GitHub Actions

Lecture Slides

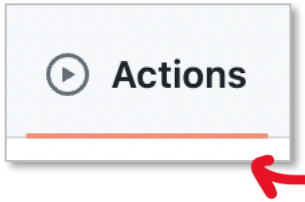
Good luck on your exam!

Introduction to GitHub Actions

GitHub Actions is a **CI/CD pipeline directly integrated with your GitHub repository.**

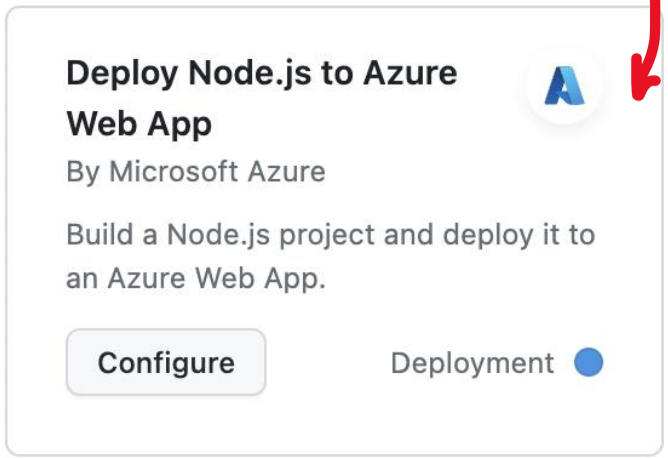
GitHub Actions allows you to **automate**:

- Running test suites
- Building images
- Compiling static sites
- Deploying code to servers
- and more…



Within a GitHub repo, you'll have a **tab for Actions**.

GitHub Actions has **templates** you can use to get started.



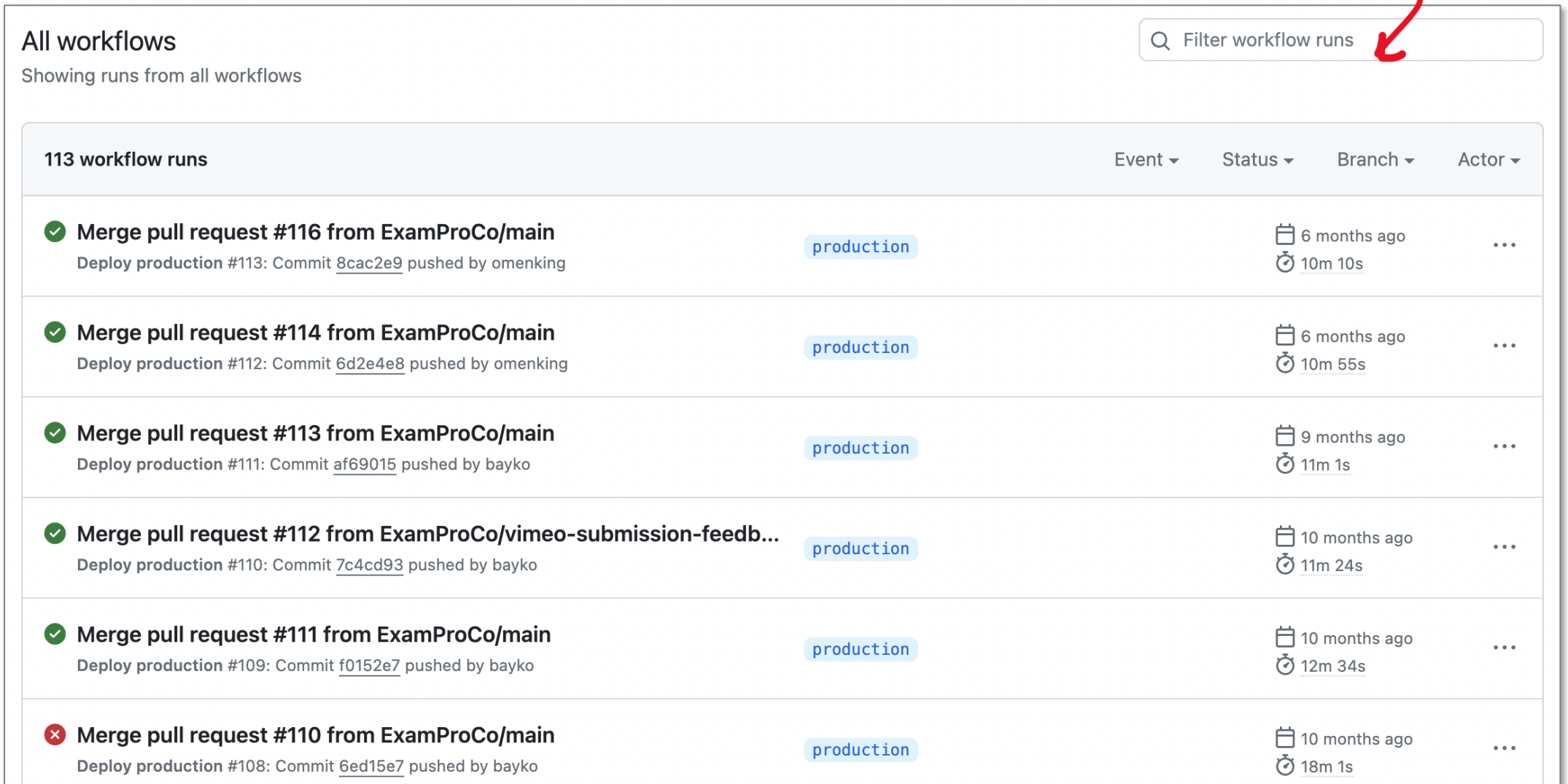
GitHub Actions files are defined as YAML files located in the **.github/workflow** folder in your repo.

You can have multiple workflows in the repo triggered by different events.

```yaml
on:
  push:
    branches: [ $default-branch ]
  workflow_dispatch:
env:
  AZURE_WEBAPP_NAME: your-app-name
  AZURE_WEBAPP_PACKAGE_PATH: '.'
  NODE_VERSION: '14.x'
permissions:
  contents: read
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v3
    ...
```

Introduction to GitHub Actions

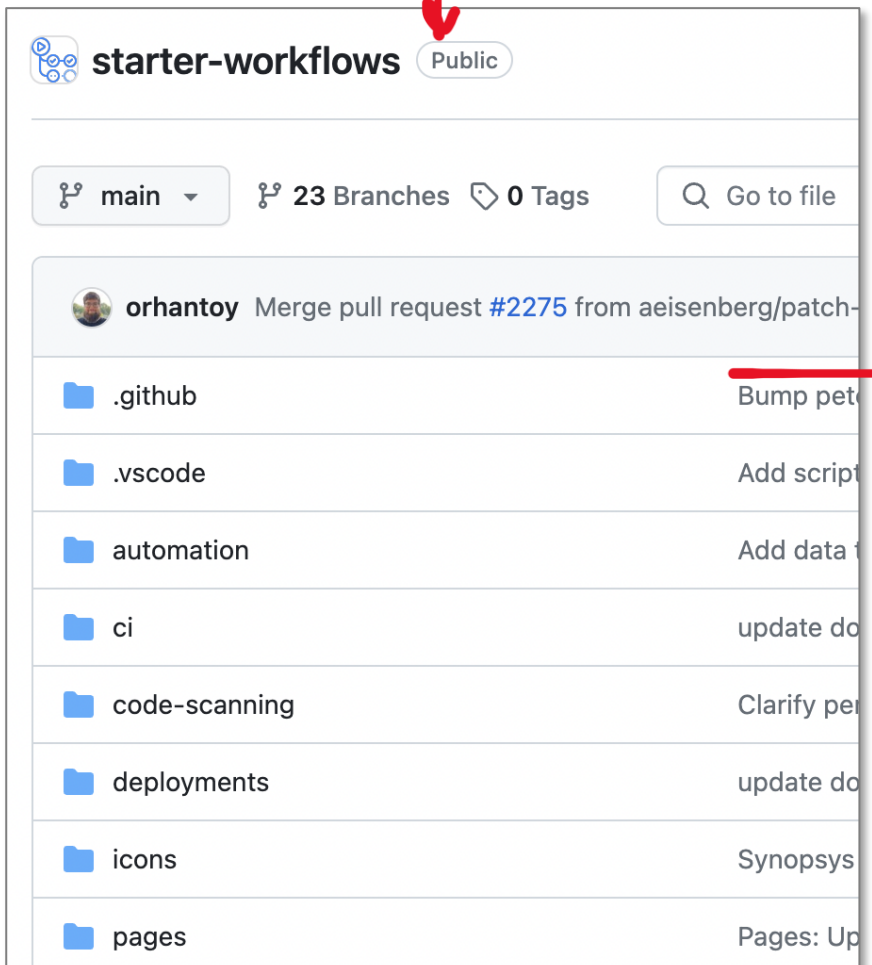
When you run GitHub Actions, you'll get **a history of workflow runs,** which will indicate if it was a success or a failure, and how long it took to run.

