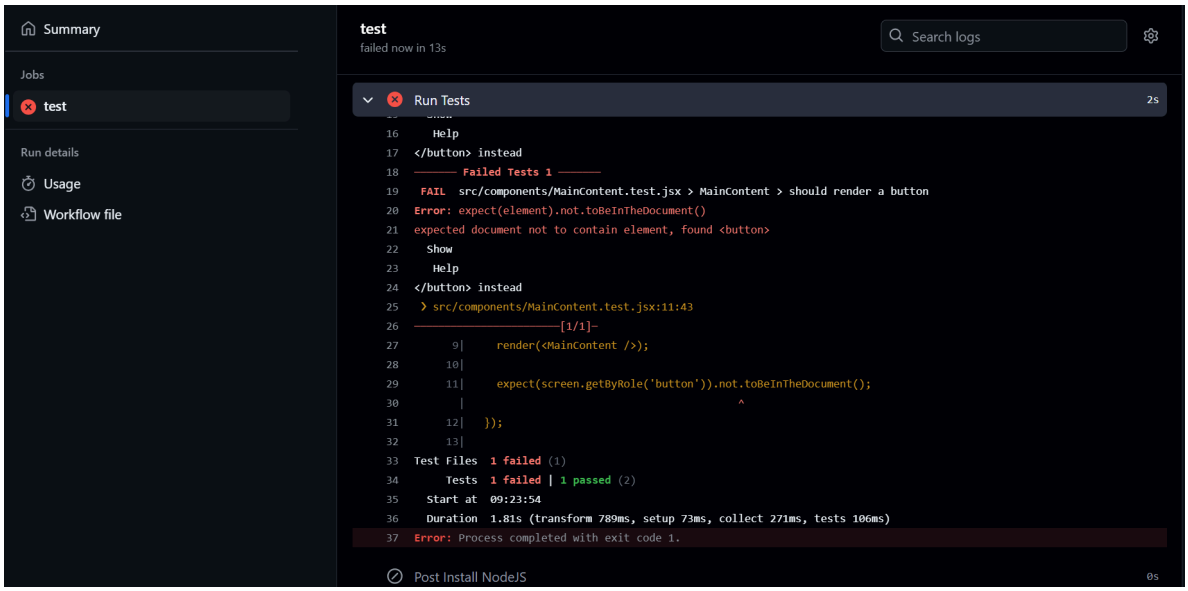


Success

Now let's cause a Bug im the MainContent.test.jsx by negating the test.

```
describe('MainContent', () => {
  it('should render a button', () => {
    render(<MainContent />);

    expect(screen.getByRole('button')).not.toBeInTheDocument();
  });
```

Now push the changes and check the Workflow. Now the workflow Fails:



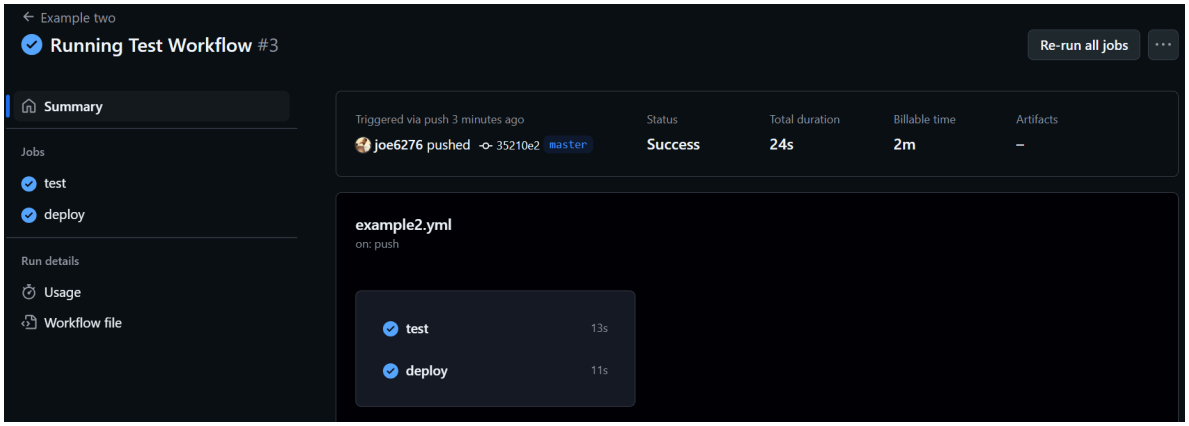
Failure

Now let's correct the above and add another step then push. The below step will mimic deploying an application

```yaml
name: Example two
on: push
jobs:
    test:
        runs-on: ubuntu-latest
        steps:
            - name: Get the Code from the repository
              uses: actions/checkout@v4

            - name: Install NodeJS
              uses: actions/setup-node@v4
              with:
                node-version: '20'

            - name: Install Dependencies
              run: npm ci

            - name: Run Tests
              run: npm run test
    deploy:
```

```
    runs-on: ubuntu-latest
    steps:
      - name: Get the Code from the repository
        uses: actions/checkout@v4

      - name: Install NodeJS
        uses: actions/setup-node@v4
        with:
          node-version: '20'

      - name: Install Dependencies
        run: npm ci

      - name: Build Projects
        run: npm run build

      - name: Deploy Project
        run: echo " Deploying Project..."
```

Now both steps are successful, but they run in parallel not sequential.

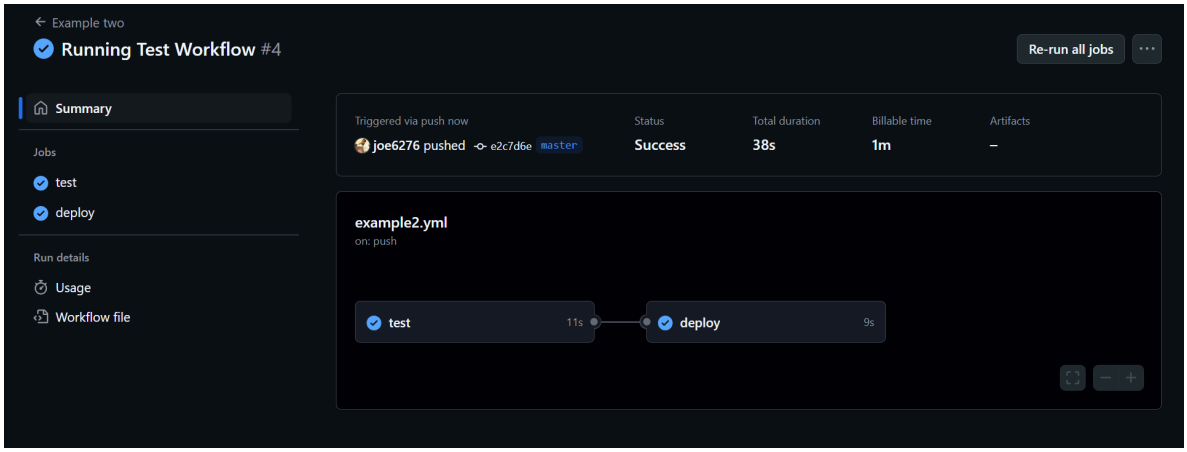


Steps in Parallel

needs

To make this sequential, in the deploy code add:

```yaml
deploy:
   needs: test
```



sequential

Multi-Events

To trigger workflow based on multiple events

```yaml
on: [push, workflow_dispatch]
```

GitHub Events

Event Activity Types and Filters

some events have Activity Types, others have filters.

Activity Types

More detailed control over when a workflow will be triggered. E.g., pull_request Event Activities include:

- opened
- closed
- edited

```yaml
name: Events Demo 1
on:
  pull_request:
    types: [opened, closed]
  workflow_dispatch:

jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Output event data
        run: echo "${{ toJSON(github.event) }}"
      - name: Get code
        uses: actions/checkout@v3
      - name: Install dependencies
        run: npm ci
      - name: Test code
        run: npm run test
      - name: Build code
        run: npm run build
```

```
      - name: Deploy project
        run: echo "Deploying..."
```

```
on:
  pull_request:
    types: [opened, closed]
  workflow_dispatch:
```

This part will make sure that when a pull request is opened or closed, the workflow will be triggered. The workflow can also be triggered manually using the workflow_dispatch(note that you have to add the semicolon at the end)

Most events with variation have defaults in case there is no type specified. E.g., for pull-request, the event's activity type is opened, synchronize, or reopened.

Filters

More detailed control over when a workflow will be triggered. E.g., push Event filter based on target branch.

```
push:
  branches:
    - master
  paths-ignore:
    - ".github/workflows/*"
```

The above actions will filter any push that targets the “master” branch. Any change made to any .yml file located in the .github>workflows directory.

Cancelling and Skipping Workflow

Cancelling

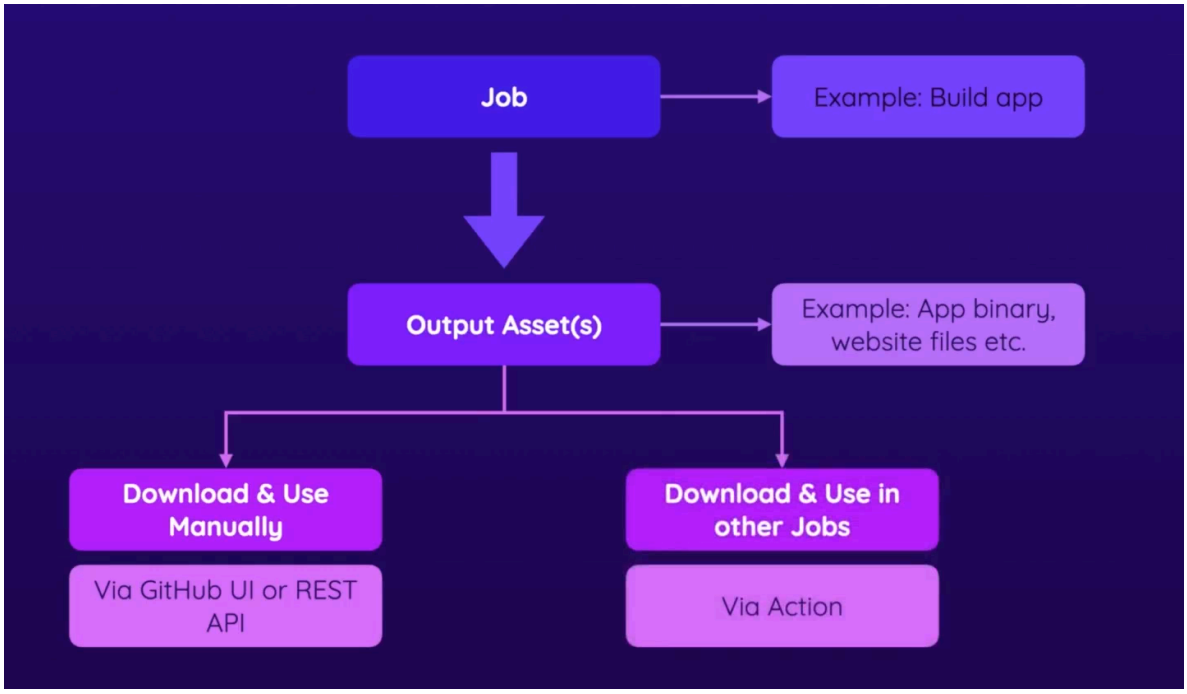
- Workflows get cancelled automatically when jobs fail
- You can manually cancel workflows

Skipping a Workflow

You can Skip via [Skip ci] etc. in the commit message. If the command is part of your commit message, the Workflow won't be executed.

