Article version: Free, Pro, and Team

Workflow syntax for GitHub Actions

A workflow is a configurable automated process made up of one or more jobs. You must create a YAML file to define your workflow configuration.

GitHub Actions is available with GitHub Free, GitHub Pro, GitHub Free for organizations, GitHub Team, GitHub Enterprise Cloud, GitHub Enterprise Server, and GitHub One. GitHub Actions is not available for private repositories owned by accounts using legacy per-repository plans. For more information, see "GitHub's products."

In this article

About YAML syntax for workflows

Workflow files use YAML syntax, and must have either a `.yml` or `.yaml` file extension. If you're new to YAML and want to learn more, see "Learn YAML in five minutes."

You must store workflow files in the `.github/workflows` directory of your repository.

`name`

The name of your workflow. GitHub displays the names of your workflows on your repository's actions page. If you omit `name`, GitHub sets it to the workflow file path relative to the root of the repository.

`on`

Required The name of the GitHub event that triggers the workflow. You can provide a single event `string`, `array` of events, `array` of event `types`, or an event configuration `map` that schedules a workflow or restricts the execution of a workflow to specific files, tags, or branch changes. For a list of available events, see "Events that

trigger workflows."

Example using a single event

```yaml
# Triggered when code is pushed to any branch in a repository
on: push
```

Example using a list of events

```yaml
# Triggers the workflow on push or pull request events
on: [push, pull_request]
```

Example using multiple events with activity types or configuration

If you need to specify activity types or configuration for an event, you must configure each event separately. You must append a colon (:) to all events, including events without configuration.

```yaml
on:
  # Trigger the workflow on push or pull request,
  # but only for the main branch
  push:
    branches:
      - main
  pull_request:
    branches:
      - main
  # Also trigger on page_build, as well as release created events
  page_build:
  release:
    types: # This configuration does not affect the page_build event above
      - created
```

on.<event_name>.types

Selects the types of activity that will trigger a workflow run. Most GitHub events are triggered by more than one type of activity. For example, the event for the release resource is triggered when a release is `published` , `unpublished` , `created` ,

`edited`, `deleted`, or `prereleased`. The `types` keyword enables you to narrow down activity that causes the workflow to run. When only one activity type triggers a webhook event, the `types` keyword is unnecessary.

You can use an array of event `types`. For more information about each event and their activity types, see "Events that trigger workflows."

```
# Trigger the workflow on pull request activity
on:
  release:
    # Only use the types keyword to narrow down the activity types that wi
    types: [published, created, edited]
```

on.<push|pull_request>.<branches|tags>

When using the `push` and `pull_request` events, you can configure a workflow to run on specific branches or tags. For a `pull_request` event, only branches and tags on the base are evaluated. If you define only `tags` or only `branches`, the workflow won't run for events affecting the undefined Git ref.

The `branches`, `branches-ignore`, `tags`, and `tags-ignore` keywords accept glob patterns that use the `*` and `**` wildcard characters to match more than one branch or tag name. For more information, see the "Filter pattern cheat sheet."

Example including branches and tags

The patterns defined in `branches` and `tags` are evaluated against the Git ref's name. For example, defining the pattern `mona/octocat` in `branches` will match the `refs/heads/mona/octocat` Git ref. The pattern `releases/**` will match the `refs/heads/releases/10` Git ref.

```
on:
  push:
    # Sequence of patterns matched against refs/heads
    branches:
      # Push events on main branch
      - main
      # Push events to branches matching refs/heads/mona/octocat
      - 'mona/octocat'
      # Push events to branches matching refs/heads/releases/10
```

```
      - 'releases/**'
    # Sequence of patterns matched against refs/tags
    tags:
      - v1             # Push events to v1 tag
      - v1.*           # Push events to v1.0, v1.1, and v1.9 tags
```

Example ignoring branches and tags

Anytime a pattern matches the `branches-ignore` or `tags-ignore` pattern, the workflow will not run. The patterns defined in `branches-ignore` and `tags-ignore` are evaluated against the Git ref's name. For example, defining the pattern `mona/octocat` in `branches` will match the `refs/heads/mona/octocat` Git ref. The pattern `releases/**-alpha` in `branches` will match the `refs/releases/beta/3-alpha` Git ref.

```
on:
  push:
    # Sequence of patterns matched against refs/heads
    branches-ignore:
      # Push events to branches matching refs/heads/mona/octocat
      - 'mona/octocat'
      # Push events to branches matching refs/heads/releases/beta/3-alpha
      - 'releases/**-alpha'
    # Sequence of patterns matched against refs/tags
    tags-ignore:
      - v1.*           # Push events to tags v1.0, v1.1, and v1.9
```

Excluding branches and tags

You can use two types of filters to prevent a workflow from running on pushes and pull requests to tags and branches.