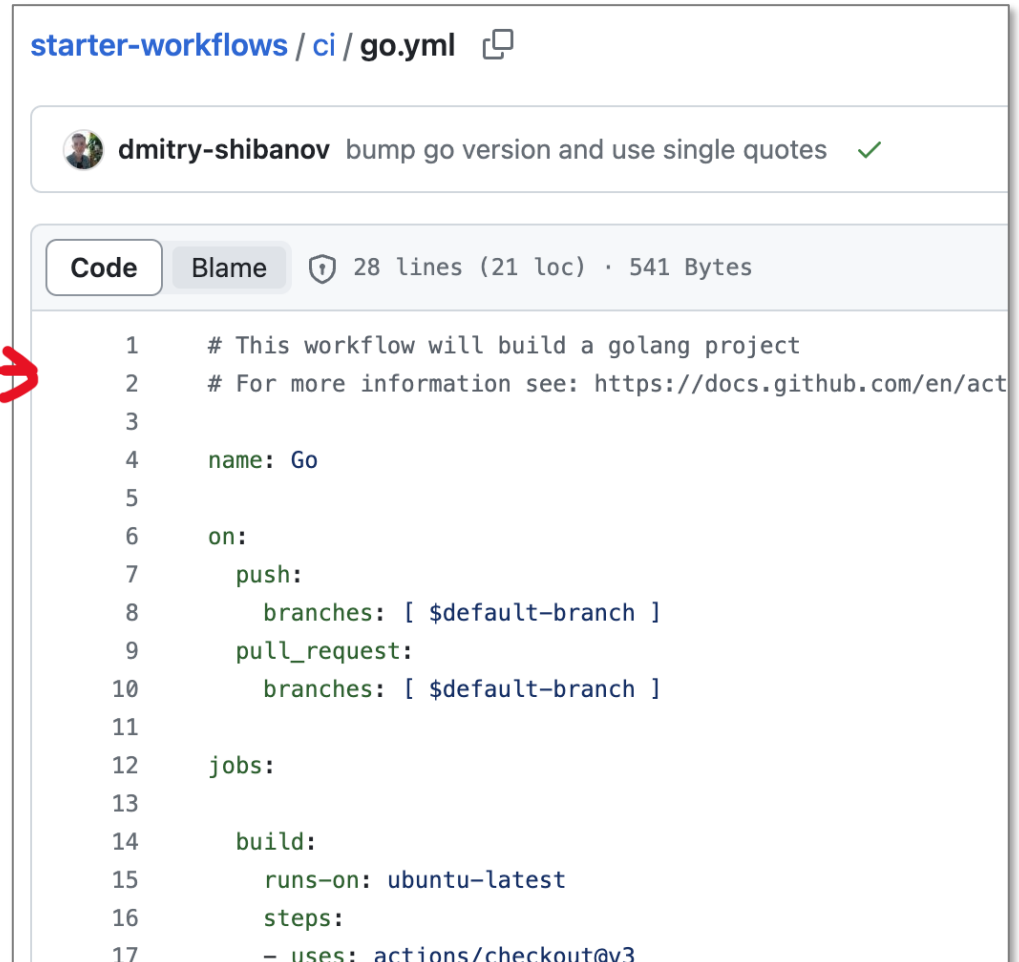


Finding Github Actions

Github has a repo of example workflows you can use to get you started

https://github.com/actions/starter-workflows





Different Types of GitHub Actions

Event Triggers causes a GitHub Action to run.

The **on** attribute specifies
the **event trigger** to be used:

```yaml
name: Example Workflow
on: [push]
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v2
    - name: Run a script
      run: echo "This script runs on every push!"
```

Github Actions has 35+ event triggers

Examples of common GitHub Actions that could be triggered:

- **Pushes:** Trigger an action on any push to the repository.
- **Pull Requests:** Run actions when pull requests are opened, updated, or merged.
- **Issues:** Execute actions based on issue activities, like creation or labeling.
- **Releases:** Automate workflows when a new release is published.
- **Scheduled Events:** Schedule actions to run at specific times.
- **Manual Triggers:** Allow manual triggering of actions through the GitHub UI.

Workflows

A **workflow** is a **configurable automated process** that will run one or more jobs.
Workflows are defined by a YAML file checked into your repository

Workflows in your repo are defined in the following directory: .github/workflows

A repo can contain multiple workflows

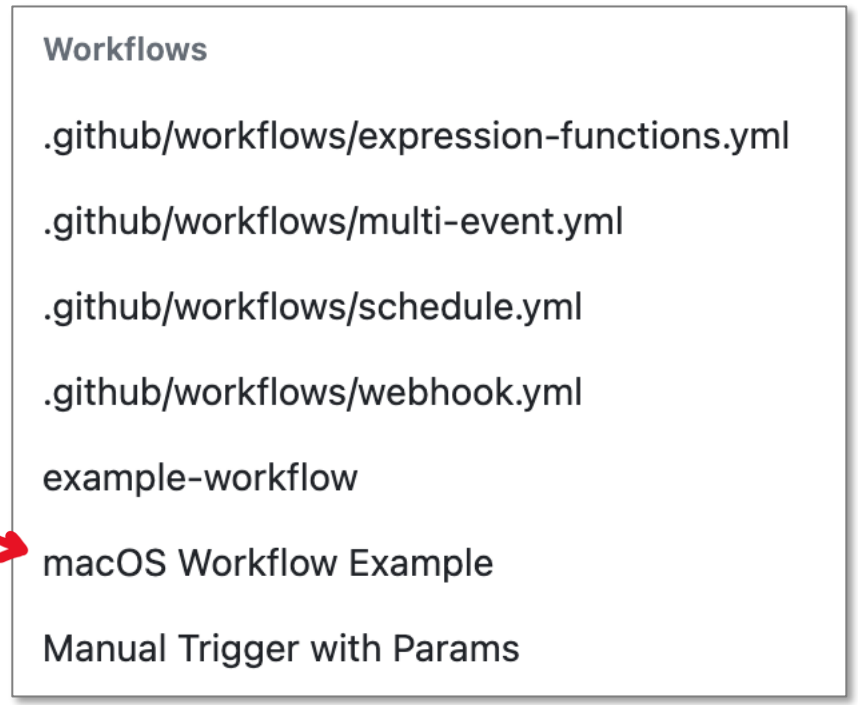
Workflow triggers are events that cause a workflow to run:

- Events that occur in your workflow's repository
- Events that occur outside of GitHub and trigger a **repository_dispatch** event on GitHub
- Scheduled times
- Manual

Workflows

The **name property** lets you easily identify Workflows.

```yaml
name: macOS Workflow Example

# ... rest of workflow template
```



Triggering Schedule Event

Schedule can use a **cron expression** to trigger a workflow at a specific time or day.

```yaml
on:
  schedule:
    - cron: '30 5 * * 1,3'
    - cron: '30 5 * * 2,4'

jobs:
  test_schedule:
    runs-on: ubuntu-latest
    steps:
      - name: Not on Monday or Wednesday
        if: github.event.schedule != '30 5 * * 1,3'
        run: echo "Skip this step on Monday and Wednesday"
      - name: Every time
        run: echo "This step will always run"
```

You can use https://crontab.guru/ to translate time into a cron expression

Triggering Single or Multiple Events

Single event. eg push

```yaml
name: CI on Push

on:
  push:
    branches:
      - main

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Run a one-line script
        run: echo "Hello, world!"
```

Multiple events eg. push,pull request, release

```yaml
name: CI on Multiple Events

on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main
  release:
    types: [published, created]

jobs:
  build-and-test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Set up Python
      # ....
```

If you specify multiple events, only one of those events needs to occur to trigger your workflow.
If multiple triggering events for your workflow occur at the same time, multiple workflow runs will be triggered.

Manual Events

GitHub allows you to trigger workflows manually using the GitHub UI, CLI (`gh`), or REST API, provided the workflow is set up for manual dispatch using the `workflow_dispatch` event.

Using the **gh workflow run** command:

```
gh workflow run greet.yml \
-f name=mona \
-f greeting=hello \
-F data=@myfile.txt
```

You can specify the name, ID, or **file name** of the workflow you want to run

Manual Events

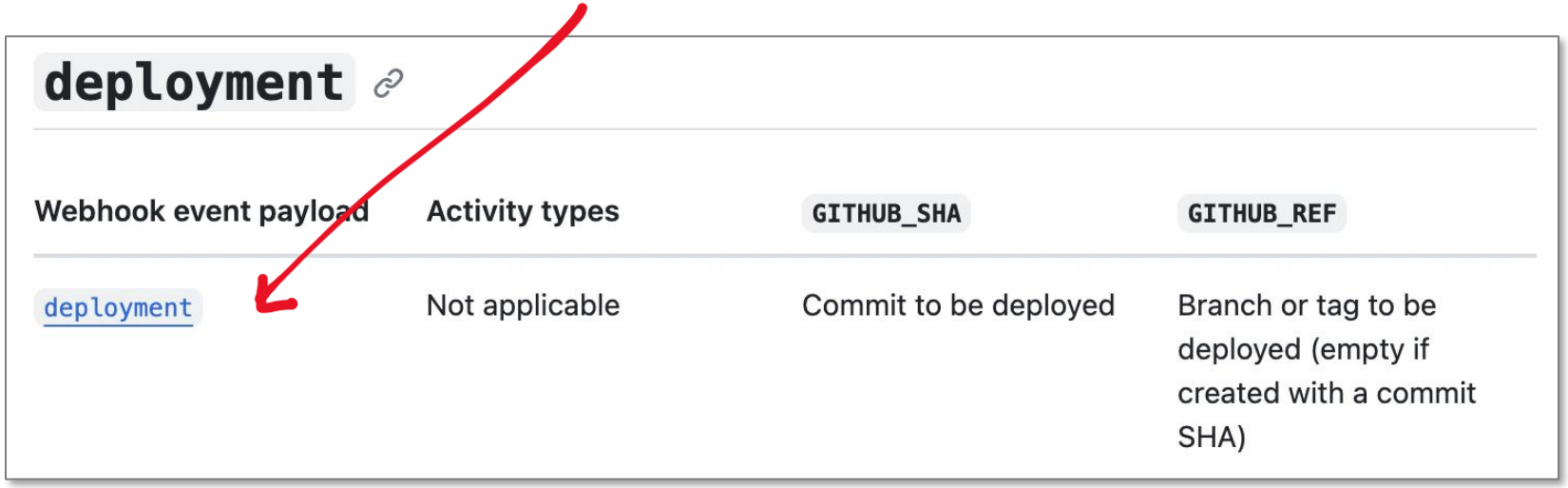
To run a workflow manually, the workflow must be configured to run on the **workflow_dispatch** event.

```yaml
on:
  workflow_dispatch:
    inputs:
      name:
        description: 'Name of the person to greet'
        required: true
        type: string
      greeting:
        description: 'Type of greeting'
        required: true
        type: string
      data:
        description: 'Base64 encoded content of a file'
        required: false
        type: string
```

You can define up to 10 inputs for a workflow_dispatch event.