Job Artifacts

Job artifacts are the assets/ output generated by a job.



Build Job

Starting Workflow:

```yaml
name: Deploy website
on:
  push:
    branches:
      - master

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - name: Get code
        uses: actions/checkout@v3
      - name: Install dependencies
        run: npm ci
```

```
      - name: Lint code
        run: npm run lint
      - name: Test code
        run: npm run test

  build:
    needs: test
    runs-on: ubuntu-latest
    steps:
      - name: Get code
        uses: actions/checkout@v3
      - name: Install dependencies
        run: npm ci
      - name: Build website
        run: npm run build

  deploy:
    needs: build
    runs-on: ubuntu-latest
    steps:
      - name: Deploy
        run: echo "Deploying..."
```

The above has three jobs a test, build and deploy. But let's focus on the build job because when the *npm run build* command is executed, it produces a dist file which id the file that is uploaded to a web server.

The problem is when we execute the script as it is right now, when the runner is done, it will shut down, and we will lose the files generated by the job.

So let's modify this, on the Build steps let's modify the code.

```
build:
  needs: test
  runs-on: ubuntu-latest
  steps:
    - name: Get code
      uses: actions/checkout@v3
```

```
    - name: Install dependencies
      run: npm ci
    - name: Build website
      run: npm run build
    - name: Upload Artifacts
      uses: actions/upload-artifact@v4
      with:
        name: dist-files
        path: dist
```

Explanation

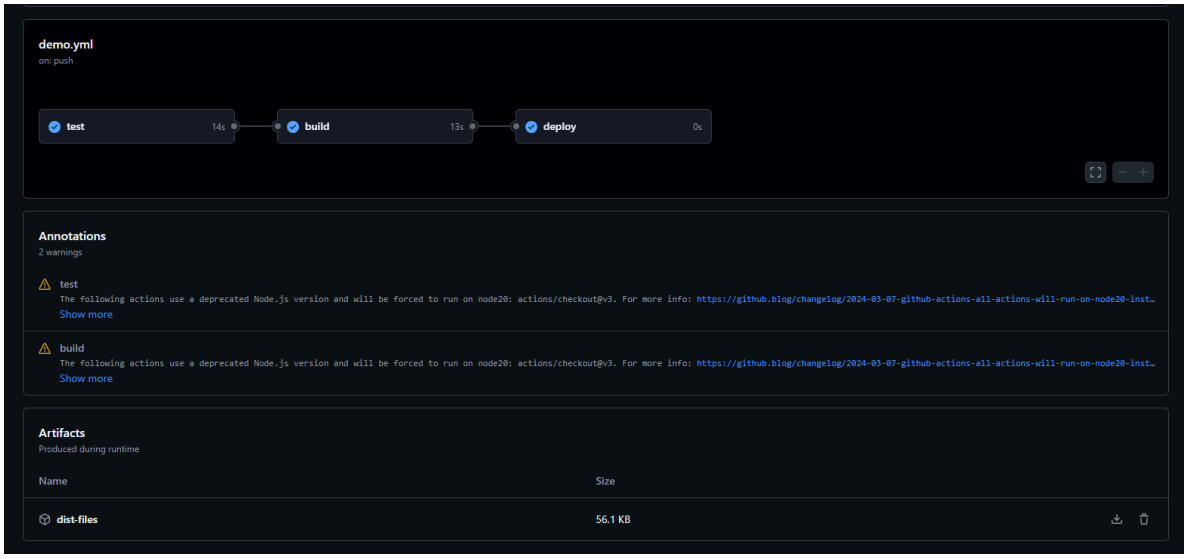
```
- name: Upload Artifacts
  uses: actions/upload-artifact@v4
```

We will give the step a name and specify that we are going to use an existing action which helps us upload the artifacts.

```
with:
    name: dist-files
    path: dist
```

we will give it a name, in this case its *dist-files*, and we will specify the path the files will be uploaded from *dist*.

and now in your workflow, you should be able to see and download the files:



Artifacts

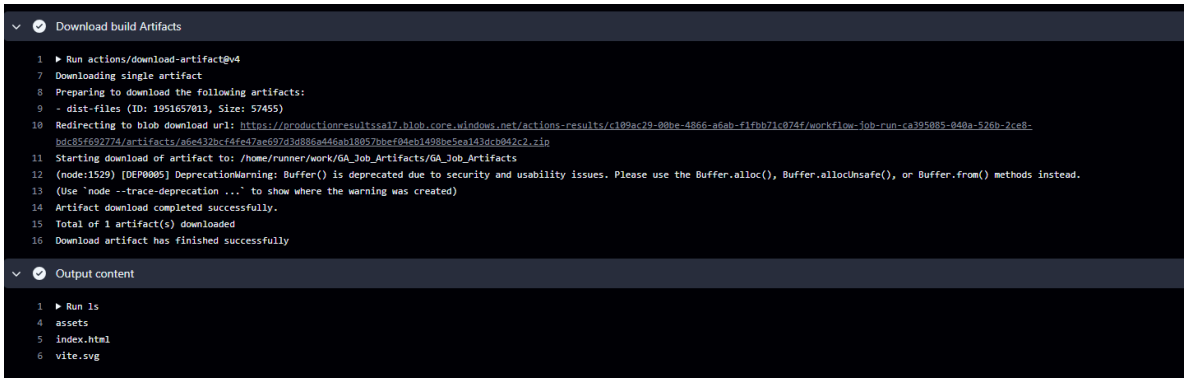
But we need to use the files in the deployment job. The files produced by the build process will not persist to the deployment job because they are on different runners even though they use the same runner definition. To get the artifacts, we need to download them in the deployment step. Let's modify the deployment job

```
deploy:
  needs: build
  runs-on: ubuntu-latest
  steps:
    - name: Download build Artifacts
      uses: actions/download-artifact@v4
      with:
        name: dist-files
    - name: Output content
      run: ls
    - name: Deploy
      run: echo "Deploying..."
```

We will add the download artifact action and specify the names of the files to be downloaded, in this case its *dist-files* which is the same name we used in the build step.

The action here will download and unzip the files: And we can now use the 'ls' step to list the contents.

Now we can see the output right here:



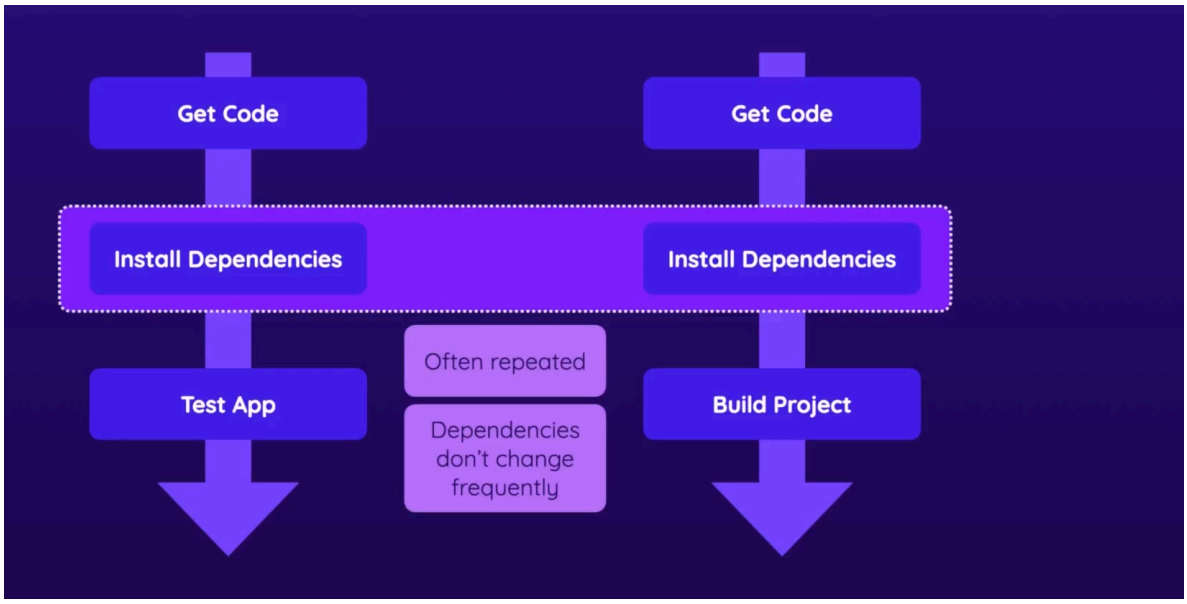
Download content

Dependencies Caching

Step Time

If we take a look at the build step, we will notice that the step that takes more time is the dependency installation step. Also, we will notice that we have repeated the step in both test and build job.

We can save some time if we cache the dependencies and reuse then in the build step.



Cache

GitHub Action has an action that will help us cache the dependencies

```
- name: Cache Dependencies
  uses: actions/cache@v4
  with:
    path: ~/.npm
    key: deps-node-modules- ${{hashFiles('**/package-lock.json')}}
```

This will now cache the dependencies and will only run *npm ci* if the package.lock.json file has changed so that means we will be able to use cache for every step provided the package-lock.json file has not changed.

