

Table of contents

GitHub Actions	2
First GitHub Action	4
GitHub Action Example Two	7
GitHub Events	14
Job Artifacts	17
Environment Variables & Secrets	25
Control Execution Flow	31
Custom Actions	42

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.

The screenshot shows the GitHub 'Create Repository' interface. It includes fields for 'Owner' (set to 'Teach2Give-Dotnet-Training'), 'Repository name' (set to 'GA-first-action'), and a note that it is available. There is a description field, a choice between 'Public' and 'Private' visibility (set to 'Public'), and options for initializing the repository with a README file (checked) and adding a .gitignore template (set to 'None'). A 'Choose a license' section is also present. At the bottom is a large 'Create Repository' button.

Import a repository.

Required fields are marked with an asterisk (*).

Owner * Repository name *

Teach2Give-Dotnet-Training / GA-first-action GA-first-action is available.

Great repository names are short and memorable. Need inspiration? How about potential-octo-fiesta ?

Description (optional)

Public Anyone on the internet can see this repository. You choose who can commit.

Private You choose who can see and commit to this repository.

Initialize this repository with:

Add a README file This is where you can write a long description for your project. [Learn more about READMEs](#).

Add .gitignore

.gitignore template:

Choose which files not to track from a list of templates. [Learn more about ignoring files](#).

Choose a license

Create Repository

Then click on the Actions tab> then configure.

The screenshot shows the GitHub Actions tab. It features a 'Get started with GitHub Actions' section with a search bar for workflows. Below it is a 'Suggested for this repository' section, which highlights a 'Simple workflow' by GitHub. This card includes a 'Configure' button. The top navigation bar includes tabs for Code, Issues, Pull requests, Actions (which is underlined), Projects, Wiki, Security, Insights, and Settings.

Get started with GitHub Actions

Build, test, and deploy your code. Make code reviews, branch management, and issue triaging work the way you want. Select a workflow to get started.

Skip this and set up a workflow yourself →

Search workflows

Suggested for this repository

Simple workflow By GitHub Start with a file with the minimum necessary structure.

Action

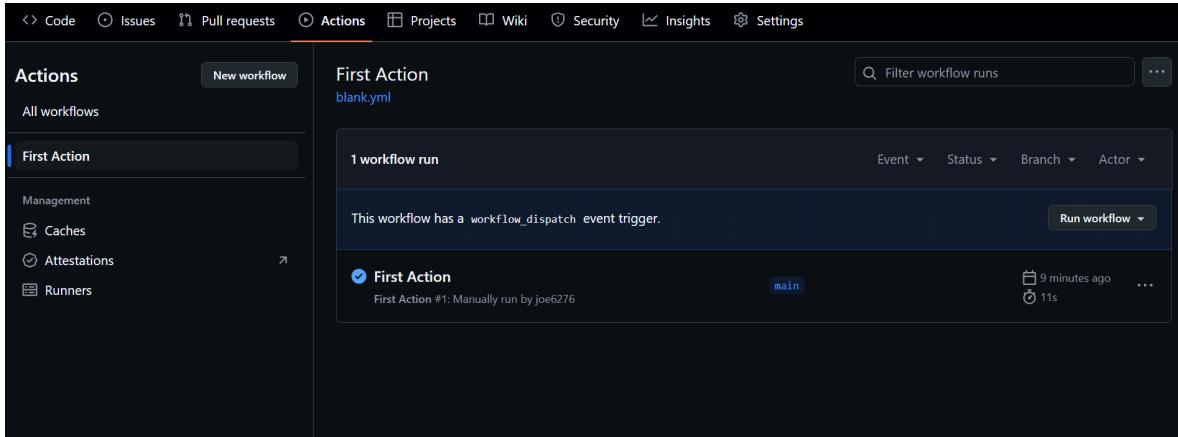
Now delete the steps on the .yml file and paste the following:

```
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.

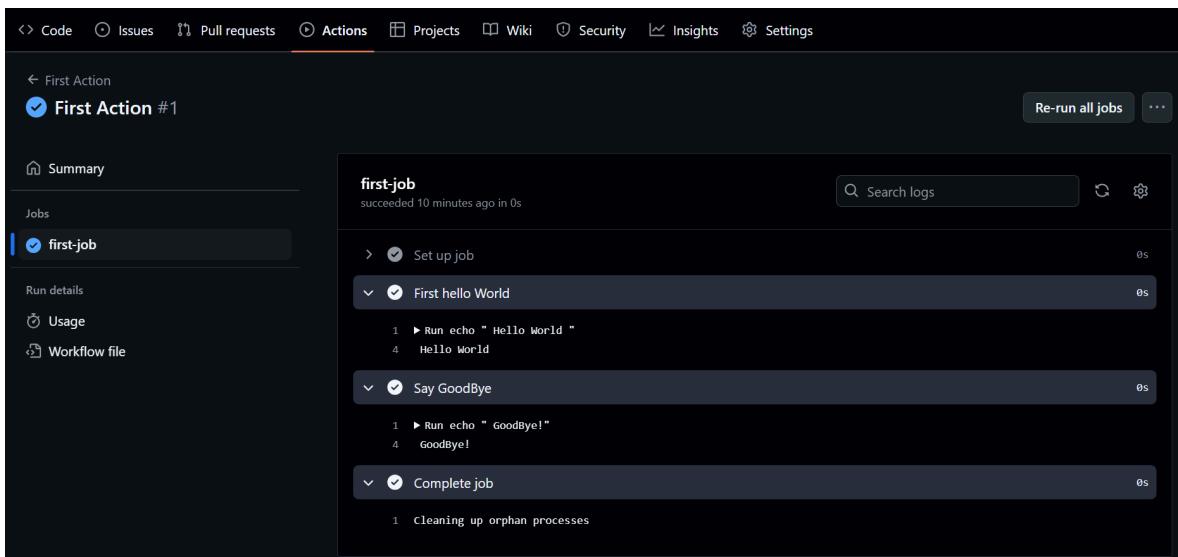
Run the workflow

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.

```
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

Repository-related			Other
push Pushing a commit	pull_request Pull request action (opened, closed, ...)	create A branch or tag was created	workflow_dispatch Manually trigger workflow
fork Repository was forked	issues An issue was opened, deleted, ...	issue_comment Issue or pull request comment action	repository_dispatch REST API request triggers workflow
watch Repository was starred	discussion Discussion action (created, deleted, ...)	Many More!	
			schedule Workflow is scheduled
			workflow_call Can be called by other workflows

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.