Webhook Events

Many of the listed GitHub Workflow Triggers
are triggered by a **webhook**



deployment

Webhook event payload	Activity types	GITHUB_SHA	GITHUB_REF
deployment	Not applicable	Commit to be deployed	Branch or tag to be deployed (empty if created with a commit SHA)

What is a Webhook?
A webhook is a public-facing URL that can be sent an HTTP request (often requiring authorization) to trigger events from external sources.

Most of these webhooks will be triggered within GitHub when users are interacting with GitHub which will in turn will trigger API actions. Users generally don't have to directly call the API to trigger the workflow.

External Webhook Events

```yaml
name: Workflow on Repository Dispatch

on:
  repository_dispatch:
    types:
      - webhook

jobs:
  respond-to-dispatch:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout repository
        uses: actions/checkout@v2
      - name: Run a script
        run: echo "Event of type ${GITHUB_EVENT_NAME}"
```

Using **repository_dispatch** with **webhook** type you can trigger the Workflow via an external HTTP endpoint.

- will only trigger a workflow run if the workflow file is on the default branch

When you **make the request to the webhook,** you must:

- Send a POST request to the rep's dispatches endpoint
- Set the Accept type for application/vnd.github+json
- Provide Authorization for your Personal Access Token
- Pass the event type "webhook"

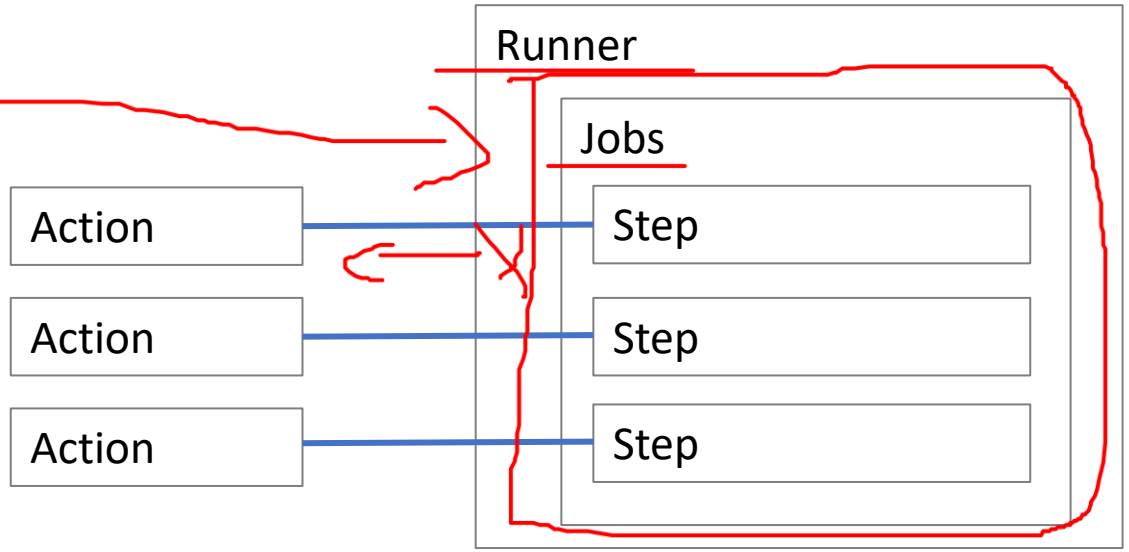
```
curl -X POST \
-H "Accept: application/vnd.github+json" \
-H "Authorization: token {personal token with repo access}" \
-d '{"event_type": "webhook", "client_payload": {"key": "value"}}' \
https://api.github.com/repos/{owner}/{repo}/dispatches
```

Workflow Syntax

Workflow Components

The following GitHub Actions Workflows components:

- **Actions:** Reusable tasks that perform specific jobs within a workflow.
- **Workflows:** Automated processes defined in your repository that coordinate one or more jobs, triggered by events or on a schedule.
- **Jobs:** Groups of steps that execute on the same runner, typically running in parallel unless configured otherwise.
- **Steps:** Individual tasks within a job that run commands or actions sequentially.
- **Runs:** Instances of workflow execution triggered by events, representing the complete run-through of a workflow.
- **Runners:** Servers that host the environment where the jobs are executed, available as GitHub-hosted or self-hosted options.
- **Marketplace:** A platform to find and share reusable actions, enhancing workflow capabilities with community-developed tools.



Workflow Contexts

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

You can access contexts using the expression syntax. **${{ <context> }}**

```yaml
steps:
  - name: Use Secret as Env Var
    run: echo "Secret Value: $MY_SECRET"
    env:
      MY_SECRET: ${{ secrets.MY_SECRET }}
```

- github — Information about the workflow run.
- env — Contains variables set in a workflow, job, or step.
- vars — Contains variables set at the repository, organization, or environment levels.
- job — Information about the currently running job.
- jobs — For reusable workflows only, contains outputs of jobs from the reusable workflow.
- steps — Information about the steps that have been run in the current job.
- Runner — information about the runner that is running the current job.
- secrets — Contains the names and values of secrets that are available to a workflow run.
- strategy — Information about the matrix execution strategy for the current job.
- matrix — Contains the matrix properties defined in the workflow that apply to the current job.
- needs — Contains the **outputs** of all jobs that are defined as a dependency of the current job.
- inputs — Contains the inputs of a reusable or manually triggered workflow.

Conditional Keywords for Steps

jobs.<job_id>.if conditional can be used to **prevent a job from running unless a condition is met**.

```yaml
name: example-workflow
on: [push]
jobs:
  production-deploy:
    if: github.repository == 'octo-org/octo-repo-prod'
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with:
          node-version: '14'
      - run: npm install -g bats
```

You must always use the **${{ }} expression syntax** or escape with '', "", or () when the expression starts with !, since ! is reserved notation in YAML format.

```yaml
if: ${{ ! startsWith(github.ref, 'refs/tags/') }}
```

Expressions

Functions

- contains(search, item) — eg contains('Hello world', 'llo’)
- startsWith(searchString, searchValue) — eg startsWith('Hello world', 'He’)
- endsWith(searchString, searchValue) — endsWith('Hello world', 'ld’)
- format(string, replaceValue0, replaceValue1, ..., replaceValueN) — format('Hello {0} {1} {2}', 'Mona', 'the', 'Octocat’)
- join(array, optionalSeparator) — join(github.event.issue.labels.*.name, ', ‘)
- toJSON(value) — toJSON(job)
- fromJSON(value) –
- hashFiles(path) — hashFiles('**/package-lock.json', '**/Gemfile.lock’)

Status Check Functions

- success()
- always()
- cancelled()
- failure()

Actions and Shell Commands

Runners

The Runner **determines the underlying compute and OS** that the the workflow will execute on.

The runner can be:

- **GitHub-hosted** — GitHub providers predefined runtime environments
  - Standard size
    - Linux: ubuntu-latest, ubuntu-22.04, ubuntu-20.04
    - Windows: windows-latest, windows-2022, windows-2019
    - MacOS: macos-latest, macos-14, macos-13, macos-12, macos-11
  - Larger Size
    - Only available for organizations and enterprises using the GitHub Team or GitHub Enterprise Cloud plans
    - More RAM, CPU, and disk space
    - Static IP addresses
    - The ability to group runners
    - Autoscaling to support concurrent workflows
- **Self-hosted** — external compute connected to GitHub using the GitHub Actions self-hosted runner application
  - create custom hardware configurations that meet your needs

Runners

Use the **runs-on** to specify the runner

```yaml
# specify a specific Github-self-hosted
runs-on: ubuntu-latest
runs-on: windows-latest
runs-on: macos-latest

# specify multiple possible runners
runs-on: [macos-14, macos-13, macos-12]

# specify Self-Hosted runner
runs-on: self-hosted
```

If you specify an **array of strings** or variables, your workflow will execute on any runner that matches all of the specified runs-on values.

Github Hosted vs Self-hosted Runners

	GitHub-Hosted Runners	Self-Hosted Runners
Setup and Maintenance	No setup required; fully managed by GitHub	Requires manual setup and maintenance
Cost	Free with limits on usage; charges for extra minutes	No cost for runner; infrastructure costs apply
Scalability	Automatically scales based on demand	Manually managed based on your infrastructure
Environment Control	Predefined environments with limited control	Full control over the environment
Operating Systems	Windows, Linux, and macOS	Any OS that can run the runner application
Security	Secure but runs in a shared environment	Potentially more secure, isolated in your infra.
Performance	Fixed performance capabilities	Can be tailored to your needs
Access to Internal Resources	Limited unless using self-hosted services	Direct access to internal networks and resources
Customization	Limited to available GitHub environments	Complete customization of the setup
Usage Limits	Subject to GitHub's usage limits and quotas	Determined by your own resources

Self-Hosted Runner

Self-hosted runners can be:

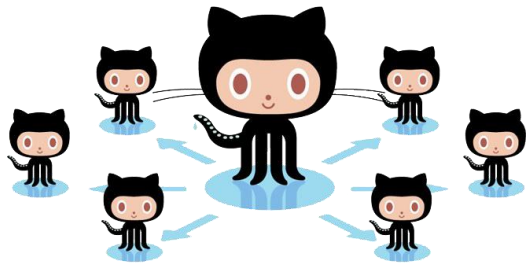
- Physical
- Virtual
- In a container
- On-premises
- In a cloud