Now the workflow file will look like:

```
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - name: Get code
        uses: actions/checkout@v4
      - name: Cache Dependencies
        uses: actions/cache@v4
        with:
```

```
          path: ~/.npm
          key: deps-node-modules- ${{hashFiles('**/package-lock.json')}}
      - name: Install dependencies
        run: npm ci
      - name: Lint code
        run: npm run lint
      - name: Test code
        run: npm run test

  build:
    needs: test
    runs-on: ubuntu-latest
    steps:
      - name: Get code
        uses: actions/checkout@v4
      - name: Cache Dependencies
        uses: actions/cache@v4
        with:
          path: ~/.npm
          key: deps-node-modules- ${{hashFiles('**/package-lock.json')}}
      - name: Install dependencies
        run: npm ci
      - name: Build website
        run: npm run build
      - name: Upload Artifacts
        uses: actions/upload-artifact@v4
        with:
          name: dist-files
          path: dist
```

Now you will notice that the Cache is used for the second Job (build). And cache will be used, hence it will be a bit faster.


Cache Dependencies
1s
▶ Run actions/cache@v4
Cache Size: ~19 MB (19568772 B)
/usr/bin/tar -xf /home/runner/work/_temp/dfd364fa-ed51-4b9d-91d6-a2fdf7f1172a/cache.tzst -P -C /home/runner/work/GA_Job_Artifacts/GA_Job_Artifacts --use-compress-program unzstd
Cache restored successfully
Cache restored from key: deps-node-modules- 5a782fe93589e2cbe03aac182697ecd9156b0019da2b2cd4ec8f40591080c823
Install dependencies
4s
▶ Run npm ci
added 375 packages, and audited 376 packages in 4s
77 packages are looking for funding
run `npm fund` for details
12 vulnerabilities (5 moderate, 6 high, 1 critical)
To address all issues, run:
npm audit fix
Run `npm audit` for details.
Build website
3s

Caching

Environment Variables & Secrets

Environment Variables are certain variables that are dynamic e.g. password to connect server to your database might depend if you are on development environment or Production environment. These two environments might have different databases, hence different passwords.

Below are some JS variables set from environment variables

```
const clusterAddress = process.env.MONGODB_CLUSTER_ADDRESS;
const dbUser = process.env.MONGODB_USERNAME;
const dbPassword = process.env.MONGODB_PASSWORD;
const dbName = process.env.MONGODB_DB_NAME;
```

You can define environment variables in different levels:

- Workflow level

```
name: Environment Variables
on:
  push:
    branches:
      - master
env:
  MONGO_DB_NAME: githubExample
```

- Job Level (Available for that step only)

```
jobs:
  test:
    env:
      MONGODB_USERNAME: test
      MONGODB_PASSWORD: Test@123
```

```
      MONGODB_CLUSTER_ADDRESS: cluster0.15...
      PORT: 80
```

- Step Level (Available for that step only)

To output/ use the environment variables, there are two ways:

```
- name: Run server
  run: npm start & npx wait-on http://127.0.0.1:$PORT
```

or

```
- name: Output Environment Variables
  run: |
      echo "MONGO DB UserName : ${{env.MONGODB_USERNAME}}"
      echo " At Port $PORT"
```

Output:

ENV output

But the Environment value is still part of the workflow file, which means anyone who can access the workflow file can access the environment variables. So we need ***Secrets***

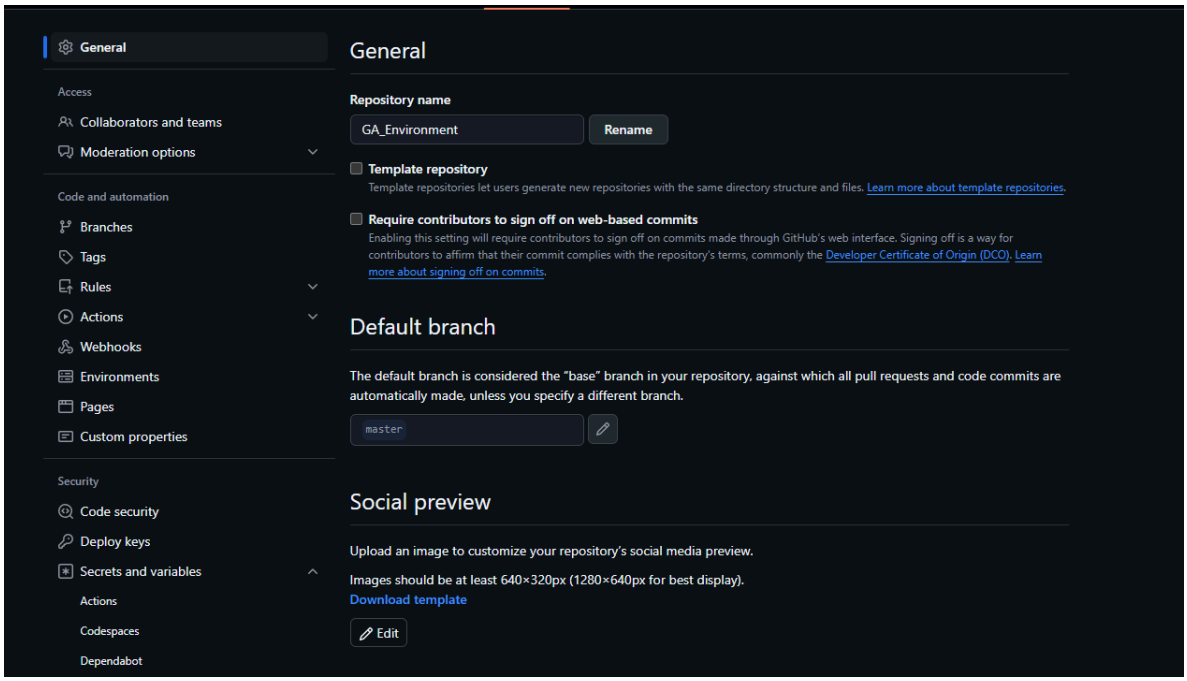
Secrets

Secrets can be stored on :

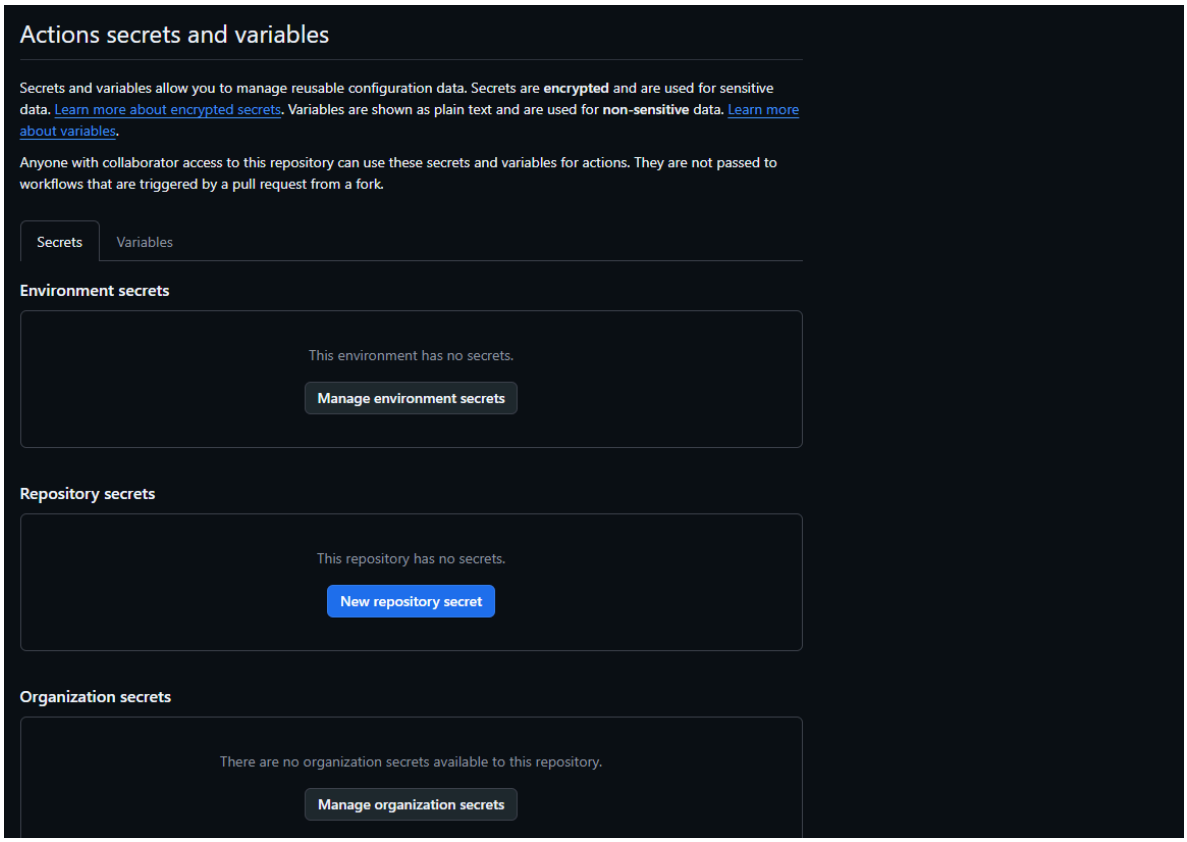
- Organizational Level
- Repository Level