- `branches` or `branches-ignore` - You cannot use both the `branches` and `branches-ignore` filters for the same event in a workflow. Use the `branches` filter when you need to filter branches for positive matches and exclude branches. Use the `branches-ignore` filter when you only need to exclude branch names.
- `tags` or `tags-ignore` - You cannot use both the `tags` and `tags-ignore` filters for the same event in a workflow. Use the `tags` filter when you need to filter tags for positive matches and exclude tags. Use the `tags-ignore` filter when you only need to exclude tag names.

Example using positive and negative patterns

You can exclude `tags` and `branches` using the `!` character. The order that you define patterns matters.

- A matching negative pattern (prefixed with `!`) after a positive match will exclude the Git ref.
- A matching positive pattern after a negative match will include the Git ref again.

The following workflow will run on pushes to `releases/10` or `releases/beta/mona`, but not on `releases/10-alpha` or `releases/beta/3-alpha` because the negative pattern `!releases/**-alpha` follows the positive pattern.

```
on:
  push:
    branches:
    - 'releases/**'
    - '!releases/**-alpha'
```

on.<push|pull_request>.paths

When using the `push` and `pull_request` events, you can configure a workflow to run when at least one file does not match `paths-ignore` or at least one modified file matches the configured `paths`. Path filters are not evaluated for pushes to tags.

The `paths-ignore` and `paths` keywords accept glob patterns that use the `*` and `**` wildcard characters to match more than one path name. For more information, see the "Filter pattern cheat sheet."

Example ignoring paths

Anytime a path name matches a pattern in `paths-ignore`, the workflow will not run. GitHub evaluates patterns defined in `paths-ignore` against the path name. A workflow with the following path filter will only run on `push` events that include at least one file outside the `docs` directory at the root of the repository.

```
on:
  push:
```

```
    paths-ignore:
    - 'docs/**'
```

Example including paths

If at least one path matches a pattern in the `paths` filter, the workflow runs. To trigger a build anytime you push a JavaScript file, you can use a wildcard pattern.

```
on:
  push:
    paths:
    - '**.js'
```

Excluding paths

You can exclude paths using two types of filters. You cannot use both of these filters for the same event in a workflow.

- `paths-ignore` - Use the `paths-ignore` filter when you only need to exclude path names.
- `paths` - Use the `paths` filter when you need to filter paths for positive matches and exclude paths.

Example using positive and negative patterns

You can exclude `paths` using the `!` character. The order that you define patterns matters:

- A matching negative pattern (prefixed with `!`) after a positive match will exclude the path.
- A matching positive pattern after a negative match will include the path again.

This example runs anytime the `push` event includes a file in the `sub-project` directory or its subdirectories, unless the file is in the `sub-project/docs` directory. For example, a push that changed `sub-project/index.js` or `sub-project/src/index.js` will trigger a workflow run, but a push changing only `sub-project/docs/readme.md` will not.

```yaml
on:
  push:
    paths:
    - 'sub-project/**'
    - '!sub-project/docs/**'
```

Git diff comparisons

> **Note:** If you push more than 1,000 commits, or if GitHub does not generate the diff due to a timeout (diffs that are too large diffs), the workflow will always run.

The filter determines if a workflow should run by evaluating the changed files and running them against the `paths-ignore` or `paths` list. If there are no files changed, the workflow will not run.

GitHub generates the list of changed files using two-dot diffs for pushes and three-dot diffs for pull requests:

- **Pull requests:** Three-dot diffs are a comparison between the most recent version of the topic branch and the commit where the topic branch was last synced with the base branch.
- **Pushes to existing branches:** A two-dot diff compares the head and base SHAs directly with each other.
- **Pushes to new branches:** A two-dot diff against the parent of the ancestor of the deepest commit pushed.

For more information, see "About comparing branches in pull requests."

`on.schedule`

You can schedule a workflow to run at specific UTC times using POSIX cron syntax. Scheduled workflows run on the latest commit on the default or base branch. The shortest interval you can run scheduled workflows is once every 5 minutes.

This example triggers the workflow every 15 minutes:

```yaml
on:
```

```
schedule:
  # * is a special character in YAML so you have to quote this string
  - cron:  '*/15 * * * *'
```

For more information about cron syntax, see "Events that trigger workflows."

`env`

A `map` of environment variables that are available to all jobs and steps in the workflow. You can also set environment variables that are only available to a job or step. For more information, see `jobs.<job_id>.env` and `jobs.<job_id>.steps.env`.

When more than one environment variable is defined with the same name, GitHub uses the most specific environment variable. For example, an environment variable defined in a step will override job and workflow variables with the same name, while the step executes. A variable defined for a job will override a workflow variable with the same name, while the job executes.

Example

```
env:
  SERVER: production
```

`defaults`

A `map` of default settings that will apply to all jobs in the workflow. You can also set default settings that are only available to a job. For more information, see `jobs.<job_id>.defaults`.