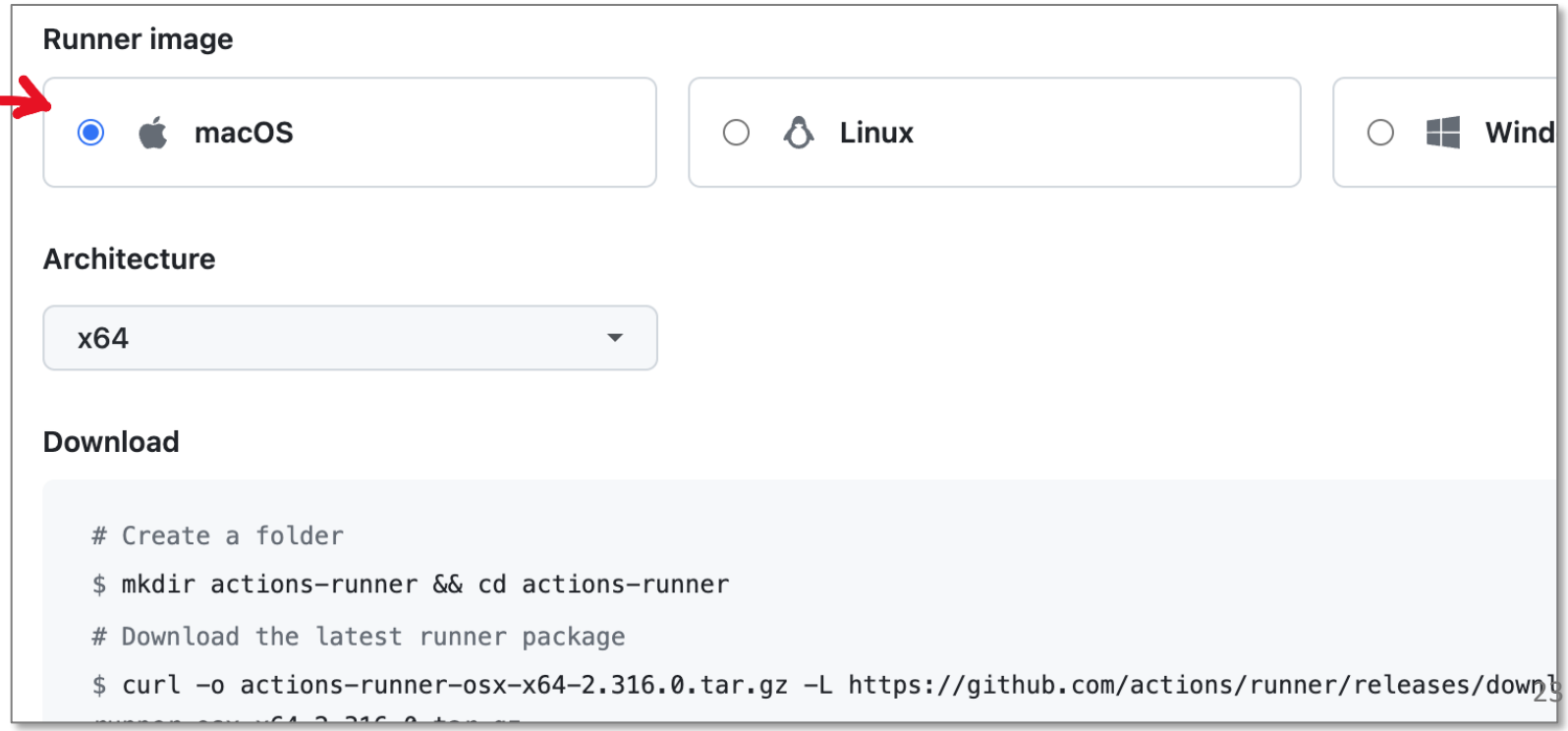
You can add self-hosted runners at various levels in the management hierarchy:

- **Repository-level** runners are dedicated to a single repository.
- **Organization-level** runners can process jobs for multiple repositories in an organization.
- **Enterprise-level** runners can be assigned to multiple organizations in an enterprise account.

To set up self-hosted, you need to add a runner and install the **GitHub Actions Runner** to connect the external compute to the self-hosted runner.

https://github.com/actions/runner





Workflow Commands

Actions can communicate with the runner machine to set environment variables, output values used by other actions, add debug messages to the output logs, and other tasks.

```yaml
name: Add Path Example

on: [push]

jobs:
  add-path:
    runs-on: ubuntu-latest
    steps:
      - name: Add directory to PATH
        run: |
          echo "$GITHUB_WORKSPACE/my_scripts" >> $GITHUB_PATH
      - name: Check new path
        run: |
          echo $PATH
```

Use the **run** command to run shell command echo

Workflow Commands

Set Env Vars
set an environment variable that will be available to subsequent steps in the job.

```yaml
steps:
  - name: Set environment variable
    run: echo "ACTION_ENV=production" >> $GITHUB_ENV
```

Adding to System Path
add a directory to the system PATH for subsequent steps in your job

```yaml
steps:
  - name: Add directory to PATH
    run: echo "/path/to/dir" >> $GITHUB_PATH
```

Setting Output Parameters
set outputs that can be used by other jobs in a multi-job workflow,

```yaml
steps:
  - name: Set output
    id: example_step
    run: echo "result=output_value" >> $GITHUB_OUTPUT

  - name: Use output
    run: echo "The output was ${{ steps.example_step.outputs.result }}"
```

Creating Debugging Messages
Create debug messages that appear in the logs of your actions, which are only visible if you enable step debug logging.

```yaml
steps:
  - name: Create debug message
    run: echo "::debug::This is a debug message"
```

Workflow Commands

Grouping Log Messages
make logs easier to read, you can group related messages together

```yaml
steps:
  - name: Group log messages
    run: |
      echo "::group::My Grouped Messages"
      echo "Message 1"
      echo "Message 2"
      echo "::endgroup::"
```

Masking Values in Logs
prevent sensitive information from appearing in logs, you can mask values

```yaml
steps:
  - name: Masking value
    run: echo "::add-mask::${{ secrets.SECRET_VALUE }}"
```

Stopping and Failing Actions
can force a workflow to stop and fail using the error command.

```yaml
steps:
  - name: Fail workflow
    run: echo "::error ::This is an error message"
```

Dependent Jobs

A workflow run is made up of one or more jobs, which run in parallel by default.
To run jobs sequentially, you can define dependencies on other jobs using:

- jobs.<job_id>.needs
- jobs.<job_id>.if

```yaml
jobs:
  job1:
  job2:
    needs: job1
  job3:
    needs: [job1, job2]
```

```yaml
jobs:
  job1:
  job2:
    needs: job1
  job3:
    if: ${{ always() }}
    needs: [job1, job2]
```

Encrypted Secrets

Encrypted Secrets are **variables that allow you to pass sensitive information** to your GitHub Actions Workflow

Secrets are accessed via the **secrets context** eg. ${{ secrets.MY_SECRET }}

Lower-level env vars override high-level env vars

Organization-Level Secrets
you can use access policies to control which repositories can use organization secrets
to share secrets between multiple repositories
Updating organization secrets propagates changes to all shared repos.

Repository-level Secrets
Secrets that are shared across all environments for a repo.

Environment-level Secrets
You can enable required reviewers to control access to the secrets

- Secret names can only contain alphanumeric characters and underscores. No spaces allowed eg. Hello_world123
- Names must not start with GITHUB_ prefix
- Names must not start with numbers
- Names are case-insensitive
- Names must be unique at the level they are created at

Encrypted Secrets – Accessing Secrets

Passing Secrets as Inputs

You can pass secrets as inputs by using the **secrets context**.

```yaml
name: Example Using Secret as Input

on: [push]

jobs:
  use-secret-as-input:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v2

      - name: Custom Action using Secret
        uses: example/action@v1
        with:
          api-key: ${{ secrets.API_KEY }}
```

Passing Secrets as Env Vars

You can also pass secrets to actions or scripts by **setting them as environment variables**.

```yaml
name: Example Using Secret as Env Var

on: [push]

jobs:
  use-secret-as-env:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v2

      - name: Run a script using Secret
        run: |
          echo "Using API key: $API_KEY"
        env:
          API_KEY: ${{ secrets.API_KEY }}
```

Encrypted Secrets – Settings Secrets

Settings Secrets at the **Repository-level**

```
# set a secret and prompt for secret value
gh secret set SECRET_NAME
# sets a secrets and sets values from a text file
gh secret set SECRET_NAME < secret.txt
```

Settings Secrets at the **Environment-level**

```
# set a secret and prompt for secret value
gh secret set --env ENV_NAME SECRET_NAME
# get list of secrets for an env
gh secret list --env ENV_NAME
```

Settings Secrets at the **Organization-level**

```
# login with admin:org scope to managed org secrets
gh auth login --scopes "admin:org"
# set a secret only for private repos and prompt for secret alue
gh secret set --org ORG_NAME SECRET_NAME
# set a secret for public, private and internal repos
gh secret set --org ORG_NAME SECRET_NAME --visibility all
# set a secret for specific repos
gh secret set --org ORG_NAME SECRET_NAME --repos REPO-NAME-1, REPO-NAME-2
# list secrets for the org
gh secret list --org ORG_NAME
```

Configuration Variables

Configuration Variables are **variables that allow you to non-sensitive information** to your GitHub Actions Workflow

Secrets are accessed via the **vars context** eg. ${{ vars.APP_ID_EXAMPLE }}

Lower level env vars override high level env vars

Organization-Level Secrets
you can use access policies to control which repositories can use organization secrets
share to secrets between multiple repositories
Updating organization secrets propagates changes to all shared repos.

Repository-level Secrets
Secrets that are shared across all environments for a repo.