Repository Level

Under Settings find Secrets & Variables > Actions

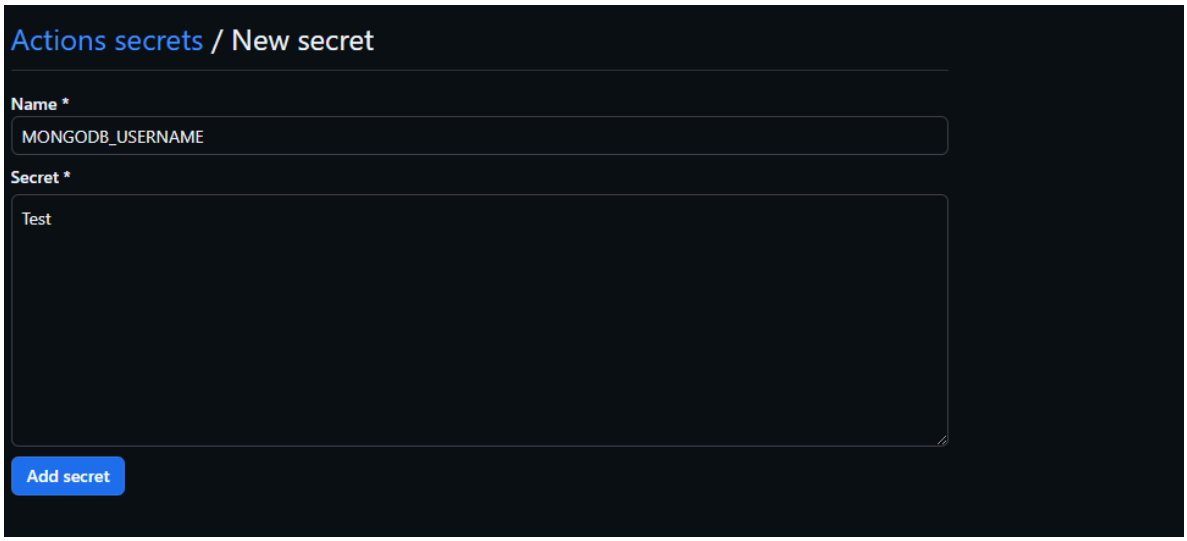


Setting



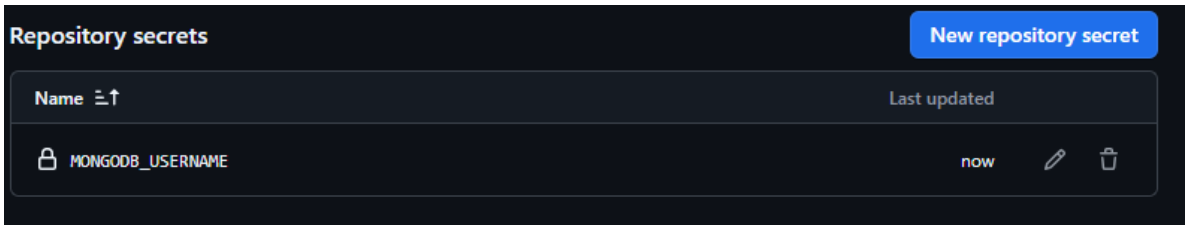
Action

Now add new repository secret:



Secret

Once a secret is stored, it can never be viewed only deleted or updated.



Secret 1g

Now, how can we access the secrets:

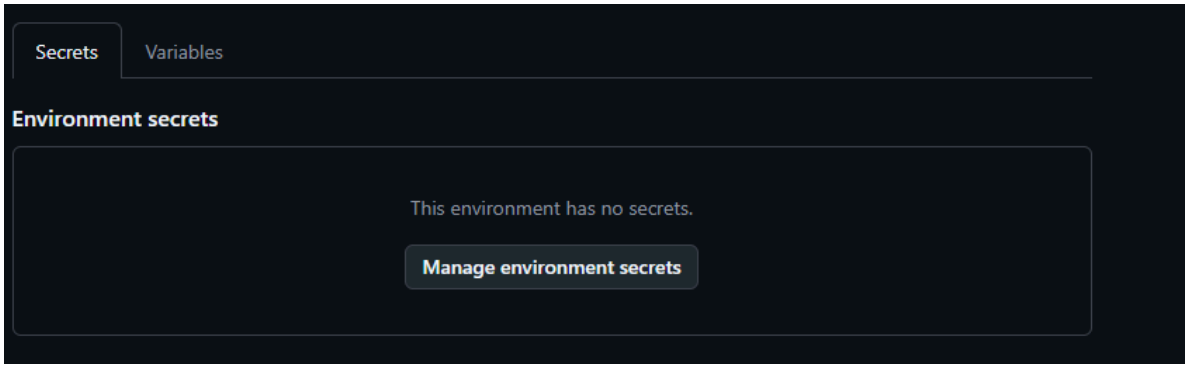
```
env:
  MONGODB_USERNAME: ${{secrets.MONGODB_USERNAME}}
```

And Now even when you try to output, GitHub will hide the secret.

Secret Output

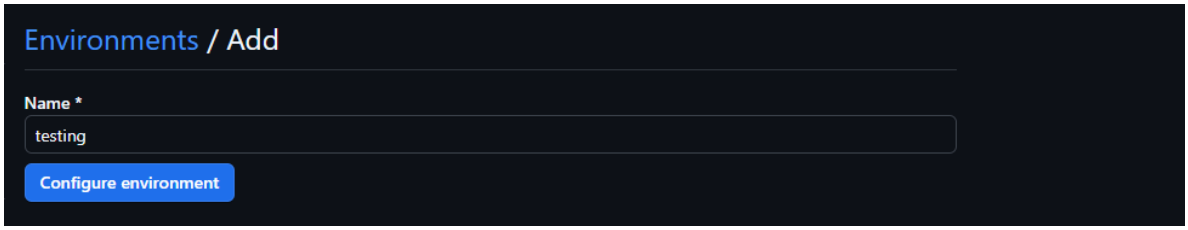
The above secrets are available for the whole workflow file, hence every job. This might work well, but sometimes you want to use different databases for different Jobs e.g., you want to use a database Named "Gh-Dev" for development and "GH-Testing" for testing. This is where environments can play a role in .

Now from your settings> Secrets & variables > Actions



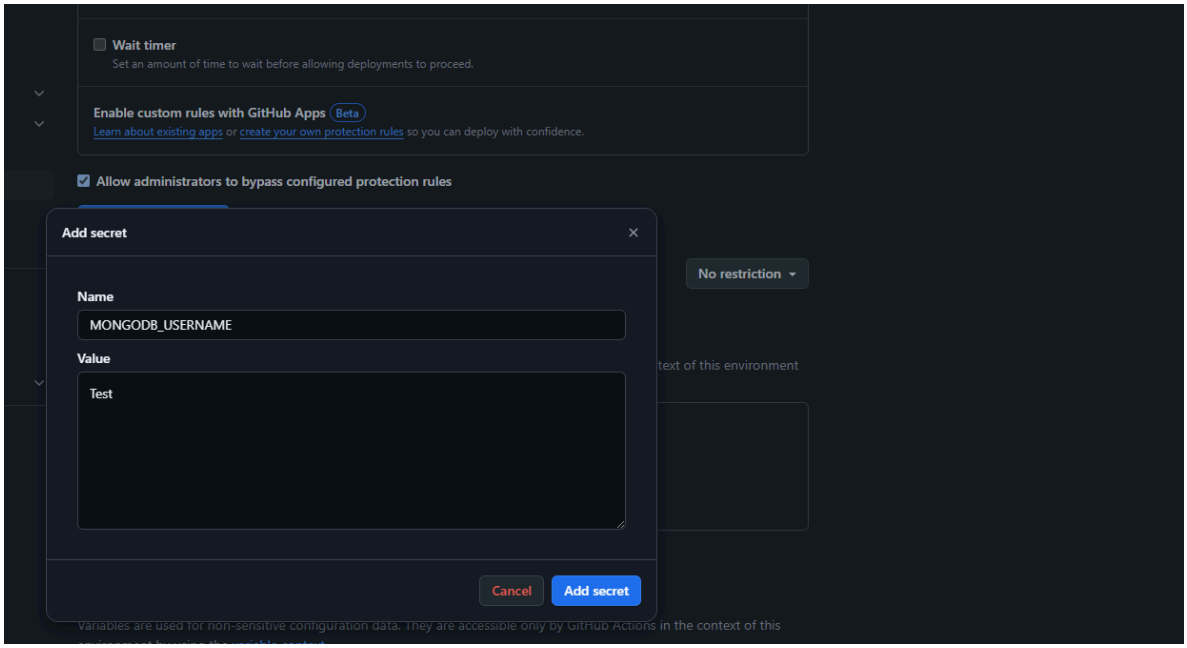
Enviroment

Create a new Environment:



Create Environment

Then Add Secrets:



Add Secrets

Now in the Workflow File

```
jobs:
  test:
    environment: testing
    env:
      MONGODB_USERNAME: ${{secrets.MONGODB_USERNAME}}
```

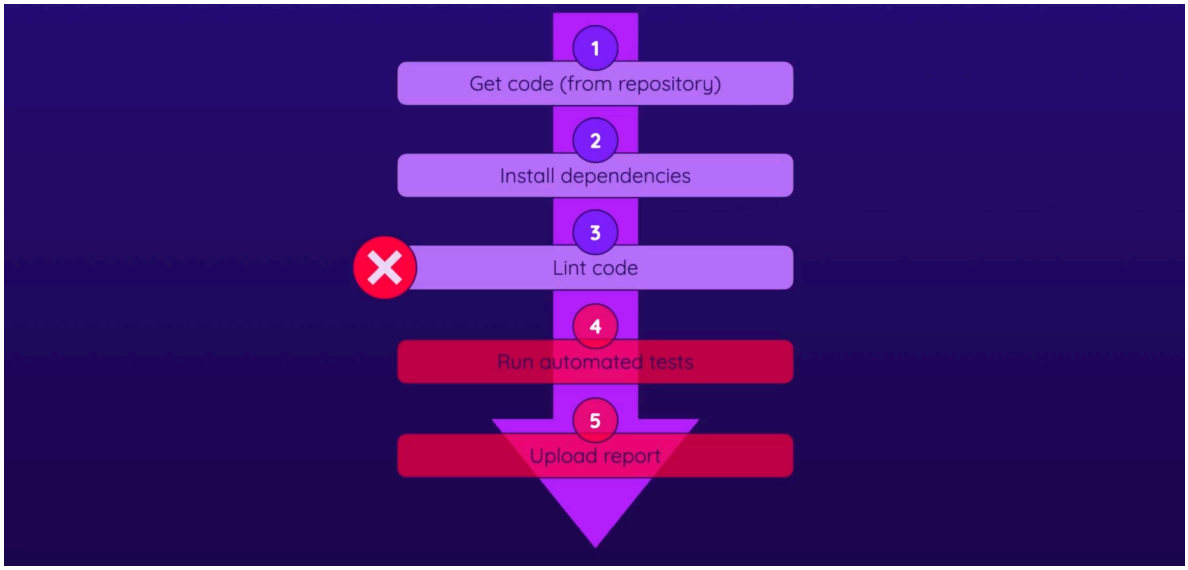
This Job will use the variable defined in the 'testing' environment, and you can now configure other environment and give different values for the environment variables.