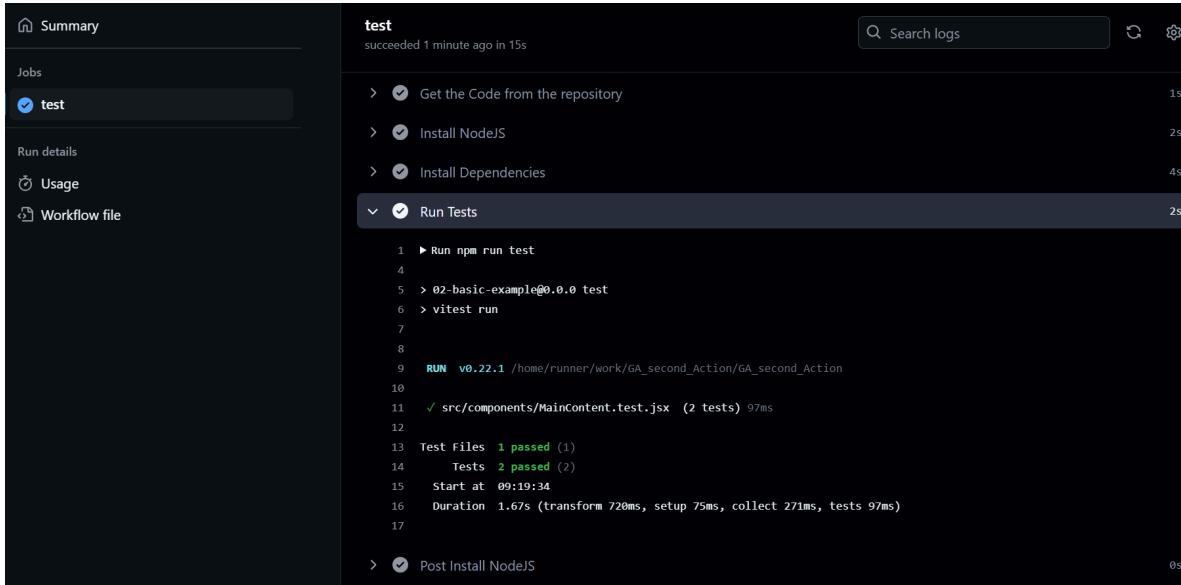
```
- name: Install Dependencies  
run: npm ci
```

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:

```

test
failed now in 13s
2s
Run Tests
Help
Failed Tests 1
FAIL src/components/MainContent.test.jsx > MainContent > should render a button
Error: expect(element).not.toBeInTheDocument()
expected document not to contain element, found <button>
Show
Help
</button> instead
> src/components/MainContent.test.jsx:11:43
    26   render(<MainContent />);
    27   |
    28   10|     expect(screen.getByRole('button')).not.toBeInTheDocument();
    29   11|   );
    30   12| });
    31   13|
Test Files 1 failed (1)
Tests 1 failed | 1 passed (2)
Start at 09:23:54
Duration 1.81s (transform 789ms, setup 73ms, collect 271ms, tests 106ms)
Error: Process completed with exit code 1.

```

Post Install NodeJS

Failure

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

```

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.

The screenshot shows the GitHub Actions interface for a workflow named "Running Test Workflow #3". The summary indicates the workflow was triggered via push 3 minutes ago by user "joe6276" to branch "master". The status is "Success" with a total duration of 24s and billable time of 2m. The workflow file is "example2.yml" and it runs on push events. Two parallel steps are listed: "test" (13s) and "deploy" (11s), both of which have passed successfully.

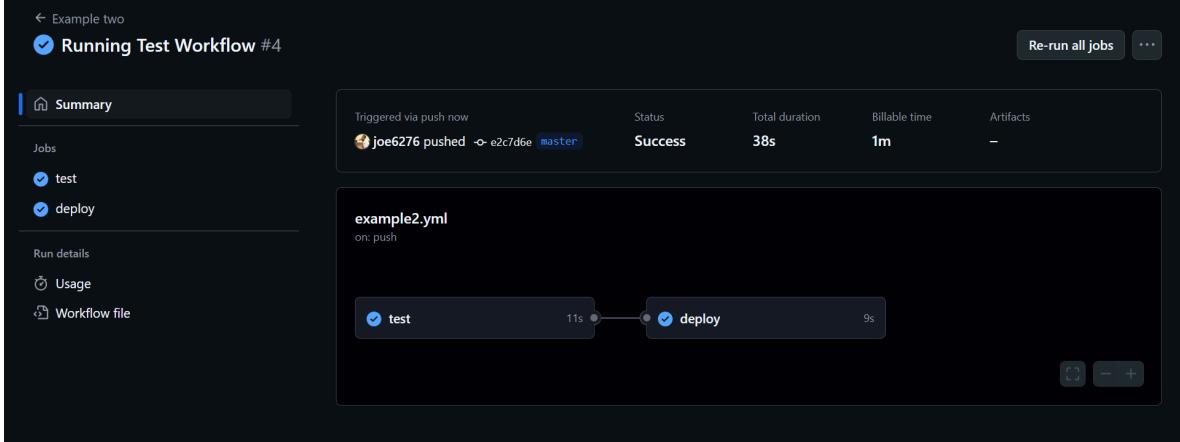
Job	Status	Total duration	Billable time	Artifacts
test	Success	13s	2m	-
deploy	Success	11s		-

Steps in Parallel

needs

To make this sequential, in the deploy code add:

```
deploy:  
  needs: test
```



sequential

Multi-Events

To trigger workflow based on multiple events

```
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

```
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

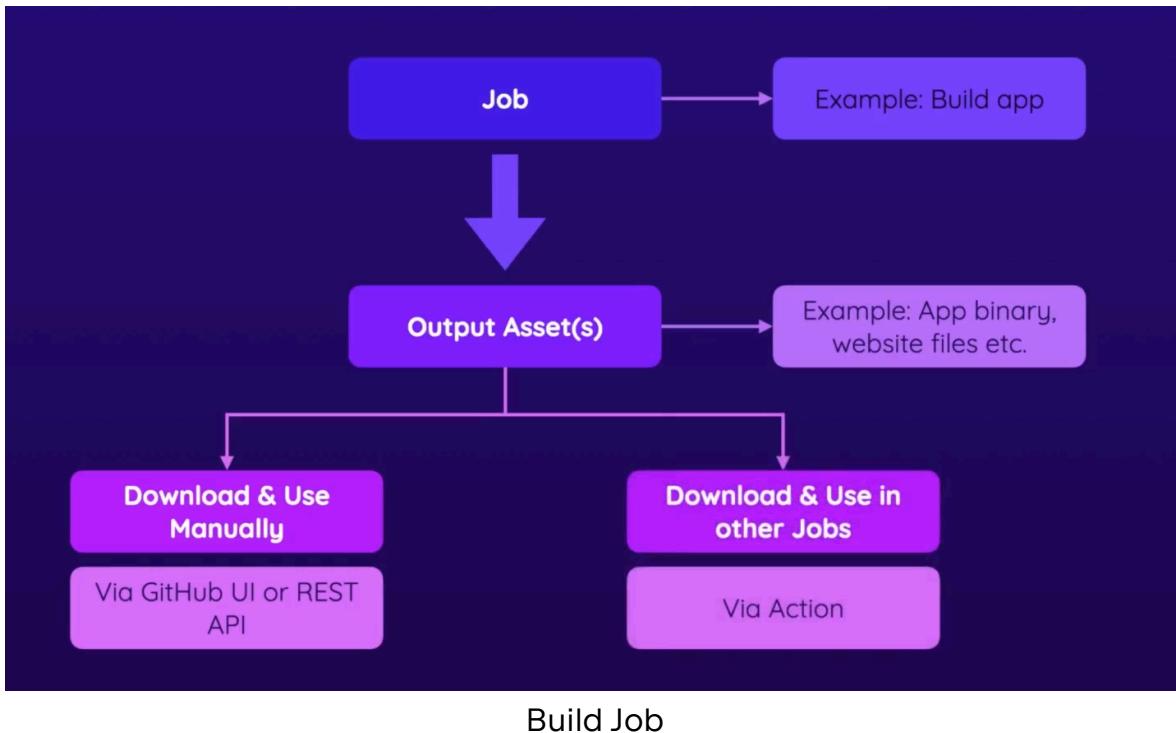
- Workflows get cancelled automatically when jobs fail
- You can manually cancel workflows

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:

```
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 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

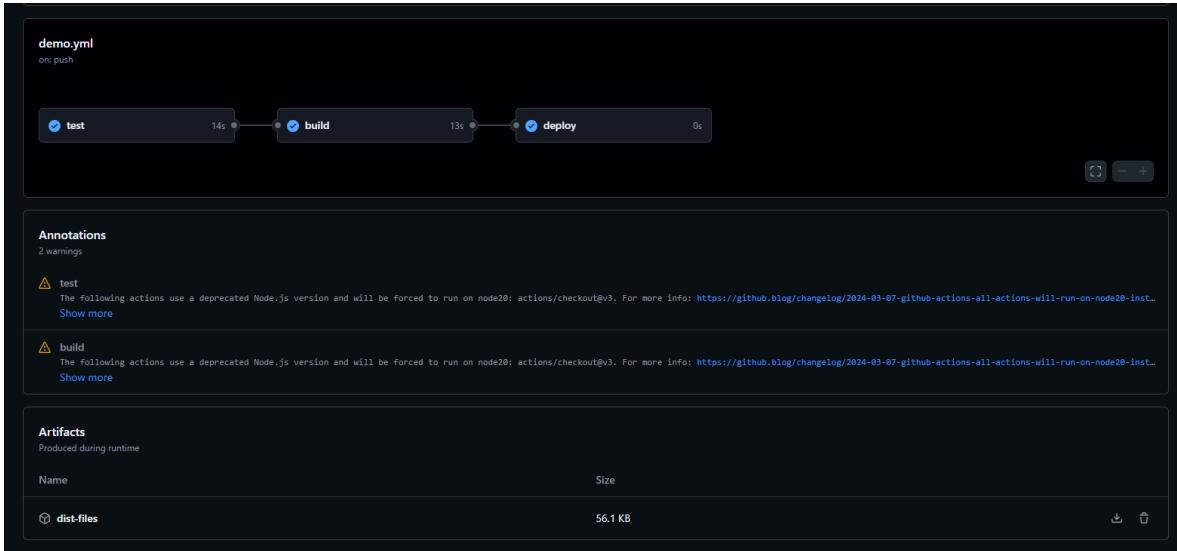
```
- name: Upload Artifacts
  uses: actions/upload-artifact@v4
```

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

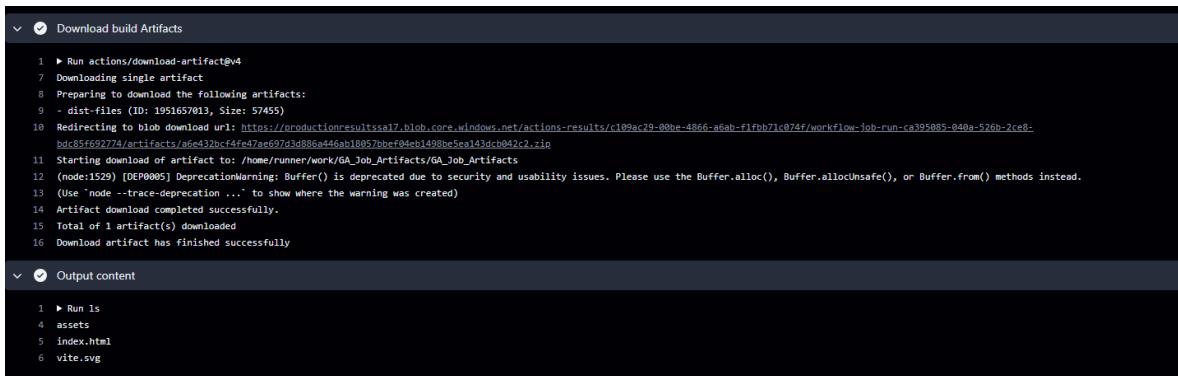
But we need to use the files in the deployment job. The files produced by the build process will not persist to the deployment job because they are on different runners even though they use the same runner definition. To get the artifacts, we need to download them in the deployment step. Let's modify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

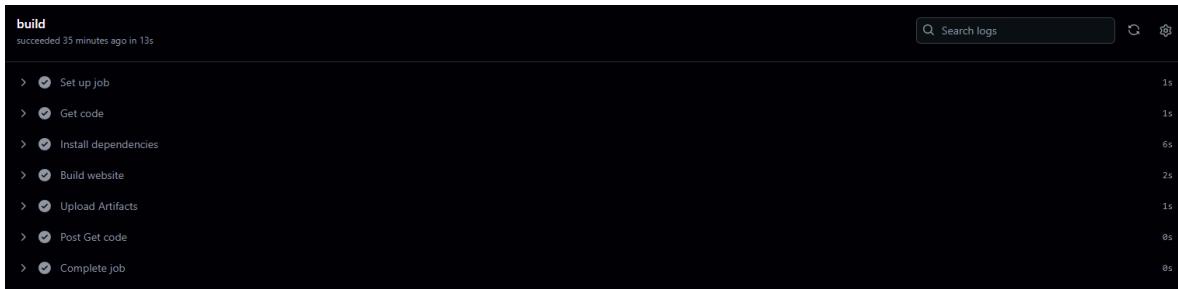


```
1 ▶ Run actions/download-artifact@v4
7 Downloading single artifact
8 Preparing to download the following artifacts:
9 - dist-files (ID: 1991657013, Size: 57455)
10 Redirecting to blob download url: https://orproductionresultsa17.blob.core.windows.net/actions-results/c109ac29-00be-4866-a5ab-f1fb71c974f/workflow-job-run-ca395085-040a-526b-2ce8-bdc85f692774/artifacts/a6e4320cf4fe47ae69763d896a44ab18057bbe0f04eb1498be5ea143dc0042c2.zip
11 Starting download of artifact to: /home/runner/work/GA_Job_Artifacts/GA_Job_Artifacts
12 (node:1529) [DEP0005] DeprecationWarning: Buffer() is deprecated due to security and usability issues. Please use the Buffer.alloc(), Buffer.allocUnsafe(), or Buffer.from() methods instead.
13 (Use `node --trace-deprecation ...` to show where the warning was created)
14 Artifact download completed successfully.
15 Total of 1 artifact(s) downloaded
16 Download artifact has finished successfully

✓ Output content
1 ▶ Run ls
4 assets
5 index.html
6 vite.svg
```