Environment-level Secrets
You can enable required reviewers to control access to the secrets

- Secret names can only contain alphanumeric characters and underscores no spaces allowed eg. Hello_world123
- Names must not start with GITHUB_ prefix
- Names must not start with numbers
- Names are case-insensitive
- Names must be unique at the level they are created at

Configuration Variables - Setting Configuration Vars

```
# Add variable value for the current repository in an interactive prompt
gh variable set MYVARIABLE

# Read variable value from an environment variable
gh variable set MYVARIABLE --body "$ENV_VALUE"

# Read variable value from a file
gh variable set MYVARIABLE < myfile.txt

# Set variable for a deployment environment in the current repository
gh variable set MYVARIABLE --env myenvironment

# Set organization-level variable visible to both public and private
repositories
gh variable set MYVARIABLE --org myOrg --visibility all

# Set organization-level variable visible to specific repositories
gh variable set MYVARIABLE --org myOrg --repos repo1,repo2,repo3

# Set multiple variables imported from the ".env" file
gh variable set -f .env
```

Default Env Vars

The default environment variables that GitHub sets are available at every step in a workflow.

```
CI
```

```
GITHUB_ACTION
GITHUB_ACTION_PATH
GITHUB_ACTION_REPOSITORY
GITHUB_ACTIONS
```

```
GITHUB_ACTOR
GITHUB_ACTOR_ID
```

```
GITHUB_API_URL
GITHUB_BASE_REF
GITHUB_ENV
GITHUB_EVENT_NAME
GITHUB_EVENT_PATH
GITHUB_GRAPHQL_URL
GITHUB_HEAD_REF
GITHUB_JOB
GITHUB_OUTPUT
GITHUB_PATH
```

```
GITHUB_REF
GITHUB_REF_NAME
GITHUB_REF_PROTECTED
GITHUB_REF_TYPE
```

```
GITHUB_REPOSITORY
GITHUB_REPOSITORY_ID
GITHUB_REPOSITORY_OWNER
GITHUB_REPOSITORY_OWNER_ID
```

```
GITHUB_RETENTION_DAYS
```

```
GITHUB_RUN_ATTEMPT
GITHUB_RUN_ID
GITHUB_RUN_NUMBER
```

```
GITHUB_SERVER_URL
GITHUB_SHA
GITHUB_STEP_SUMMARY
GITHUB_TRIGGERING_ACTOR
```

```
GITHUB_WORKFLOW
GITHUB_WORKFLOW_REF
GITHUB_WORKFLOW_SHA
```

```
GITHUB_WORKSPACE
```

```
RUNNER_ARCH
RUNNER_DEBUG
RUNNER_NAME
RUNNER_OS
RUNNER_TEMP
RUNNER_TOOL_CACHE
```

How to use **default env vars**

```yaml
name: Example Workflow Using Default Env Variables

on: [push]

jobs:
  example_job:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v2

      - name: Print GitHub Environment Variables
        run: |
          echo "Repository Name: $GITHUB_REPOSITORY"
          echo "Workflow: $GITHUB_WORKFLOW"
          echo "Action: $GITHUB_ACTION"
          echo "Actor: $GITHUB_ACTOR"
```

Set Custom Env Vars

You can define Environment Variables inline within your GitHub Actions workflow.

Workflow-level
Set env vars of the entire workflow

Job-level
Set env vars for the entire job

Step-level
Set env vars for the step

```yaml
name: Greeting on variable day

on:
  workflow_dispatch

env:
  DAY_OF_WEEK: Monday

jobs:
  greeting_job:
    runs-on: ubuntu-latest
    env:
      Greeting: Hello
    steps:
      - name: "Say Hello Mona it's Monday"
        run: echo "$Greeting $First_Name. Today is $DAY_OF_WEEK!"
        env:
          First_Name: Mona
```

Notice we are accessing env vars **the native way for Linux**

Set Custom Env Vars

We can also access env vars via the **env context**

```yaml
name: Conditional env variable

on: workflow_dispatch

env:
  DAY_OF_WEEK: Monday

jobs:
  greeting_job:
    runs-on: ubuntu-latest
    env:
      Greeting: Hello
    steps:
      - name: "Say Hello Mona it's Monday"
        if: ${{ env.DAY_OF_WEEK == 'Monday' }}
        run: echo "$Greeting $First_Name. Today is $DAY_OF_WEEK!"
        env:
          First_Name: Mona
```

Set Env Vars with Workflow Commands

In GitHub Actions, you can dynamically set env vars during the execution of your workflows using the **$GITHUB_ENV** special workflow command.

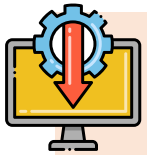
This is useful for passing values between steps, dynamically adjusting behavior based on runtime data, or configuring tools and scripts executed by your workflow.

```yaml
name: Set Environment Variables Example

on: [push]

jobs:
  setup-and-use-env:
    runs-on: ubuntu-latest
    steps:
      - name: Set dynamic environment variable
        run: |
          # Using workflow command to set an environment variable
          echo "DYNAMIC_VAR=Hello from GitHub Actions" >> $GITHUB_ENV

      - name: Use the environment variable
        run: |
          # Using the environment variable in a subsequent step
          echo "The value of DYNAMIC_VAR is: $DYNAMIC_VAR"
```

Anything placed into $GITHUB_ENV will be accessible anywhere in your workflow

GITHUB_TOKEN Secret

At the start of each workflow job, GitHub automatically creates a unique GITHUB_TOKEN secret to use in your workflow. You can use the GITHUB_TOKEN to authenticate in the workflow job.



When you enable GitHub Actions, GitHub installs a GitHub App on your repository. The GITHUB_TOKEN secret is a GitHub App installation access token.

```yaml
name: Open new issue
on: workflow_dispatch

jobs:
  open-issue:
    runs-on: ubuntu-latest
    permissions:
      contents: read
      issues: write
    steps:
      - run: |
          gh issue --repo ${{ github.repository }} \
            create --title "Issue title" --body "Issue body"
        env:
          GH_TOKEN: ${{ secrets.GITHUB_TOKEN }}
```

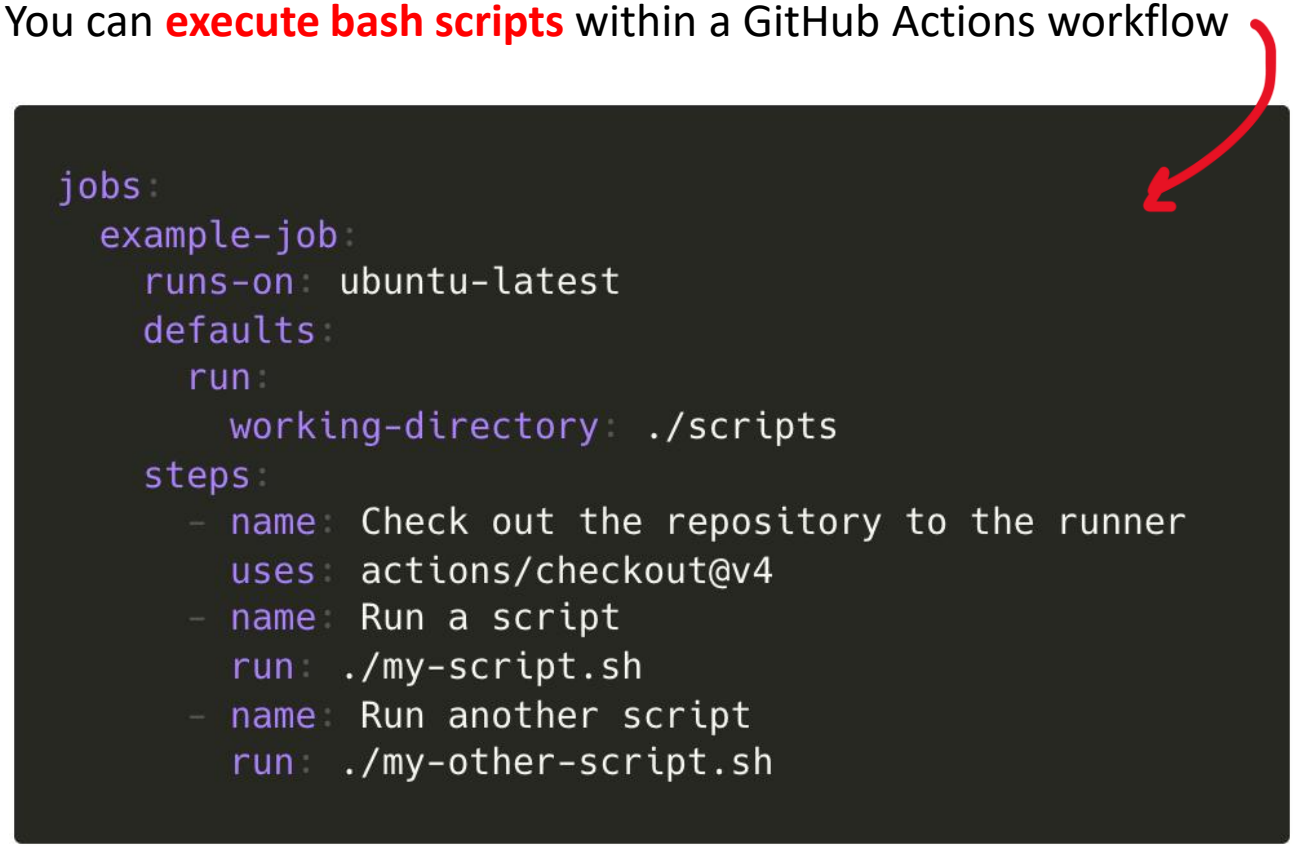
GITHUB_TOKEN Secret

Example of Using GITHUB_TOKEN using the **REST API**

```yaml
name: Create issue on commit

on: [ push ]

jobs:
  create_issue:
    runs-on: ubuntu-latest
    permissions:
      issues: write
    steps:
      - name: Create issue using REST API
        run: |
          curl --request POST \
          --url https://api.github.com/repos/${{ github.repository }}/issues \
          --header 'authorization: Bearer ${{ secrets.GITHUB_TOKEN }}' \
          --header 'content-type: application/json' \
          --data '{
            "title": "Automated issue for commit: ${{ github.sha }}",
            "body": "Automatically created by the GitHub Action workflow **${{
github.workflow }}**. \n\n The commit hash was: _${{ github.sha }}_."
            }' \
          --fail
```

Add Script to Workflow

You can **execute bash scripts** within a GitHub Actions workflow



```yaml
jobs:
  example-job:
    runs-on: ubuntu-latest
    defaults:
      run:
        working-directory: ./scripts
    steps:
      - name: Check out the repository to the runner
        uses: actions/checkout@v4
      - name: Run a script
        run: ./my-script.sh
      - name: Run another script
        run: ./my-other-script.sh
```