Summary

Summary

Control Execution Flow

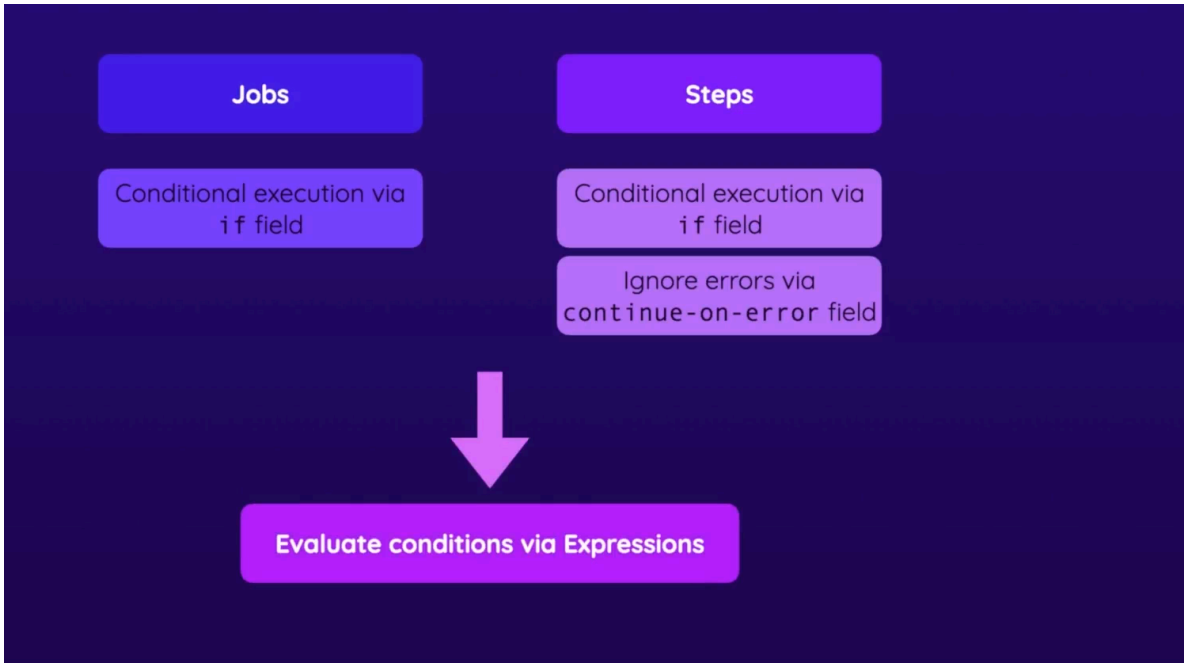
At this point, we have jobs executed in a series, and if one fails, the rest are not executed. That's the default behavior.



Execution

But sometimes you want to keep executing even if one previous step fails, or even trigger another step if it fails or even control your execution flow.

Conditional Jobs & steps



Control Flow

```
- name: Test code
  run: npm run test
  - name: Upload test report
    uses: actions/upload-artifact@v3
     with:
       name: test-report
       path: test.json
```

For the above case, it would make sense if we upload a test report if the tests fail. So we should run the Upload test report conditionally and only if when the test fail.

```
- name: Test code
   id: steps-test
   run: npm run test
 - name: Upload test report
   if: failure() && steps.steps-test.outcome =='failure'
   uses: actions/upload-artifact@v3
   with:
```

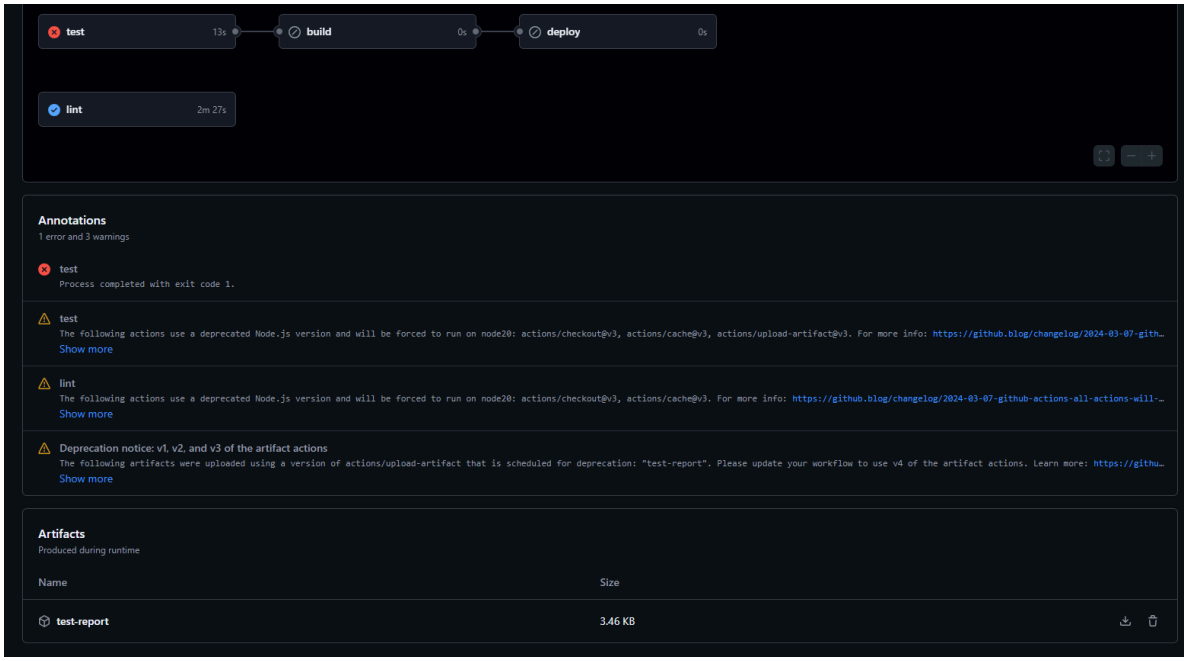
```
      name: test-report
      path: test.json
```

We will first start by assigning the step an ID so that we can know the outcome of the test step. On the upload test report step We execute the failure() which should return true if a previous step fails, and also the outcome of the previous step is checked. If both are true, then we allow the upload step to execute, and it will upload the necessary files.

Special Condition Functions

Special Condition Function

Now when the tests fail, we get a report:



Report

Condition on Jobs

```yaml
report:
  needs: [lint,deploy]
  runs-on: ubuntu-latest
  if: failure()
  steps:
    - name: Output Text
      run: echo "Either Lint or test failed"
```

The above job will only run when the lint or deploy a j ob fails

Output:

Conditional Jobs

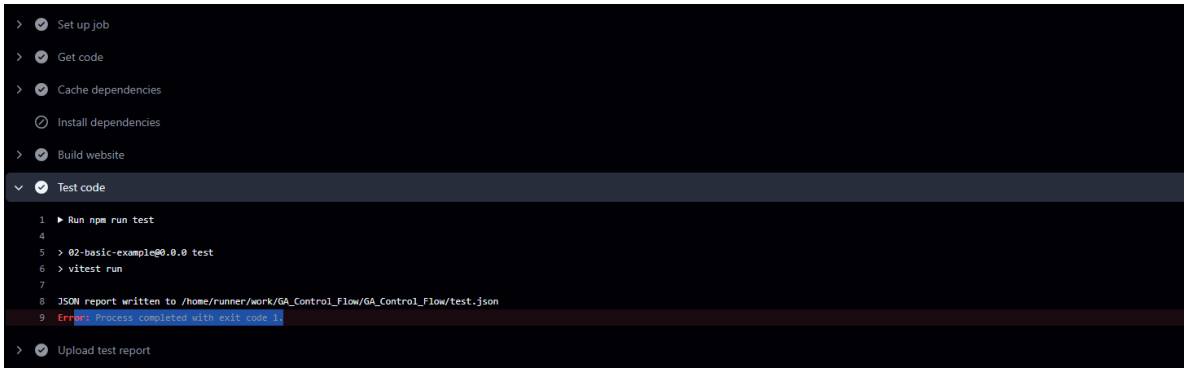
Continue

Instead of using an if statement, we can use *continue-on-error* which is going to make sure the workflow still executes even if there is an error.

We can make our test a failing test then modify the workflow as follows:

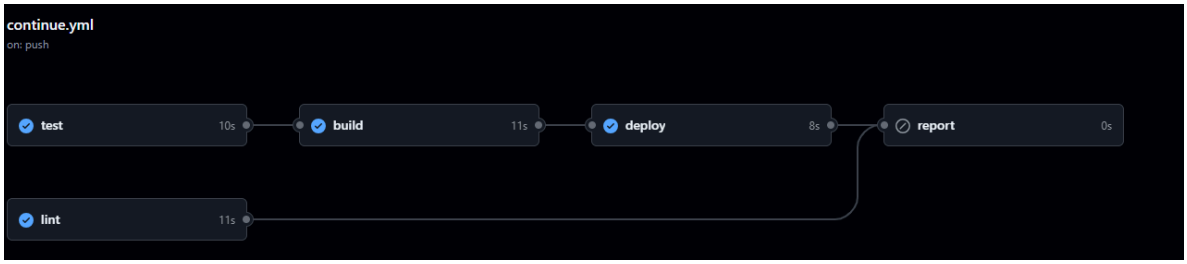
```yaml
- name: Test code
  id: steps-test
  continue-on-error: true
  run: npm run test
- name: Upload test report
  uses: actions/upload-artifact@v3
  with:
    name: test-report
    path: test.json
```

Output is: Even through the Test Step Fails:



Test Fails

The Workflow succeeds:



Workflow Success

Matrix

Sometimes you have a workflow :