Download content

Dependencies Caching

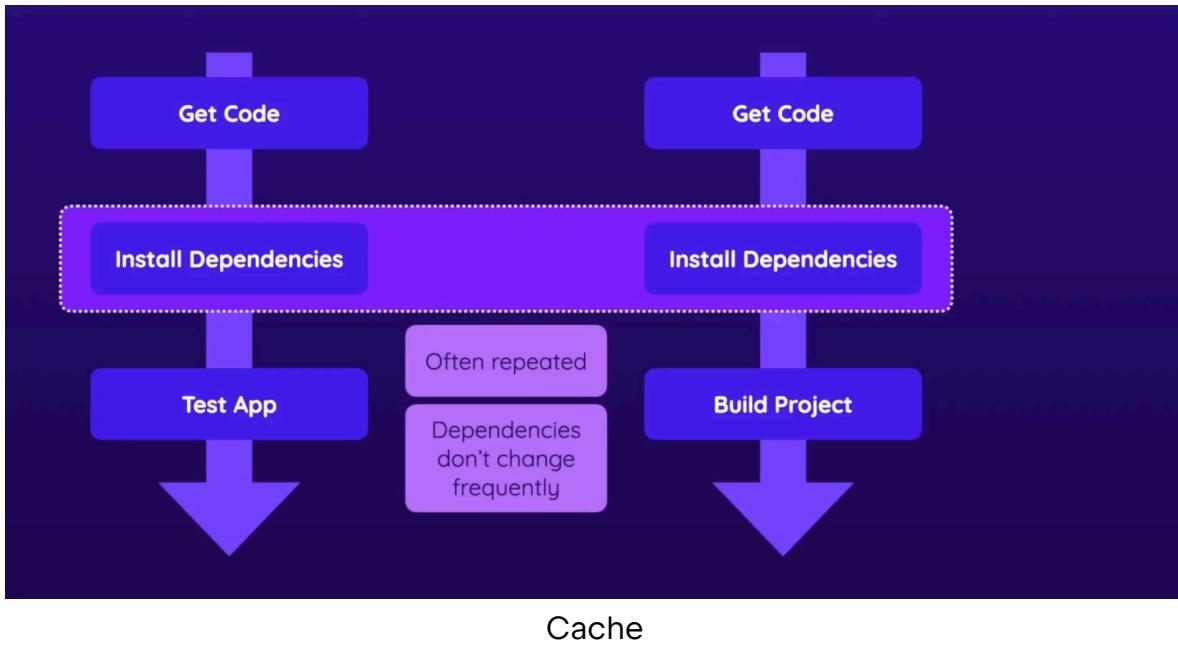


Step	Time
Set up job	1s
Get code	1s
Install dependencies	6s
Build website	2s
Upload Artifacts	1s
Post Get code	0s
Complete job	0s

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 them in the build step.



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
1 ► Run actions/cache@v4
9 Cache Size: >19 MB (19568772 B)
10 /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
11 Cache restored successfully
12 Cache restored from key: deps-node-modules- 5a782fe93589e2cbe03aac182697ecd9156b0019da2b2cd4ec8f48591088c823

Install dependencies
1 ► Run npm ci
4 added 375 packages, and audited 376 packages in 4s
5 77 packages are looking for funding
6   run `npm fund` for details
7 12 vulnerabilities (5 moderate, 6 high, 1 critical)
8 To address all issues, run:
9   npm audit fix
10 Run `npm audit` for details.

Build website
```

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:



A screenshot of a terminal window titled "Output Environment Variables". The window shows the following text:
1 ► Run echo "MONGO DB UserName : test"
11 MONGO DB UserName : test
12 At Port 80

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

The screenshot shows the 'General' settings page for a GitHub repository named 'GA_Environment'. The left sidebar contains sections for Access, Code and automation, Security, and Dependabot. The main area includes fields for 'Repository name' (GA_Environment), 'Default branch' (master), and a 'Social preview' section.

Setting

The screenshot shows the 'Actions secrets and variables' page. It features three main sections: 'Environment secrets' (which has none), 'Repository secrets' (which has none), and 'Organization secrets' (which also has none). Each section includes a 'Manage [secret type] secrets' button.

Action

Now add new repository secret:

Name *
MONGODB_USERNAME

Secret *
Test

Add secret

Secret

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

Repository secrets		New repository secret
Name	Last updated	
🔒 MONGODB_USERNAME	now	

Secret 1g

Now, how can we access the secrets:

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

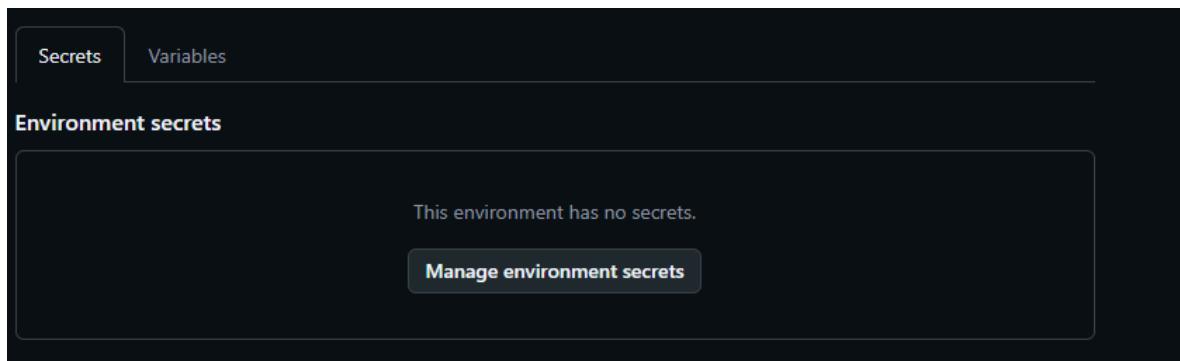
And Now even when you try to output, GitHub will hide the secret.