Publish GitHub Package using Workflow

You can use a workflow to build a GitHub Package

```yaml
name: Create and publish a Docker image
on:
  push:
    branches: ['release']

env:
  REGISTRY: ghcr.io
  IMAGE_NAME: ${{ github.repository }}

jobs:
  build-and-push-image:
    runs-on: ubuntu-latest
    permissions:
      contents: read
      packages: write
    steps:
    # ....
```

```yaml
steps:
  - name: Checkout repository
    uses: actions/checkout@v4
  - name: Log in to the Container registry
    uses: docker/login-action@65b78e6e13532edd9afa3aa52ac7964289d1a9c1
    with:
      registry: ${{ env.REGISTRY }}
      username: ${{ github.actor }}
      password: ${{ secrets.GITHUB_TOKEN }}
  - name: Extract metadata (tags, labels) for Docker
    id: meta
    uses: docker/metadata-action@9ec57ed1fcdbf14dcef7dfbe97b2010124a938b7
    with:
      images: ${{ env.REGISTRY }}/${{ env.IMAGE_NAME }}
  - name: Build and push Docker image
    uses: docker/build-push-action@f2a1d5e99d037542a71f64918e516c093c6f3fc4
    with:
      context: .
      push: true
      tags: ${{ steps.meta.outputs.tags }}
      labels: ${{ steps.meta.outputs.labels }}
```

Publish Docker Hub Registry using Workflow

You can use a workflow to build and publish to Docker Hub Registry

```yaml
name: Publish Docker image

on:
  release:
    types: [published]

jobs:
  push_to_registry:
    name: Push Docker image to Docker Hub
    runs-on: ubuntu-latest
    steps:
    # ...
```

```yaml
steps:
  - name: Check out the repo
    uses: actions/checkout@v4

  - name: Log in to Docker Hub
    uses: docker/login-action@f4ef78c080cd8ba55a85445d5b36e214a81df20a
    with:
      username: ${{ secrets.DOCKER_USERNAME }}
      password: ${{ secrets.DOCKER_PASSWORD }}

  - name: Extract metadata (tags, labels) for Docker
    id: meta
    uses: docker/metadata-action@9ec57ed1fcdbf14dcef7dfbe97b2010124a938b7
    with:
      images: my-docker-hub-namespace/my-docker-hub-repository

  - name: Build and push Docker image
    uses: docker/build-push-action@3b5e8027fcad23fda98b2e3ac259d8d67585f671
    with:
      context: .
      file: ./Dockerfile
      push: true
      tags: ${{ steps.meta.outputs.tags }}
      labels: ${{ steps.meta.outputs.labels }}
```

Publish GitHub Container Registry using Workflow

Pushing container to the GitHub Container Registry

```yaml
name: Publish Docker Image

on: [push]

jobs:
  build-and-push:
    runs-on: ubuntu-latest
    steps:
    - name: Check out the repo
      uses: actions/checkout@v2
    - name: Log in to GitHub Container Registry
      uses: docker/login-action@v1
      with:
        registry: ghcr.io
        username: ${{ github.actor }}
        password: ${{ secrets.CR_PAT }}  # or use GITHUB_TOKEN
    - name: Set up Docker Buildx
      uses: docker/setup-buildx-action@v1
    - name: Build and push Docker image
      uses: docker/build-push-action@v2
      with:
        context: .
        file: ./Dockerfile
        push: true
        tags: ghcr.io/${{ github.repository_owner }}/my-image:latest
    - name: Verify the image was pushed
      run: docker pull ghcr.io/${{ github.repository_owner }}/my-image:latest
```

Service Containers

Service containers are **Docker containers** used host services that you might need to test or operate your application in a workflow.

- You can **configure service containers** for each job in a workflow.
- GitHub creates a fresh Docker container for each service configured in the workflow and destroys the service container when the job completes.
- Steps in a job can communicate with all service containers that are part of the same job.
- However, you cannot create and use service containers inside a composite action.

If your workflows use Docker container actions, job containers, or service containers, then you must use a Linux runner:
If you are using GitHub-hosted runners, you must use an Ubuntu runner.
If you are using self-hosted runners, you must use a Linux machine as your runner and Docker must be installed.

You can configure jobs in a workflow to run directly on a runner machine or in a Docker container.

Running jobs in a container	Running jobs on the runner machine
Using Docker **bridge** mode. Access the container via its hostname (eg. redis)	Using Docker **host** mode. Access using localhost:<port> or 127.0.0.1:<port>

You must map ports since containers are not directly exposed

Service Containers

You can use the **services** keyword to create service containers that are part of a job in your workflow.

```yaml
name: Redis container example
on: push

jobs:
  runner-job:
    runs-on: ubuntu-latest
    services:
      redis:
        image: redis
        ports:
          - 6379:6379
```

Notice the runner job **specifies no container.**

For a runner job you need to **map the ports**.

```yaml
name: Redis container example
on: push

jobs:
  container-job:
    runs-on: ubuntu-latest
    container: node:16-bullseye
    services:
      redis:
        image: redis
```

Notice the container job **specifies container**

Service Containers

You can specify **credentials** for your service containers in case you need to authenticate with an image registry.

```yaml
jobs:
  build:
    services:
      redis:
        # Docker Hub image
        image: redis
        ports:
          - 6379:6379
        credentials:
          username: ${{ secrets.dockerhub_username }}
          password: ${{ secrets.dockerhub_password }}
      db:
        # Private registry image
        image:  ghcr.io/octocat/testdb:latest
        credentials:
          username: ${{ github.repository_owner }}
          password: ${{ secrets.ghcr_password }}
```

Service Containers

Running PostgreSQL as a service container

```yaml
jobs:
  container-job:
    runs-on: ubuntu-latest
    container: node:10.18-jessie
    services:
      postgres:
        image: postgres
        env:
          POSTGRES_PASSWORD: postgres
        options: >-
          --health-cmd pg_isready
          --health-interval 10s
          --health-timeout 5s
          --health-retries 5
        ports:
          - 5432:5432
```

Connecting the postgres service container in a step.

```yaml
steps:
  - name: Check out repository code
    uses: actions/checkout@v4
  - name: Install dependencies
    run: npm ci
  - name: Connect to PostgreSQL
    run: node client.js
    env:
      POSTGRES_HOST: postgres
      POSTGRES_PORT: 5432
```

```javascript
const { Client } = require('pg');

const pgclient = new Client({
    host: process.env.POSTGRES_HOST,
    port: process.env.POSTGRES_PORT,
    user: 'postgres',
    password: 'postgres',
    database: 'postgres'
});
pgclient.connect();
pgclient.query('SELECT * FROM student', (err, res) => {
    if (err) throw err
    console.log(err, res.rows) // Print the data in student table
    pgclient.end()
});
```

Routing Workflow to Runner

A self-hosted runner automatically receives certain labels when it is added to GitHub Actions.

Default Labels

These are used to indicate its operating system and hardware platform:

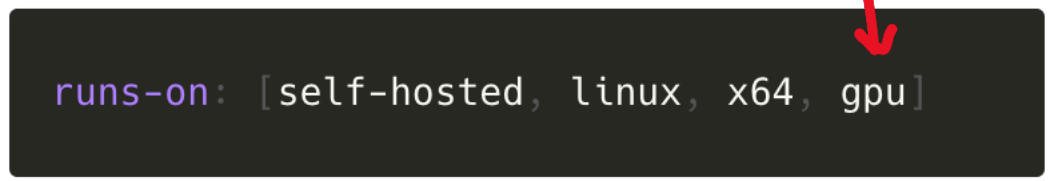
- **self-hosted:** Default label applied to self-hosted runners.
- **linux, windows, or macOS:** Applied depending on operating system.
- **x64, ARM, or ARM64:** Applied depending on hardware architecture.

```
runs-on: [self-hosted, linux, ARM64]
```

Custom Labels

- You can create custom labels and assign them to your self-hosted runners at any time.
- Custom labels let you send jobs to particular types of self-hosted runners, based on how they're labeled.
- These labels operate **cumulatively**, so a self-hosted runner must have all four labels to be eligible to process the job.

GPU is the **custom label**



Routing Workflow to Runner

Runner groups are used to **collect sets of runners** and create a security boundary around them.

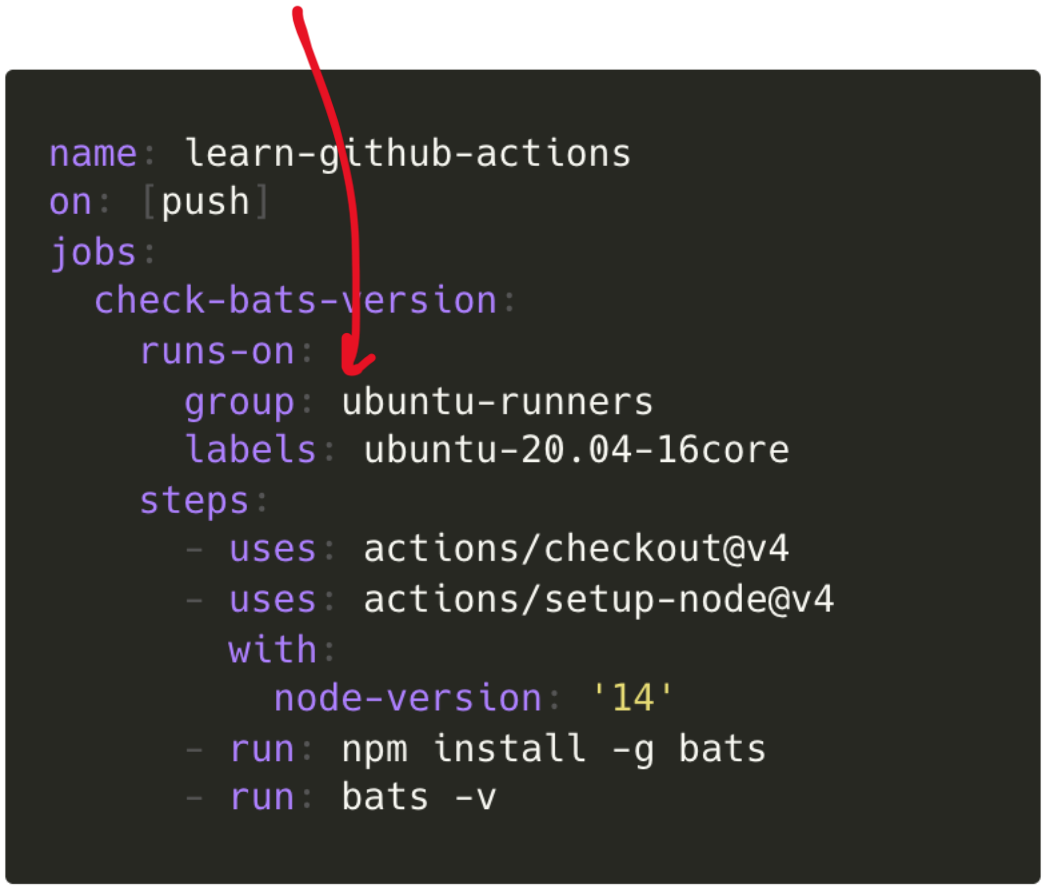
Enterprise accounts, organizations owned by enterprise accounts, and organizations using GitHub Team can create and manage additional runner groups.