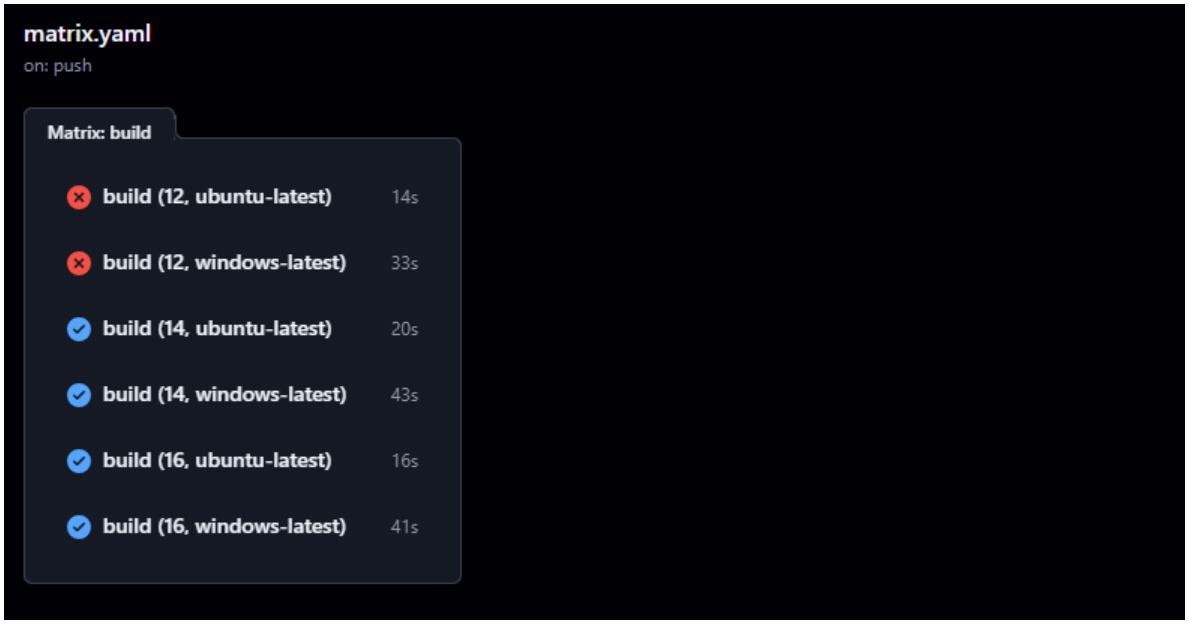
```yaml
name: Matrix -Demo
on: push
jobs:
    build:
        runs-on: ubuntu-latest
        steps:
            - name: Get Code
              uses: actions/checkout@v4

            - name: Install NodeJS
              uses: actions/setup-node@v4
              with:
                node-version: 14

            - name: Install Dependencies
              run: npm ci

            - name: Build Project
              run: npm run build
```

And you want to see how the workflow would behave in different environments.E.g. How would it work on node version 18, 20 or even on different runners. You need to use a matrix in that case:

```yaml
name: Matrix -Demo
on: push
jobs:
    build:
        continue-on-error: true
        strategy:
            matrix:
                node-version: [12,14,16]
                operating-system: [ubuntu-latest, windows-latest]
```

```
    runs-on: ${{matrix.operating-system}}
    steps:
      - name: Get Code
        uses: actions/checkout@v4

      - name: Install NodeJS
        uses: actions/setup-node@v4
        with:
          node-version: ${{matrix.node-version}}

      - name: Install Dependencies
        run: npm ci

      - name: Build Project
        run: npm run build
```

Now in this case, we will run the job with different operating systems and also in different node versions.



matrix

But assuming you need Node-version 18 and ubuntu-latest operating system and not Node-version 18 and Windows-latest operating system, you can use the include

keyword. You can also exclude a certain combination.

```
strategy:
          matrix:
              node-version: [12,14,16]
              operating-system: [ubuntu-latest, windows-latest]
              include:
                  - node-version: 18
                    operating-system: ubuntu-latest
              exclude:
                  - node-version: 12
                    operating-system: windows-latest
runs-on: ${{matrix.operating-system}}
steps:
```

Reusable Workflows

Sometimes you have workflows that others workflows can reuse, for example, lets have the deployment job in a separate file.

in a separate file write:

```
name: Reusable Workflow
on: workflow_call
jobs:
    deploy:
        runs-on: ubuntu-latest
        steps:
            - name: Output information
              run: echo " Deploying & Uploading Site..."
```

The only thing that is different is this will be called on *workflow_call*, so that is an event that means another workflow must call it.

on the target workflow write:

```
Deploy:
      needs: build
```

```
    uses: ./.github/workflows/reusable.yaml
```

you must give the relative path to the workflow we want to reuse.

Now separating the two files brings a problem. The problem is that the deploy job needs the artifacts produced by the build job. So we need to pass the file from the Main workflow to the reusable deploy job.

on the deploy job, we are going to accept inputs:

```
name: Reusable Workflow
on:
    workflow_call:
        inputs:
            artifacts:
                description: The name of the deployable artifact files
                required: false
                default: dist
                type: string
jobs:
    deploy:
        runs-on: ubuntu-latest
        steps:
            - name: Get Code
              uses: actions/download-artifact@v4
              with:
                name: ${{inputs.artifacts}}

            - name: List Files
              run: ls

            - name: Output information
              run: echo " Deploying & Uploading Site..."
```

Here we will define an input that has a description, a required boolean to show if the value must be passed, a default value for any default you would like to pass in case the required is false and the type of the value you will pass.

On the other workflow we need to pass the artifacts:

```
Deploy:
      needs: build
      uses: ./.github/workflows/reusable.yaml
      with:
        artifacts: dist-files
```

besides inputs you can also pass secrets. and also outputs.

Outputs

Below is the reusable workflow:

```
name: Reusable Workflow
on:
    workflow_call:
        inputs:
            artifacts:
                description: The name of the deployable artifact files
                required: false
                default: dist
                type: string

        outputs:
            result:
                description: The result of the deployment operation
                value: ${{jobs.deploy.outputs.outcome}}

jobs:
    deploy:
        outputs:
            outcome: ${{steps.output-step.outputs.step-result}}
        runs-on: ubuntu-latest
        steps:
            - name: Get Code
              uses: actions/download-artifact@v4
              with:
                name: ${{inputs.artifacts}}
```

```
      - name: List Files
        run: ls

      - name: Output information
        run: echo " Deploying & Uploading Site..."
      - name: Set Result Output
        id: output-step
        run: echo "::set-output name=step-result::success"
```

```
- name: Set Result Output
  id: output-step
  run: echo "::set-output name=step-result::success"
```

- name: "Set Result Output."
- id: The ID is output-step, used to reference this step's output. run: Sets the output variable step-result to success using echo "::set-output name=step-result::success".
- outputs: Defines the output outcome for this job, which is taken from the result of the step with ID output-step.

```
outputs:
   outcome: ${{steps.output-step.outputs.step-result}}
```

- jobs: Defines a job named deploy.
- outputs: Defines the output outcome for this job, which is taken from the result of the step with ID output-step.

Custom Actions

We have been using public actions so far :

- actions/checkout@v3
- actions/cache@v3
- actions/upload-artifact@v3
- actions/download-artifact@v3

It's time to build custom actions now, but why?

Why Custom Action

- Simplify workflow steps Instead of writing multiple (possibly very complex) step definitions, you can build and use a single custom action. Multiple steps can be grouped into a single custom action, which can be reused.
- No existing (community) action Existing, public actions might not solve the specific problem you have in your workflow. Custom actions can contain any logic you need to solve your specific workflow problems.
- You can publish it in the marketplace and ccontribute to the community.

Different Types of custom Actions

- Javascript Actions
- Docker Actions
- Composite Actions

Javascript Actions

- Execute a JavaScript file

- Use JavaScript (Node.js) + any packages of your choice.
- Pretty Straightforward (If you know JavaScript)

Docker Actions

- Create a Dockerfile with your required configuration.
- Perform any tasks of your choice with any Language
- Lots of flexibility but required Docker Knowledge.

Composite Actions

- Combine multiple workflow steps in one single Action
- Combine run (commands) and use (Actions)
- Allows for reusing shared steps (without extra skills)

Here is out starting Workflow:

```yaml
name: Deployment
on:
  push:
    branches:
      - master

jobs:
  lint:
    runs-on: ubuntu-latest
    steps:
      - name: Get code
        uses: actions/checkout@v3
      - name: Cache dependencies
        id: cache
        uses: actions/cache@v3
        with:
          path: node_modules
```

```
          key: deps-node-modules-${{ hashFiles('**/package-lock.json')
}}
      - name: Install dependencies
        if: steps.cache.outputs.cache-hit != 'true'
        run: npm ci
      - name: Lint code
        run: npm run lint
  test:
    runs-on: ubuntu-latest
    steps:
      - name: Get code
        uses: actions/checkout@v3
      - name: Cache dependencies
        id: cache
        uses: actions/cache@v3
        with:
          path: node_modules
          key: deps-node-modules-${{ hashFiles('**/package-lock.json')
}}
      - name: Install dependencies
        if: steps.cache.outputs.cache-hit != 'true'
        run: npm ci
      - name: Test code
        id: run-tests
        run: npm run test
      - name: Upload test report
        if: failure() && steps.run-tests.outcome == 'failure'
        uses: actions/upload-artifact@v3
        with:
          name: test-report
          path: test.json
  build:
    needs: test
    runs-on: ubuntu-latest
    steps:
      - name: Get code
        uses: actions/checkout@v3
      - name: Cache dependencies
```

```
        id: cache
        uses: actions/cache@v3
        with:
          path: node_modules
          key: deps-node-modules-${{ hashFiles('**/package-lock.json')
}}
      - name: Install dependencies
        if: steps.cache.outputs.cache-hit != 'true'
        run: npm ci
      - name: Build website
        run: npm run build
      - name: Upload artifacts
        uses: actions/upload-artifact@v3
        with:
          name: dist-files
          path: dist
  deploy:
    needs: build
    runs-on: ubuntu-latest
    steps:
      - name: Get code
        uses: actions/checkout@v3
      - name: Get build artifacts
        uses: actions/download-artifact@v3
        with:
          name: dist-files
          path: ./dist
      - name: Output contents
        run: ls
      - name: Deploy site
        run: echo "Deploying..."
```

if you look at the workflow file, you will notice that there are repeated steps :

- Getting the code

- Caching dependencies

lets build a composite action

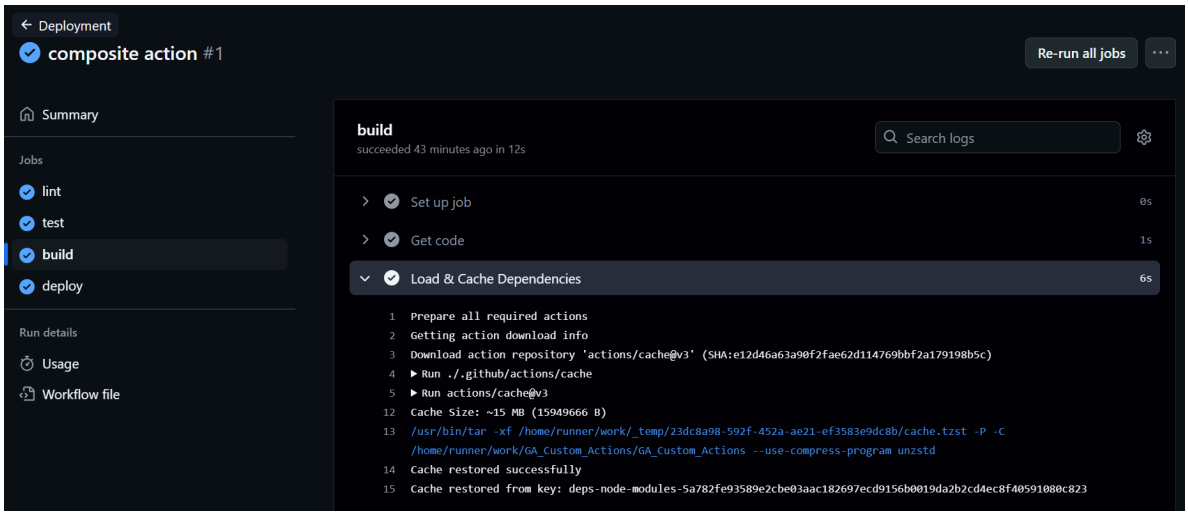
```yaml
name: 'Get  and Cache Dependencies'
description: ' Get the dependencies and Cache them'
runs:
  using: 'composite'
  steps:
    - name: Cache dependencies
      id: cache
      uses: actions/cache@v3
      with:
        path: node_modules
        key: deps-node-modules-${{ hashFiles('**/package-lock.json') }}

    - name: Install dependencies
      if: steps.cache.outputs.cache-hit != 'true'
      run: npm ci
      shell: bash
```

- Here, we will give it a name and a description first
- **runs**: Defines how the action will be executed.
- This particular action uses a composite type, meaning it contains multiple steps within.
- For any step that uses the run command, we must pass the *shell: bash*

to use it, we need to specify a pathe of where the action file is located:

```yaml
- name: Load & Cache Dependencies
  uses: ./.github/actions/cache
```

And now the workflow runs well, and it's streamlined.