```
Output Environment Variables
Run echo "MONGO DB UserName : ***"
MONGO DB UserName : ***
```

Secret Output

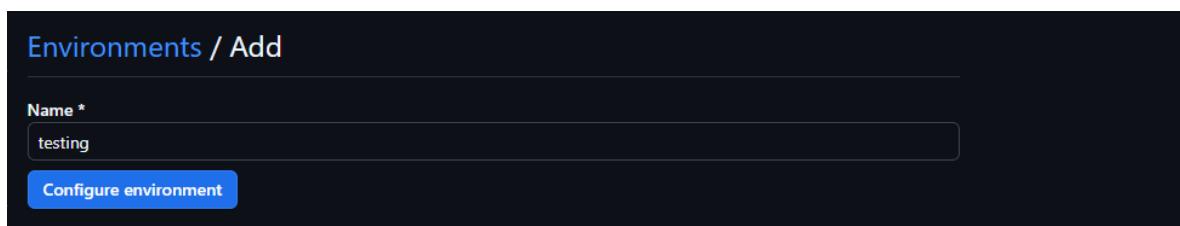
The above secrets are available for the whole workflow file, hence every job. This might work well, but sometimes you want to use different databases for different Jobs e.g., you want to use a database Named "Gh-Dev" for development and "GH-Testing" for testing. This is where environments can play a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

Now from your settings > Secrets & variables > Actions



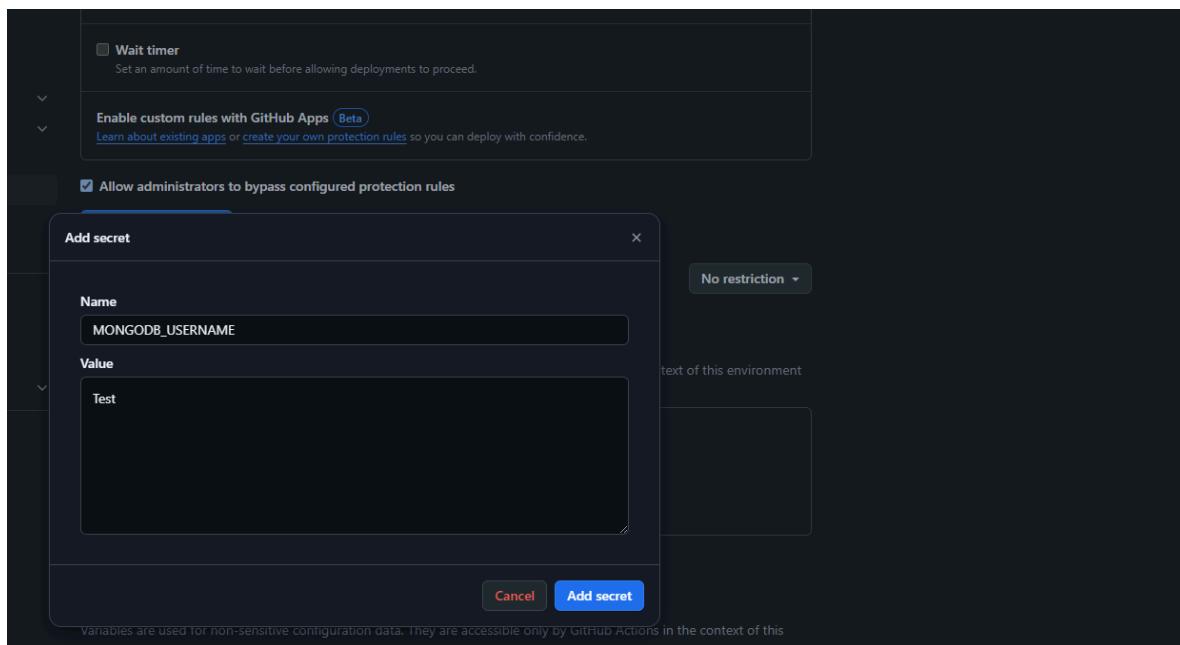
Enviroment

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.

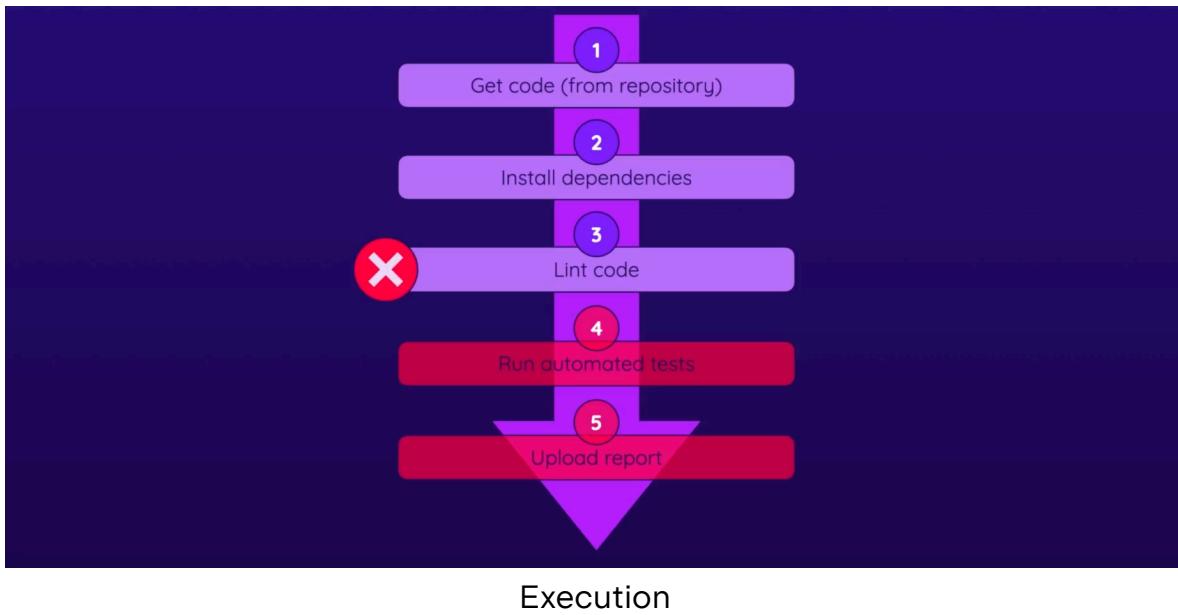
Summary

Environment Variables	Secrets	GitHub Actions Environments
Dynamic values used in code (e.g., database name)	Some dynamic values should not be exposed anywhere	Jobs can reference different GitHub Actions Environments
May differ from workflow to workflow	Examples: Database credentials, API keys etc.	Environments allow you to set up extra protection rules
Can be defined on Workflow-, Job- or Step-level	Secrets can be stored on Repository-level or via Environments	You can also store Secrets on Environment-level
Can be used in code and in the GitHub Actions Workflow	Secrets can be referenced via the <code>secrets</code> context object	
Accessible via interpolation and the <code>env</code> context object		

Summary

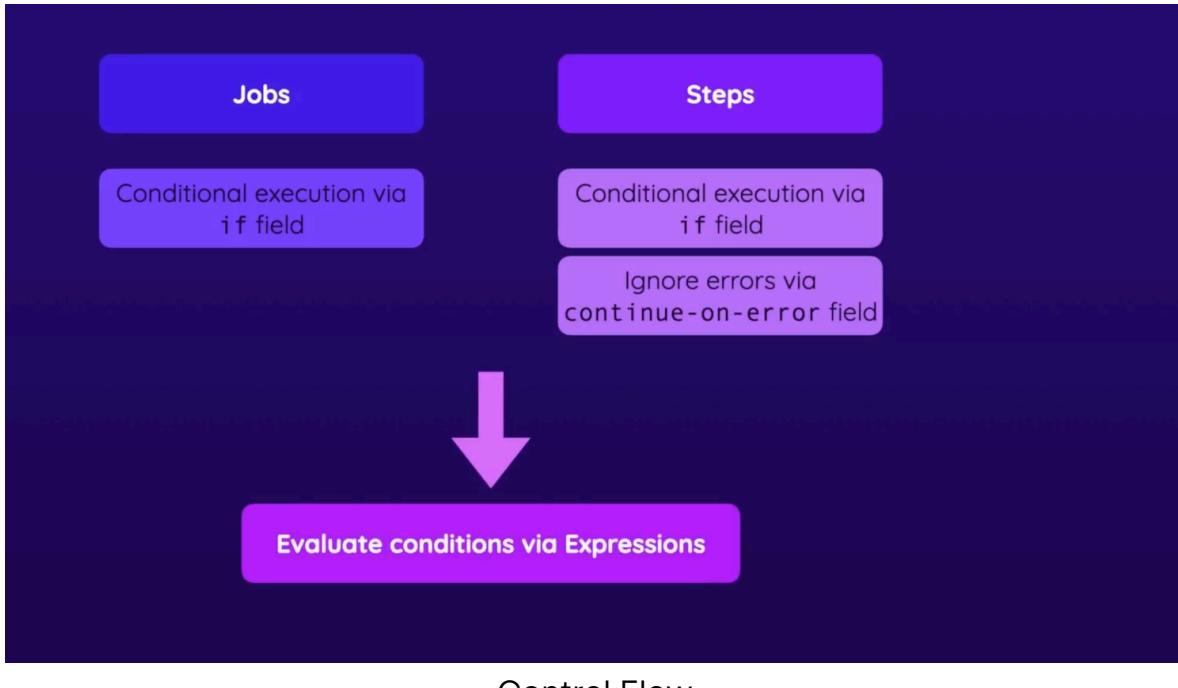
Control Execution Flow

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



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
```