runs-on key sends the job to any available runner in the ubuntu-runners **group**

```yaml
name: learn-github-actions
on: [push]
jobs:
  check-bats-version:
    runs-on:
      group: ubuntu-runners
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with:
          node-version: '14'
      - run: npm install -g bats
      - run: bats -v
```

Routing Workflow to Runner

Labels and Groups can be **combined** when routing workflows to the appropriate runner.



```yaml
name: learn-github-actions
on: [push]
jobs:
  check-bats-version:
    runs-on:
      group: ubuntu-runners
      labels: ubuntu-20.04-16core
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with:
          node-version: '14'
      - run: npm install -g bats
      - run: bats -v
```

CodeQL Step

CodeQL into a GitHub Actions workflow can automate the process of code analysis, allowing you to find and fix security issues before they are exploited.

```yaml
name: "CodeQL"

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]
  schedule:
    - cron: '0 14 * * 1'
jobs:
  analyze:
    name: Analyze
    runs-on: ubuntu-latest
    strategy:
      fail-fast: false
      matrix:
        language: [ 'java', 'javascript', 'python' ]
    steps:
    - name: Checkout repository
      uses: actions/checkout@v2

    - name: Initialize CodeQL
      uses: github/codeql-action/init@v1
      with:
        languages: ${{ matrix.language }}
    - name: Perform CodeQL Analysis
      uses: github/codeql-action/analyze@v1
```

Publish Component as GitHub Release

Publishing Components as GitHub Release

```yaml
name: Release Workflow
on:
  push:
    tags:
      - 'v*'
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
    # ...
```

```yaml
steps:
  - name: Checkout code
    uses: actions/checkout@v2

  - name: Build project
    run: |
      echo "Build your project here"
      # Example: gcc -o output_binary source_code.c

  - name: Create Release
    id: create_release
    uses: actions/create-release@v1
    env:
      GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
    with:
      tag_name: ${{ github.ref }}
      release_name: Release ${{ github.ref }}
      draft: false
      prerelease: false

  - name: Upload Release Asset
    uses: actions/upload-release-asset@v1
    env:
      GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
    with:
      upload_url: ${{ steps.create_release.outputs.upload_url }}
      asset_path: ./path/to/your/output_binary
      asset_name: output_binary_name
      asset_content_type: application/octet-stream
```

Deploy Release to Cloud Provider

You can deploy releases to specific CSP:

- Amazon Elastic Container Service (ECS)
- Google Kubernetes Engine
- Azure App Services, Azure Kubernetes Service (EKS), **Azure Static Web Apps**

```yaml
name: Deploy web app to Azure Static Web Apps

env:
  APP_LOCATION: "/"
  API_LOCATION: "api"
  OUTPUT_LOCATION: "build"

on:
  push:
    branches:
      - main
  pull_request:
    types: [opened, synchronize, reopened, closed]
    branches:
      - main
permissions:
  issues: write
  contents: read
  pull-requests: write

jobs:
  # ....
```

```yaml
jobs:
  build_and_deploy:
    if: github.event_name == 'push' || (github.event_name == 'pull_request' && github.event.action != 'closed')
    runs-on: ubuntu-latest
    name: Build and Deploy
    steps:
      - uses: actions/checkout@v4
        with:
          submodules: true
      - name: Build And Deploy
        uses: Azure/static-web-apps-deploy@1a947af9992250f3bc2e68ad0754c0b0c11566c9
        with:
          azure_static_web_apps_api_token: ${{ secrets.AZURE_STATIC_WEB_APPS_API_TOKEN }}
          repo_token: ${{ secrets.GITHUB_TOKEN }}
          action: "upload"
          app_location: ${{ env.APP_LOCATION }}
          api_location: ${{ env.API_LOCATION }}
          output_location: ${{ env.OUTPUT_LOCATION }}

  close_pull_request:
    if: github.event_name == 'pull_request' && github.event.action == 'closed'
    runs-on: ubuntu-latest
    name: Close Pull Request
    steps:
      - name: Close Pull Request
        uses: Azure/static-web-apps-deploy@1a947af9992250f3bc2e68ad0754c0b0c11566c9
        with:
          azure_static_web_apps_api_token: ${{ secrets.AZURE_STATIC_WEB_APPS_API_TOKEN }}
          action: "close"
```

Caching Package and Dependency Files

To make your workflows faster and more efficient, you can create and use caches for dependencies and other commonly reused files.

Workflow runs often reuse the same outputs or downloaded dependencies from one run to another. **Caching package and dependency management** can greatly improve performance.

```yaml
# caching ruby package manager
- uses: ruby/setup-ruby@v1
  with:
    bundler-cache: true
```

The following package manages and their related Action

- npm, Yarn, pnpm — setup-node
- pip, pipenv, Poetry — setup-python
- Gradle, Maven — setup-java
- RubyGems — **setup-ruby**
- Go go.sum — setup-go

Caching Job Dependencies and Build Output

Cache action allows caching dependencies and build outputs to improve workflow execution time.

Restoring and saving cache using a single action

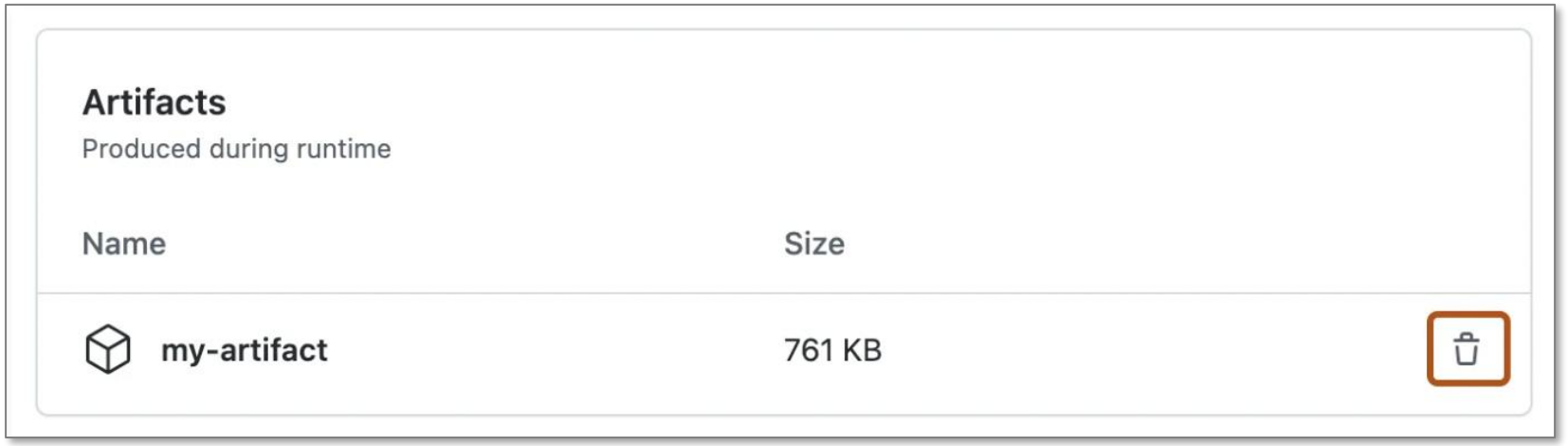
There is also the
- Save Action
- Restore Action

If you want to perform these operations separately

```yaml
name: Caching Primes

on: push

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v3

    - name: Cache Primes
      id: cache-primes
      uses: actions/cache@v4
      with:
        path: prime-numbers
        key: ${{ runner.os }}-primes

    - name: Generate Prime Numbers
      if: steps.cache-primes.outputs.cache-hit != 'true'
      run: /generate-primes.sh -d prime-numbers

    - name: Use Prime Numbers
      run: /primes.sh -d prime-numbers
```

Remove Workflow Artifact from Github

To remove workflow artifacts you need to delete them via the UI



Once you delete an artifact, it cannot be restored.

By default, GitHub stores build logs and artifacts for 90 days, and this retention period can be customized

Workflow Status Badge

You can display a status badge in your repository to indicate the status of your workflows.


```
![example workflow](https://github.com/github/docs/actions/workflows/main.yml/badge.svg)
```

You use a special URL provide your workfile file name and you place this into your Readme.md file.

You can specify a specific branch or based on an event for the badge

```
...main.yml/badge.svg?branch=feature-1
...main.yml/badge.svg?event=push
```

Workflow badges in a private repository are not accessible externally, so you won't be able to embed them or link to them from an external site.