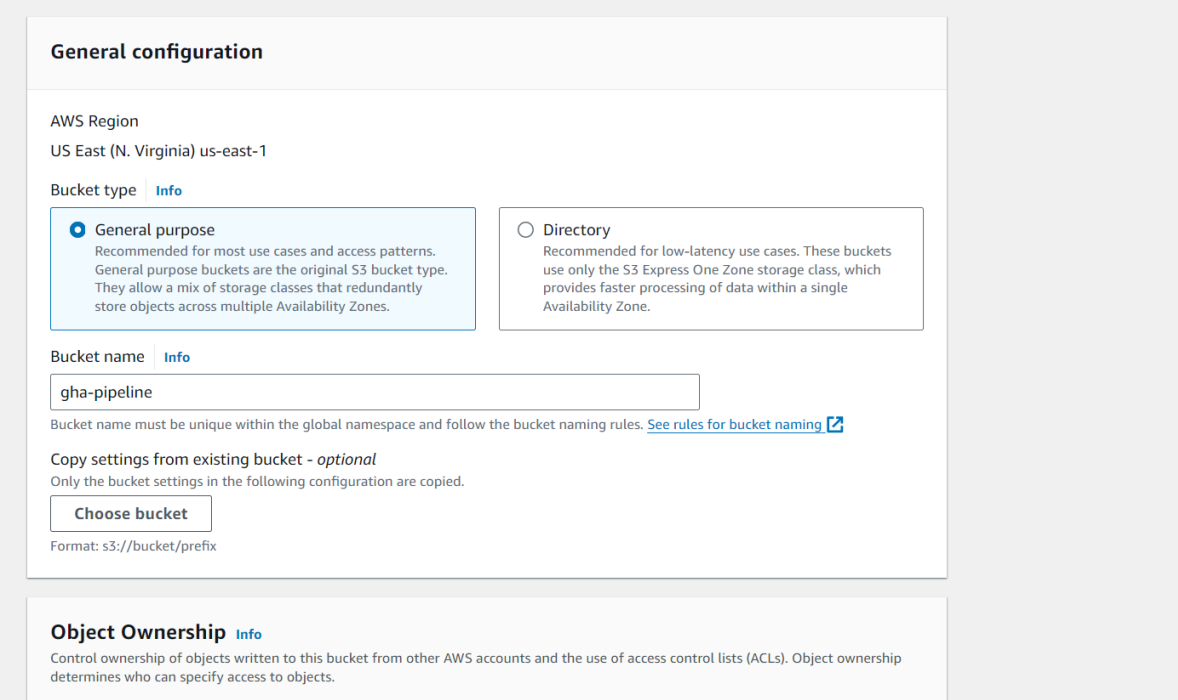


Workflow Output

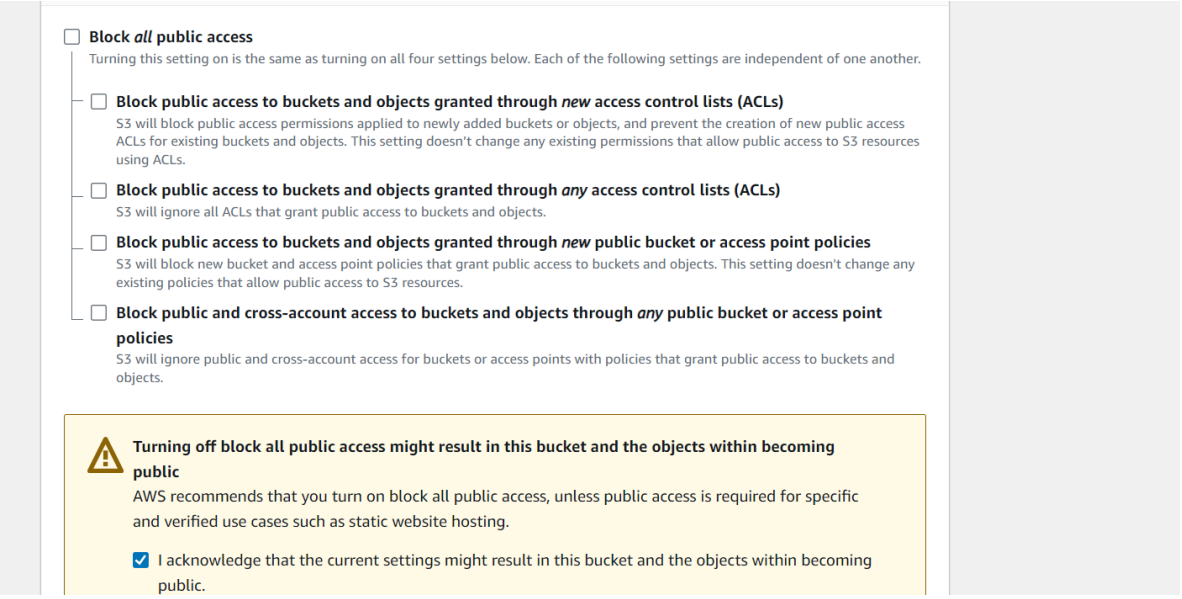
Composite actions can also have inputs and outputs too.

JavaScript Action

Let create an S3 bucket



S3 Creation



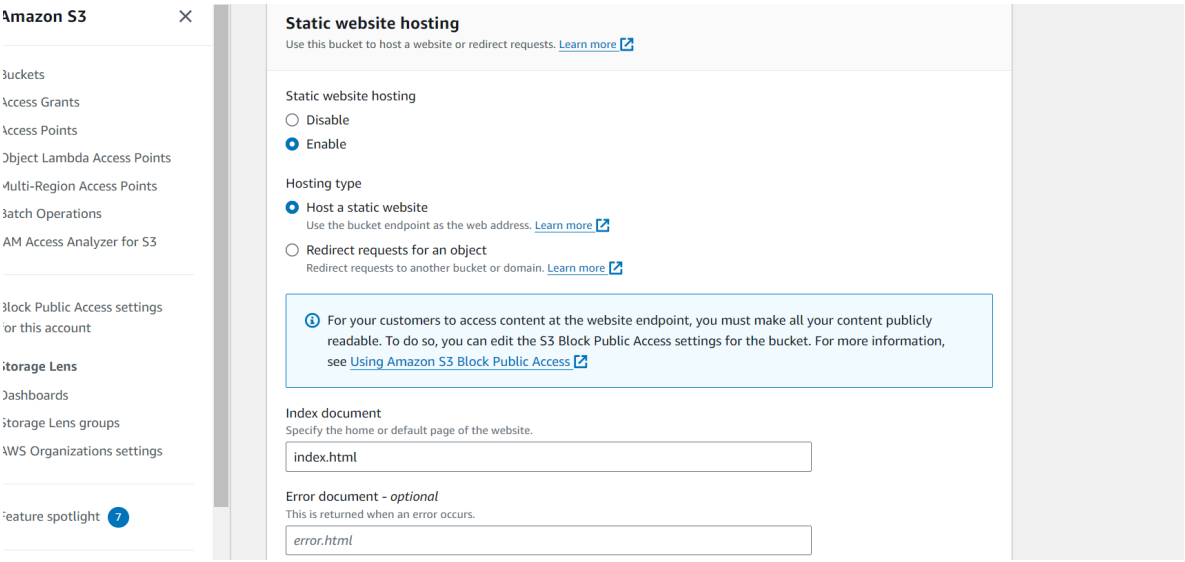
S3 Access

Leave the rest as default and create it.

Static website hosting

Go to the bucket> then properties > Static website hosting> Edit

Enable website hosting and also make sure the index.html is filled in the index document then save the changes.