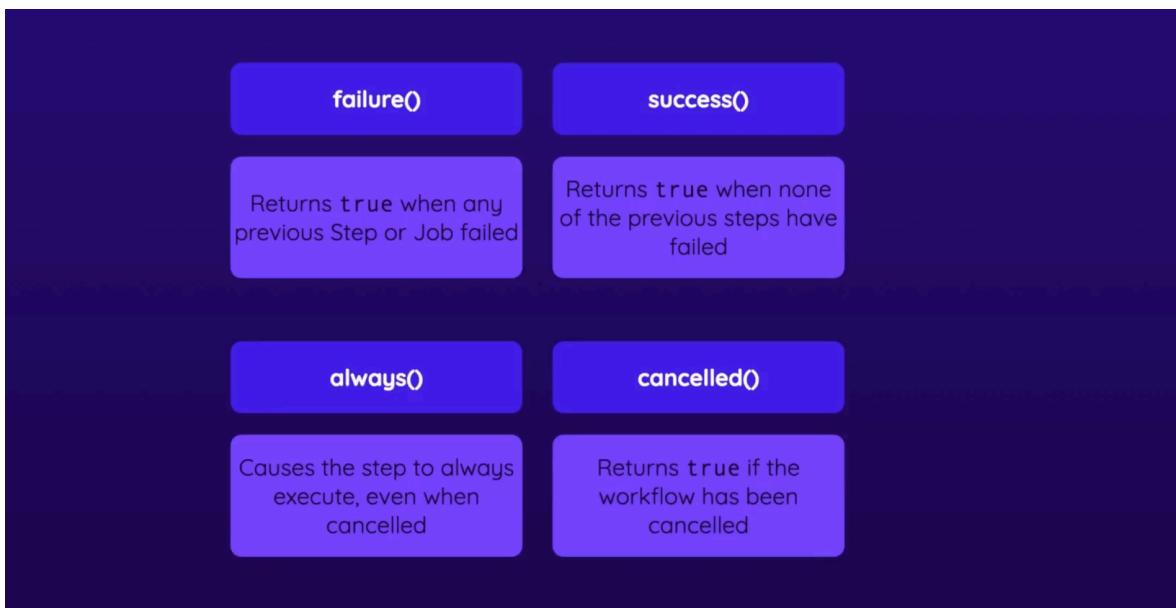
For the above case, it would make sense if we upload a test report if the tests fail. So we should run the Upload test report conditionally and only if when the test fail.

```
- 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
```

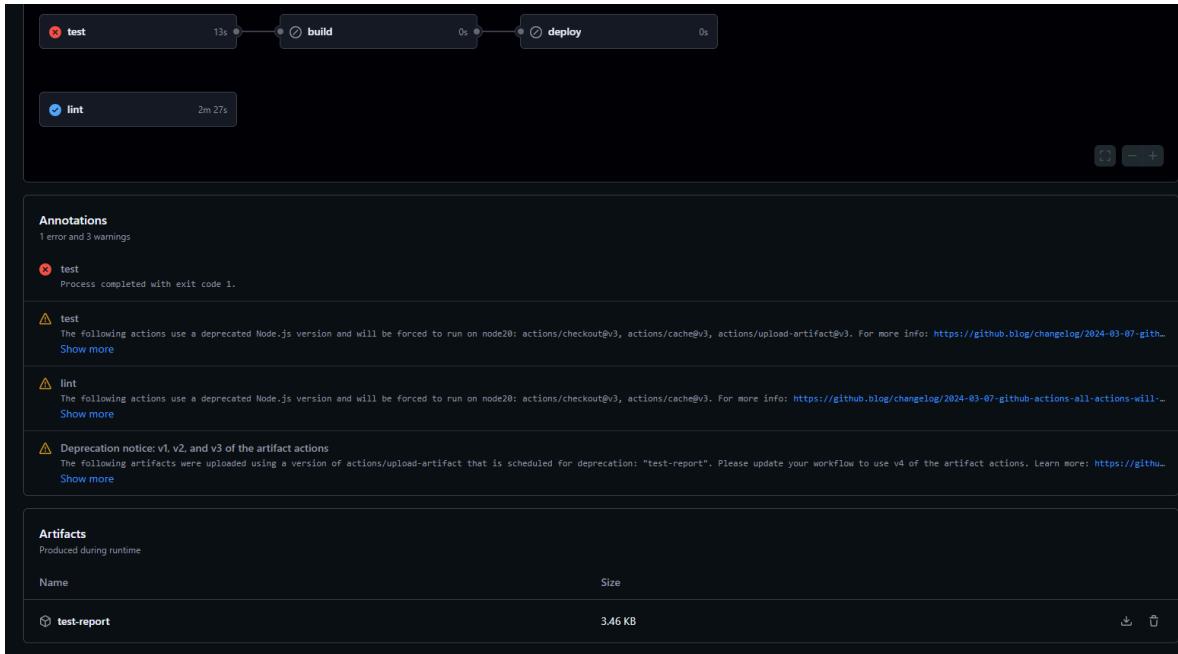
We will first start by assigning the step an ID so that we can know the outcome of the test step. On the upload test report step We execute the failure() which should return true if a previous step fails, and also the outcome of the previous step is checked. If both are true, then we allow the upload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



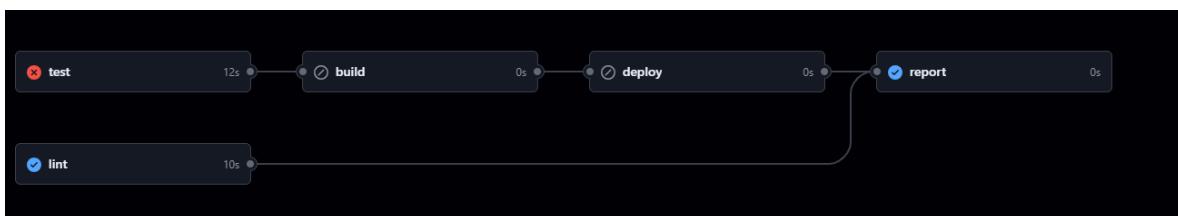
Report

Condition on Jobs

```
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 job fails

Output:



Conditional Jobs

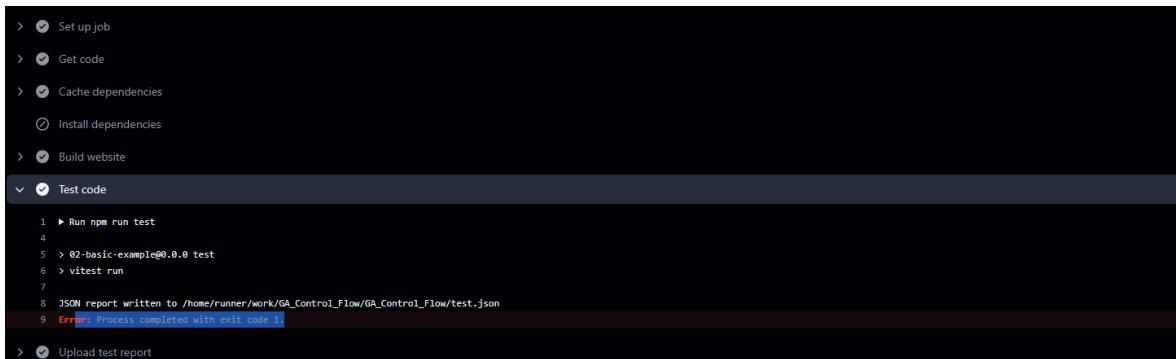
Continue

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:

```
- 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
```

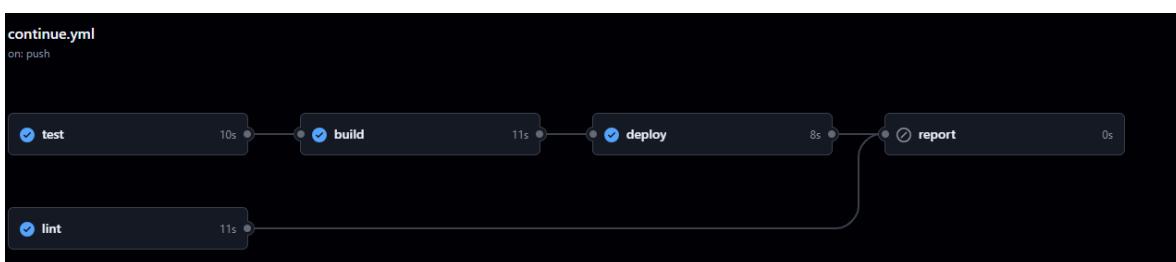
Output is: Even through the Test Step Fails:



The screenshot shows a GitHub Actions workflow log. The steps listed are: Set up job, Get code, Cache dependencies, Install dependencies, Build website, Test code, and Upload test report. The 'Test code' step is expanded, showing the command: Run npm run test. The output shows steps 1 through 8, followed by an error message: 'Error: Process completed with exit code 1'. The workflow then moves on to the 'Upload test report' step.

Test Fails

The Workflow succeeds:



Workflow Success

Matrix

Sometimes you have a workflow :

```
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:

```
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.

Job	Status	Time
build (12, ubuntu-latest)	Failed	14s
build (12, windows-latest)	Failed	33s
build (14, ubuntu-latest)	Passed	20s
build (14, windows-latest)	Passed	43s
build (16, ubuntu-latest)	Passed	16s
build (16, windows-latest)	Passed	41s

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 contribute 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 our starting Workflow:

```

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

```
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 path of where the action file is located:

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

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

The screenshot shows a GitHub Actions deployment interface. On the left, a sidebar lists jobs: lint, test, build (selected), and deploy. The main area shows a 'build' job that succeeded 43 minutes ago in 12s. The log details the following steps:

- Set up job (0s)
- Get code (1s)
- Load & Cache Dependencies (6s)
  - Prepare all required actions
  - Getting action download info
  - Download action repository 'actions/cache@v3' (SHA: e12d46a63a90f2fae62d114769bbf2a179198b5c)
  - Run ./github/actions/cache
  - Run actions/cache@v3
  - Cache Size: ~15 MB (15949666 B)
  - /usr/bin/tar -xf /home/runner/work/\_temp/23dc8a98-592f-452a-ae21-ef3583e9dc8b/cache.tzst -P -C /home/runner/work/GA\_Custom\_Actions/GA\_Custom\_Actions --use-compress-program unzstd
  - Cache restored successfully
  - Cache restored from key: deps-node-modules-5a782fe93589e2cbe03aac182697ecd9156b0019da2b2cd4ec8f40591080c823

## Workflow Output

Composite actions can also have inputs and outputs too.

## JavaScript Action

Let's create an S3 bucket

**General configuration**

AWS Region: US East (N. Virginia) us-east-1

Bucket type:  General purpose  
Recommended for most use cases and access patterns. General purpose buckets are the original S3 bucket type. They allow a mix of storage classes that redundantly store objects across multiple Availability Zones.

Directory  
Recommended for low-latency use cases. These buckets use only the S3 Express One Zone storage class, which provides faster processing of data within a single Availability Zone.

Bucket name:  Bucket name must be unique within the global namespace and follow the bucket naming rules. [See rules for bucket naming](#)

Copy settings from existing bucket - *optional*  
Only the bucket settings in the following configuration are copied.

Format: s3://bucket/prefix

**Object Ownership** [Info](#)  
Control ownership of objects written to this bucket from other AWS accounts and the use of access control lists (ACLs). Object ownership determines who can specify access to objects.

## S3 Creation

**Block all public access**  
 Turning this setting on is the same as turning on all four settings below. Each of the following settings are independent of one another.

- Block public access to buckets and objects granted through new access control lists (ACLs)**  
 S3 will block public access permissions applied to newly added buckets or objects, and prevent the creation of new public access ACLs for existing buckets and objects. This setting doesn't change any existing permissions that allow public access to S3 resources using ACLs.
- Block public access to buckets and objects granted through any access control lists (ACLs)**  
 S3 will ignore all ACLs that grant public access to buckets and objects.
- Block public access to buckets and objects granted through new public bucket or access point policies**  
 S3 will block new bucket and access point policies that grant public access to buckets and objects. This setting doesn't change any existing policies that allow public access to S3 resources.
- Block public and cross-account access to buckets and objects through any public bucket or access point policies**  
 S3 will ignore public and cross-account access for buckets or access points with policies that grant public access to buckets and objects.

**⚠️ Turning off block all public access might result in this bucket and the objects within becoming public**

AWS recommends that you turn on block all public access, unless public access is required for specific and verified use cases such as static website hosting.

I acknowledge that the current settings might result in this bucket and the objects within becoming public.

## 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:

The screenshot shows the AWS Identity and Access Management (IAM) console. The left sidebar has a search bar with 'postgres' typed in. The main area is titled 'Access keys (1)' and contains a table with one row. The table columns are 'Actions', 'Status', 'Created', 'Last used service', 'Last used region', 'Last used service', and 'Last used region'. The status is 'Active', created 5 hours ago, and last used service is 'N/A'. Below this table is another section titled 'X.509 Signing certificates (0)' with a 'Create X.509 certificate' button.

| Actions                   | Status | Created     | Last used service | Last used region | Last used service | Last used region |
|---------------------------|--------|-------------|-------------------|------------------|-------------------|------------------|
| <a href="#">Actions ▾</a> | Active | 5 hours ago | N/A               | N/A              | N/A               | N/A              |

Access Key

The screenshot shows the AWS IAM Access Key Management interface. On the left, there's a sidebar with steps: Step 1 (Access key best practices & alternatives), Step 2 - optional (Set description tag), and Step 3 (Retrieve access keys). The main area is titled "Retrieve access keys" and contains an "Access key" section with a table. The table has two columns: "Access key" and "Secret access key". The "Access key" column shows a placeholder icon and the value "AKIATCKAO6IJXTMRSQXM". The "Secret access key" column shows a placeholder icon and the value "\*\*\*\*\*" followed by a "Show" link. Below this is a section titled "Access key best practices" with a bulleted list: "Never store your access key in plain text, in a code repository, or in code.", "Disable or delete access key when no longer needed.", "Enable least-privilege permissions.", and "Rotate access keys regularly.". A note at the bottom says "For more details about managing access keys, see the [best practices for managing AWS access keys](#)".

## Output Security Credentials

Now, back to my Workflow:

```
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 them 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.