Env Protections

You can configure **environments** with protection rules and references an environment in your Workflow Job

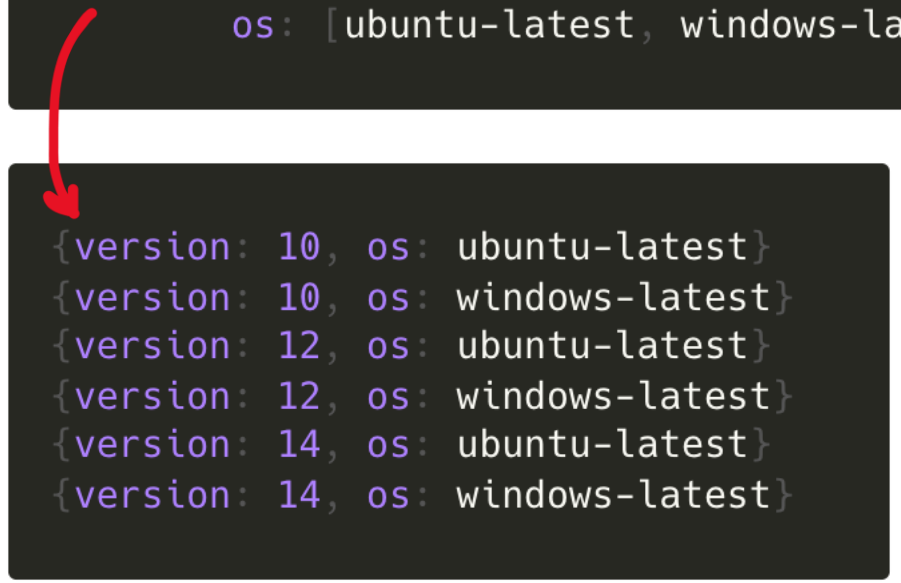
- Each job in a workflow can reference a single environment.
- Any protection rules configured for the environment must pass before a job referencing the environment is sent to a runner.
- The job can access the environment's secrets only after the job is sent to a runner.

```yaml
name: Deployment

on:
  push:
    branches:
      - main

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

Job Matrix Configuration

A matrix strategy lets you use variables in a single job definition to automatically create multiple job runs that are based on the combinations of the variables.

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



This will result in 6 job runs

Workflow Approval Gates

Can only be done via the UI, use blog article to recreate and take your own screenshots.

Identifying Triggering Event

Identifying the event that triggered a GitHub Actions workflow involves understanding the various events that GitHub can use to initiate workflows. Each type of event has specific characteristics and can affect the repository, issues, or pull requests in distinct ways. Here's how you can determine the triggering event based on its effects and how to use this information in your GitHub Actions workflo

?What to do here?

Common GitHub Actions Trigger Events

1. **Push**
   - **Effect**: Commits are added to a branch in the repository.
   - **Indicators**: Look for new commits or tags being pushed to specific branches.
   - **Workflow Example**:

yaml Copy code

```yaml
on:
  push:
    branches:
      - main
    tags:
      - 'v*'
```

2. **Pull Request**
   - **Effect**: Creation, modification, or closing of pull requests.
   - **Indicators**: Changes in pull request status, new commits in a PR, or edits to the PR's

Configuration File Workflow Effect

Failed Workflow Runs

Workflow Logs from UI

Workflow Logs from REST API

Step Debug Logging

Default Env Var in Workflow

SEEMS LIKE REPEAT for Default Env Var

Passing Custom Env Var

SEEMS LIKE REPEAT for Custom Env Var

Locate Workflow in Repo

Seems pretty obvious where…

Disabling vs Deleting Workflow

Disabling a Workflow	Deleting a Workflow
Temporarily stop a workflow from being triggered.	Permanently remove the workflow from the repository.
Easily reversible. You can re-enable the workflow at any time.	Not directly reversible; you would need to recreate the file or restore it from version history.
When updates or maintenance are needed, or if the workflow is triggering too often or unnecessarily.	When the workflow is no longer needed, or you're cleaning up the repository.
You can disable the Workflow using the GitHub CLI	You have to the GitHub UI to delete a Wokflow

Download Workflow Artifacts from UI

<Get Screenshots when you do the lab>

Sharing Data between jobs

If your job generates files that you want to share with another job in the same workflow, or if you want to save the files for later reference, you can store them in GitHub as artifacts.

Uploading a file to artifact

```yaml
jobs:
  example-job:
    name: Save output
    runs-on: ubuntu-latest
    steps:
      - shell: bash
        run: |
          expr 1 + 1 > output.log
      - name: Upload output file
        uses: actions/upload-artifact@v4
        with:
          name: output-log-file
          path: output.log
```

Download an artifact to use locally

```yaml
jobs:
  example-job:
    runs-on: ubuntu-latest
    steps:
      - name: Download a single artifact
        uses: actions/download-artifact@v4
        with:
          name: output-log-file
```

Organization Templated Workflow

No clue what this is supposed to be

Trustworthy Actions

GitHub Actions about how to know if an action is safe to use

Action Type for Actions

Types of Actions	Linux	MacOs	Windows
Docker container	Yes	No	
JavaScript	Yes		
Composite Actions	Yes		

- Docker container runs your action code in a docker container
- JavaScript runs directly on the runner host OS
- A composite action allows you to combine multiple workflow steps within one action.

Actions require a metadata file to define the inputs, outputs and main entrypoint for your action.

Inputs and Outputs for Actions

specify data that the action expects to use during runtime.

```yaml
inputs:
  num-octocats:
    description: 'Number of Octocats'
    required: false
    default: '1'
  octocat-eye-color:
    description: 'Eye color of the Octocats'
    required: true
```

allow you to declare data that an action sets. Actions that run later in a workflow can use the output data set in previously run actions.

```yaml
outputs:
  sum: # id of the output
    description: 'The sum of the inputs'
```

Using determines whether this is a JavaScript action, a composite action, or a Docker container action and how the action is executed.

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

Action Versions

GitHub Recommendations for release management
practices when you plan to release an action for public use

General Recommendations

- patch version to include necessary critical fixes and security patches, while still remaining compatible with their existing workflows.
- consider releasing a new major version whenever your changes affect compatibility.
- Tell your users to specify a major version when using your action, and only direct them to a more specific version if they encounter issues.

Tagging Recommendations

- Create and validate a release on a release branch (such as release/v1) before creating the release tag (for example, v1.0.2).
- Create a release using semantic versioning
- Move the major version tag (such as v1, v2) to point to the Git ref of the current release
- Introduce a new major version tag (v2) for changes that will break existing workflows. For example, changing an action's inputs would be a breaking change.
- Major versions can be initially released with a beta tag to indicate their status, for example, v2-beta. The -beta tag can then be removed when ready.

```yaml
# specific major
steps:
    - uses: actions/javascript-action@v1

# specific major, minor and patch
steps:
    - uses: actions/javascript-action@v1.0.1

# specific branch
steps:
    - uses: actions/javascript-action@v1-beta
```

Action Type Scenarios

Junk slide maybe?

Troubleshooting Javascript Actions

Lab…

Troubleshooting Docker Actions

Lab…

Files and Directories for JavaScript Actions

Directory Structure
Your GitHub repository should contain a root directory named after your action or something meaningful. Inside, you'll organize your source code and configuration files.

```
/your-action-name
  /node_modules
  action.yml
  index.js
  package.json
  README.md
```

```yaml
name: 'My JavaScript Action'
description: 'A description of your action'
inputs:
  my-input:
    description: 'Input to use in the action'
    required: true
    default: 'default input value'
outputs:
  my-output:
    description: 'Output from the action'
runs:
  using: 'node12'
  main: 'index.js'
```

action.yml

Essential Files

- action.yml/yaml: Defines the action's inputs, outputs, and main entry point.
- index.js: The main JavaScript file that runs when the action is executed.
- package.json: Manages project dependencies and scripts.
- README.md: Documentation on how to use the action.

Files and Directories for Docker Container Actions

```
/your-docker-action-name
  Dockerfile
  entrypoint.sh
  action.yml
  README.md
```

```yaml
name: 'My Docker Action'
description: 'A description of what your action does'
inputs:
  my-input:
    description: 'Input to use in the action'
    required: true
outputs:
  my-output:
    description: 'Outputs of the action'
runs:
  using: 'docker'
  image: 'Dockerfile'
  args:
    - ${{ inputs.my-input }}
```

action.yml

Exit Codes for GitHub Actions

You can use exit codes to set the status of an action

Bash Script within your Acction

```bash
#!/bin/bash

# Example command
make build
exit_status=$?

if [ $exit_status -ne 0 ]; then
  echo "::error ::Build failed with exit code $exit_status"
  exit $exit_status
fi

echo "::set-output name=status::success"
exit 0
```

Entrypoint.sh for Docker Container

```bash
#!/bin/sh -l

# Setting environment variable
echo "API_KEY=abc123" >> $GITHUB_ENV

# Running a script and checking the exit code
./run-tests.sh
if [ $? -ne 0 ]; then
  echo "::error ::Tests failed"
  exit 1
fi

# Setting output parameters
echo "test-result=passed" >> $GITHUB_OUTPUT

# Final success message
echo "Action completed successfully"
```

Action Distribution Models

Custom Action Distributions

Release Strategy for Actions

Publish Actions to GitHub Marketplace

Reuse Template for Actions and Workflow

Workflow Templates is an enterprise feature that allow you to create reusable templates that other enterprises members can use.

User's with write access to the enterprise **.github** repo can create workflows from workflow templates

You need create a workflow file and metadata file in a public .github repo within a directory called workflow templates. The Workflow YML file and Metadata JSON file need to have the same name.

```yaml
name: Octo Organization CI

on:
  push:
    branches: [ $default-branch ]
  pull_request:
    branches: [ $default-branch ]
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2

      - name: Run a one-line script
        run: echo Hello from Octo Organization
```

```json
{
  "name": "Octo Org Workflow",
  "description": "Octo Org CI workflow template.",
  "iconName": "example-icon",
  "categories": [
      "Go"
  ],
  "filePatterns": [
      "package.json$",
      "^Dockerfile",
      ".*\\.md$"
  ]
}
```

Organizational Use Policies

Reusable Components

Distribute Actions for Enterprise

Control Access to Actions for Enterprise

Configuration GitHub Allow List

Runners for Support Workloads

Github-hosted vs Self-hosted Runners

Feels like repeat.

Configure Self-hosted Runners for Enterprise

Self-hosted runners for GitHub Enterprise have additional configuration options

Proxy Servers
The following environment variables are available.

- https_proxy — URLs for HTTP Traffic to proxy
- http_proxy — URLs for HTTP Traffic to proxy
- no_proxy — URLs that should not be used as a proxy

Additional URLS are comma-separated

IP Allowlists
You must add the IP address or range of your self-hosted runners to the IP allowlist of communication to work.

Manage Self-hosted Runners for Groups

Encrypted Secrets for Enterprise

Encrypted Secrets within Actions and Workflows

Org-level Encrypted Secrets

Repo-level Encrypted Secrets