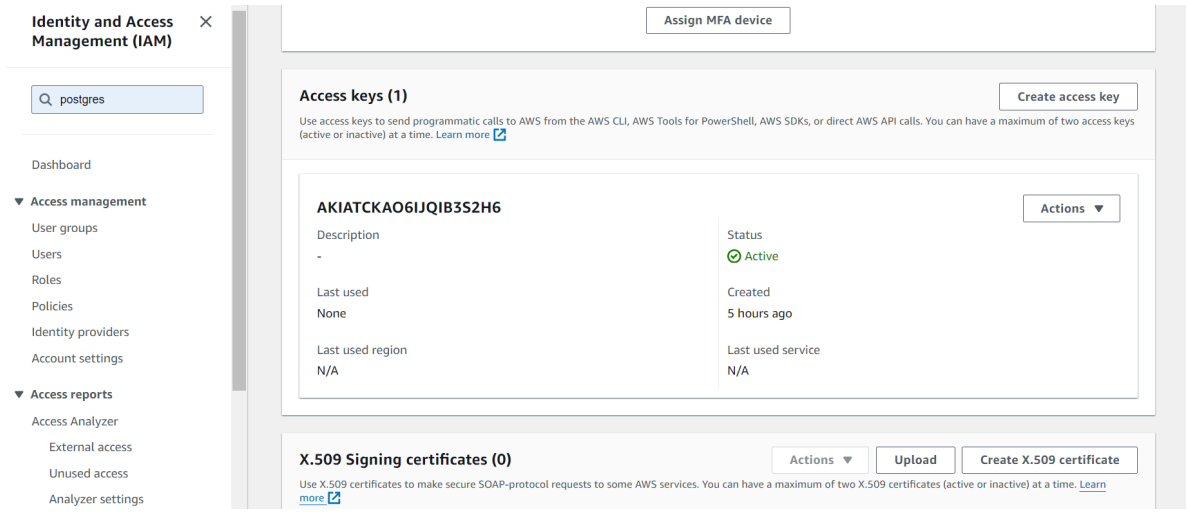
Static Website Hosting

Permission

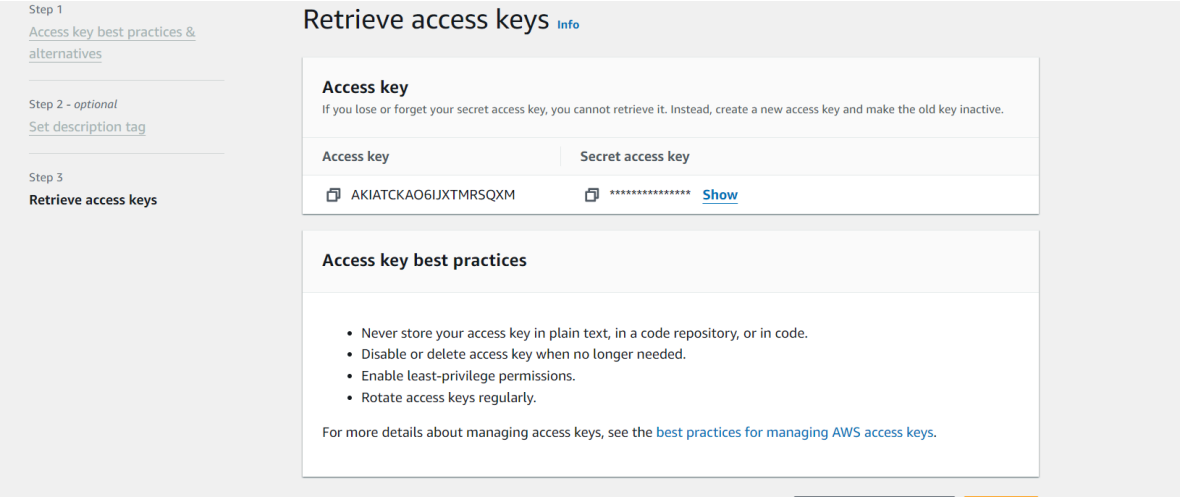
We need permission to be able to protect the bucket.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "Statement1",
            "Principal": "*",
            "Effect": "Allow",
            "Action": [
                "s3:GetObject"
            ],
            "Resource": [
                "arn:aws:s3:::gha-pipeline/*"
            ]
        }
    ]
}
```

While in AWS Create Access Key:



Access Key



Output Security Credentials

Now, back to my Workflow:

```yaml
name: 'Get  and Cache Dependencies'
description: ' Get the dependencies and Cache them'
inputs:
  bucket:
    description: ' The S3 Bucket Name'
    required: true

  bucket-region:
    description: ' The S3 Bucket region'
    required: true

  dist-folder:
    description: ' The folder containing artifacts'
    required: true

outputs:
  website-url:
    description: The Website URL

runs:
```

```
  using: 'node20'
  main: 'main.js'
```

The run part now will use Node.js version 20, and we will have to create a *main.js* file for the JavaScript code

We will have some inputs that must be passed:

- bucket
- bucket region
- dist folder

And also output the website URL once it's uploaded to the S3 bucket.

Let's look at the main.js:

First, let's install some packages we need :

Inside the .github> actions> deploy-s3-javascript

run

```
npm init -y
```

then:

```
npm install @actions/core @actions/github @actions/exec
```

and now the code :

```
const core = require('@actions/core')
const github = require('@actions/github')
const exec= require('@actions/exec')
async function run(){

    const bucket= core.getInput('bucket', {required:true})
    const bucketRegion= core.getInput('bucket-region', {required:true})
    const distFolder= core.getInput('dist-folder', {required:true})
```

```
    const s3Uri= `s3://${bucket}`
    exec.exec (` aws s3 sync ${distFolder} ${s3Uri} --region
${bucketRegion}`)

    const websiteUrl= `http://${bucket}.s3-
website-${bucketRegion}.amazonaws.com`
    core.setOutput('website-url', websiteUrl)
}

run()
```

```
const core = require('@actions/core')
const github = require('@actions/github')
const exec= require('@actions/exec')
```

Here we will require the packages we have just installed.

```
const bucket= core.getInput('bucket', {required:true})
const bucketRegion= core.getInput('bucket-region', {required:true})
const distFolder= core.getInput('dist-folder', {required:true})
```

then now we will read the inputs that will be passed and save then to the variables defined.

```
const s3Uri= `s3://${bucket}`
exec.exec (` aws s3 sync ${distFolder} ${s3Uri} --region
${bucketRegion}`)
```

Now let's construct the url then we will sync the dist folder received as input with the S3 bucket.

Then Lastly, we will construct the website URI too:

```
const websiteUrl= `http://${bucket}.s3-
website-${bucketRegion}.amazonaws.com`
core.setOutput('website-url', websiteUrl)
```

then this will set the output.

Now in the Main Workflow:

```
deploy:
  needs:  build
  runs-on: ubuntu-latest
  steps:
    - name: Get code
      uses: actions/checkout@v3

    - name: Get Build artifacts
      uses:  actions/download-artifact@v3
      with:
        name: dist-files
        path: ./dist
    - name: Deploy site
      id: deploy
      uses: ./.github/actions/deploy-s3-javascript
      env:
        AWS_ACCESS_KEY_ID: ${{secrets.AWS_ACCESS_KEY_ID}}
        AWS_SECRET_ACCESS_KEY: ${{secrets.AWS_SECRET_ACCESS_KEY}}
      with:
        bucket: gha-pipeline
        dist-folder: ./dist

    - name: Output information
      run: |
        echo " Live URL ${{steps.deploy.outputs.website-url}}"
```

We will first get the code, then download the uploaded artifacts (from the build job), and now we will use the custom actions.

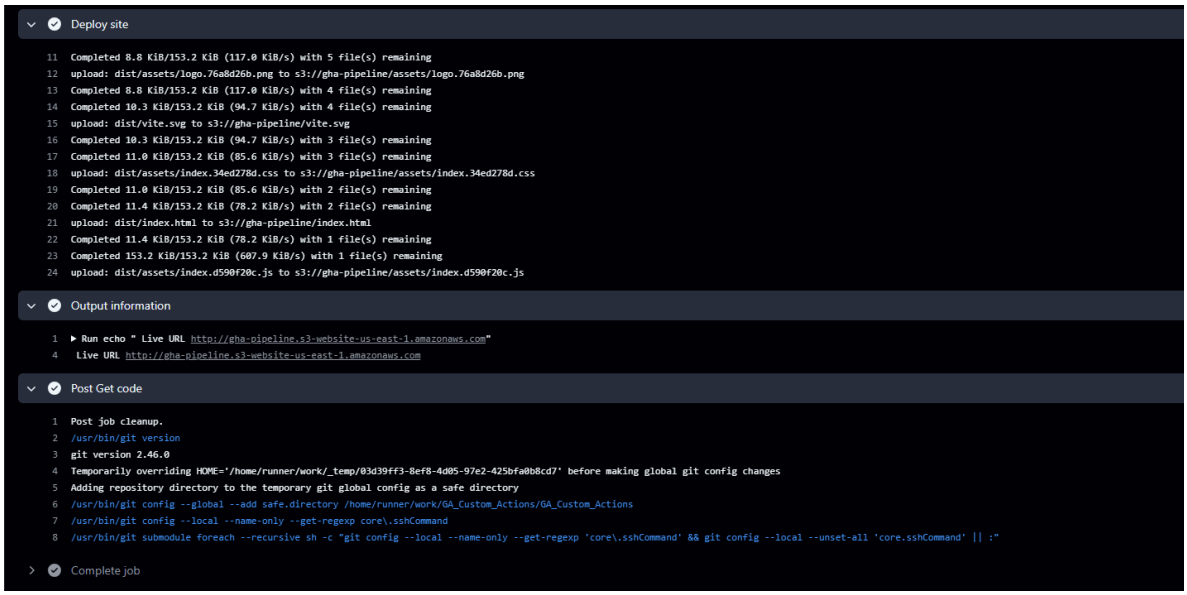
As part of the environment variables, we need to pass the access key ID and Access Secret Key for AWS authentication.

We will also pass the inputs:

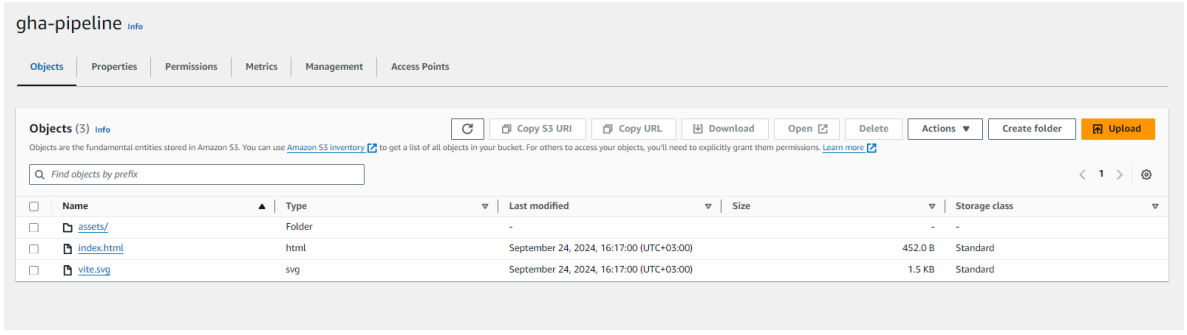
- The bucket
- the Dist-folder

The output the URL.

Outputs



Job Output



S3


Not secure
gha-pipeline.s3-website-us-east-1.amazonaws.com
Learn & Master GitHub Actions
Hide Help
What is Git?
Git is a version control system, helping you to manage your code and create code snapshots.
What is GitHub?
GitHub is a company and online offering, providing you with tons of Git-related services (e.g., cloud repositories).
What is GitHub Actions?
GitHub Actions is an automation service (or CI / CD service) that helps you automate repository-related workflows and processes.

Site