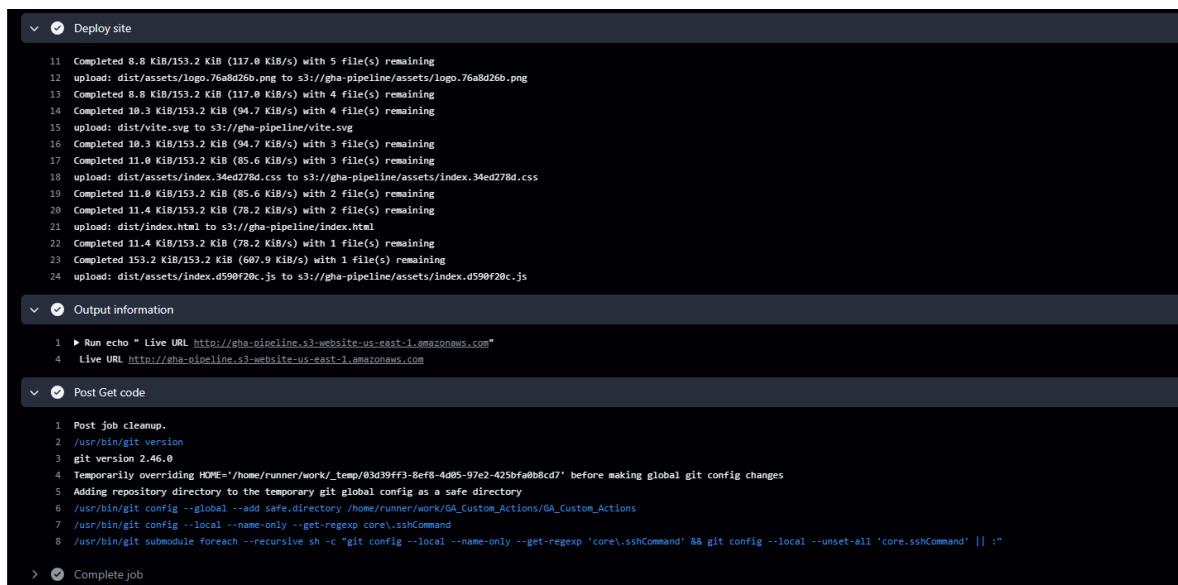
As part of the environment variables, we need to pass the access key ID and Access Secret Key for AWS authentication.

We will also pass the inputs:

- The bucket
- the Dist-folder

The output the URL.

## Outputs



The screenshot shows a GitHub Actions deployment log. It includes sections for 'Deploy site', 'Output information', and 'Post Get code'. The 'Deploy site' section lists file upload details. The 'Output information' section shows a live URL. The 'Post Get code' section contains a series of commands related to job cleanup and git configuration.

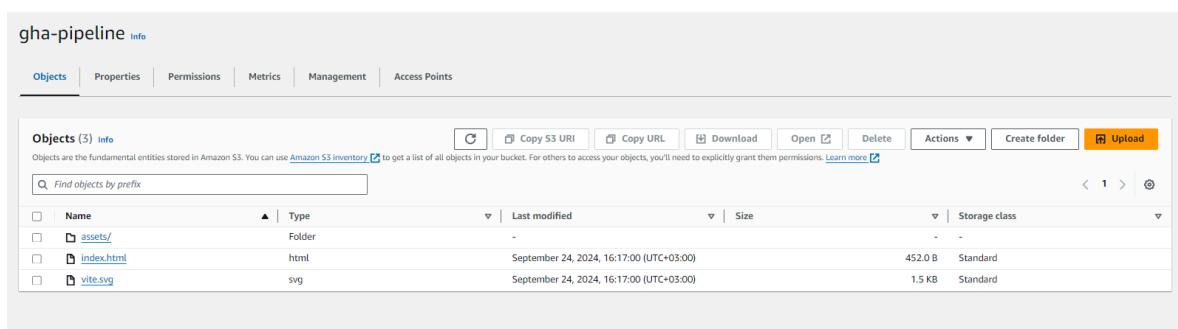
```
11 Completed 8.8 KiB/153.2 KiB (117.0 KiB/s) with 5 file(s) remaining
12 upload: dist/assets/logo.76a8d26b.png to s3://gha-pipeline/assets/logo.76a8d26b.png
13 Completed 8.8 KiB/153.2 KiB (117.0 KiB/s) with 4 file(s) remaining
14 Completed 10.3 KiB/153.2 KiB (94.7 KiB/s) with 4 file(s) remaining
15 upload: dist/vite.svg to s3://gha-pipeline/vite.svg
16 Completed 10.3 KiB/153.2 KiB (94.7 KiB/s) with 3 file(s) remaining
17 Completed 11.0 KiB/153.2 KiB (85.6 KiB/s) with 3 file(s) remaining
18 upload: dist/assets/index.34ed278d.css to s3://gha-pipeline/assets/index.34ed278d.css
19 Completed 11.0 KiB/153.2 KiB (85.6 KiB/s) with 2 file(s) remaining
20 Completed 11.4 KiB/153.2 KiB (78.2 KiB/s) with 2 file(s) remaining
21 upload: dist/index.html to s3://gha-pipeline/index.html
22 Completed 11.4 KiB/153.2 KiB (78.2 KiB/s) with 1 file(s) remaining
23 Completed 153.2 KiB/153.2 KiB (607.9 KiB/s) with 1 file(s) remaining
24 upload: dist/assets/index.d590f20c.js to s3://gha-pipeline/assets/index.d590f20c.js

▶ Run echo "Live URL http://gha-pipeline.s3-website-us-east-1.amazonaws.com"
Live URL http://gha-pipeline.s3-website-us-east-1.amazonaws.com

Post job cleanup.
/usr/bin/git version
git version 2.46.0
Temporarily overriding HOME='"/home/runner/work/_temp/03d39ff3-8ef8-4d05-97e2-425bfa0b8cd7' before making global git config changes
Adding repository directory to the temporary git global config as a safe directory
/usr/bin/git config --global --add safe.directory /home/runner/work/GA_Custom_Actions/GA_Custom_Actions
/usr/bin/git config --local --name-only --get-regexp core\\.sshCommand
/usr/bin/git submodule foreach --recursive sh -c "git config --local --name-only --get-regexp 'core\\.sshCommand' && git config --local --unset-all 'core.sshCommand' || :"

> Complete job
```

Job Output



The screenshot shows the AWS S3 console for the 'gha-pipeline' bucket. It displays a list of objects: 'assets/' (Folder), 'index.html' (html), and 'vite.svg' (svg). The objects were uploaded on September 24, 2024, at 16:17:00 (UTC+05:00). The storage class for all objects is Standard.

| Name       | Type   | Last modified                            | Size    | Storage class |
|------------|--------|------------------------------------------|---------|---------------|
| assets/    | Folder | -                                        | -       | -             |
| index.html | html   | September 24, 2024, 16:17:00 (UTC+05:00) | 452.0 B | Standard      |
| vite.svg   | svg    | September 24, 2024, 16:17:00 (UTC+05:00) | 1.5 KB  | Standard      |

S3



Site