When more than one default setting is defined with the same name, GitHub uses the most specific default setting. For example, a default setting defined in a job will override a default setting that has the same name defined in a workflow.

`defaults.run`

You can provide default `shell` and `working-directory` options for all `run` steps in a workflow. You can also set default settings for `run` that are only available to a job.

For more information, see `jobs.<job_id>.defaults.run`. You cannot use contexts or expressions in this keyword.

When more than one default setting is defined with the same name, GitHub uses the most specific default setting. For example, a default setting defined in a job will override a default setting that has the same name defined in a workflow.

Example

```
defaults:
  run:
    shell: bash
    working-directory: scripts
```

`jobs`

A workflow run is made up of one or more jobs. Jobs run in parallel by default. To run jobs sequentially, you can define dependencies on other jobs using the `jobs.<job_id>.needs` keyword.

Each job runs in an environment specified by `runs-on`.

You can run an unlimited number of jobs as long as you are within the workflow usage limits. For more information, see "Usage limits and billing" for GitHub-hosted runners and "About self-hosted runners" for self-hosted runner usage limits.

If you need to find the unique identifier of a job running in a workflow run, you can use the GitHub API. For more information, see "Workflow Jobs."

`jobs.<job_id>`

Each job must have an id to associate with the job. The key `job_id` is a string and its value is a map of the job's configuration data. You must replace `<job_id>` with a string that is unique to the `jobs` object. The `<job_id>` must start with a letter or `_` and contain only alphanumeric characters, `-`, or `_`.

Example

```
jobs:
  my_first_job:
    name: My first job
  my_second_job:
    name: My second job
```

jobs.<job_id>.name

The name of the job displayed on GitHub.

jobs.<job_id>.needs

Identifies any jobs that must complete successfully before this job will run. It can be a string or array of strings. If a job fails, all jobs that need it are skipped unless the jobs use a conditional statement that causes the job to continue.

Example

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

In this example, `job1` must complete successfully before `job2` begins, and `job3` waits for both `job1` and `job2` to complete.

The jobs in this example run sequentially:

1. `job1`
2. `job2`
3. `job3`

`jobs.<job_id>.runs-on`

Required The type of machine to run the job on. The machine can be either a GitHub-hosted runner or a self-hosted runner.

GitHub-hosted runners

If you use a GitHub-hosted runner, each job runs in a fresh instance of a virtual environment specified by `runs-on`.

Available GitHub-hosted runner types are:

Virtual environment	YAML workflow label
Windows Server 2019	**`windows-latest`** or **`windows-2019`**
Ubuntu 20.04	**`ubuntu-20.04`**
Ubuntu 18.04	**`ubuntu-latest`** or **`ubuntu-18.04`**
Ubuntu 16.04	**`ubuntu-16.04`**
macOS Catalina 10.15	**`macos-latest`** or **`macos-10.15`**

> **Note:** The Ubuntu 20.04 virtual environment is currently provided as a preview only. The `ubuntu-latest` YAML workflow label still uses the Ubuntu 18.04 virtual environment.

Example

```
runs-on: ubuntu-latest
```

For more information, see "Virtual environments for GitHub-hosted runners."

Self-hosted runners

To specify a self-hosted runner for your job, configure `runs-on` in your workflow file with self-hosted runner labels.

All self-hosted runners have the `self-hosted` label, and you can select any self-hosted runner by providing only the `self-hosted` label. Alternatively, you can use `self-hosted` in an array with additional labels, such as labels for a specific operating system or system architecture, to select only the runner types you specify.

Example

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

For more information, see "About self-hosted runners" and "Using self-hosted runners in a workflow."

`jobs.<job_id>.outputs`

A `map` of outputs for a job. Job outputs are available to all downstream jobs that depend on this job. For more information on defining job dependencies, see `jobs.<job_id>.needs`.

Job outputs are strings, and job outputs containing expressions are evaluated on the runner at the end of each job. Outputs containing secrets are redacted on the runner and not sent to GitHub Actions.

To use job outputs in a dependent job, you can use the `needs` context. For more information, see "Context and expression syntax for GitHub Actions."

Example

```yaml
jobs:
  job1:
    runs-on: ubuntu-latest
    # Map a step output to a job output
    outputs:
      output1: ${{ steps.step1.outputs.test }}
      output2: ${{ steps.step2.outputs.test }}
    steps:
    - id: step1
      run: echo "::set-output name=test::hello"
    - id: step2
      run: echo "::set-output name=test::world"
```

```
  job2:
    runs-on: ubuntu-latest
    needs: job1
    steps:
    - run: echo ${{needs.job1.outputs.output1}} ${{needs.job1.outputs.outp
```

jobs.<job_id>.env

A `map` of environment variables that are available to all steps in the job. You can also set environment variables for the entire workflow or an individual step. For more information, see `env` and `jobs.<job_id>.steps.env`.

When more than one environment variable is defined with the same name, GitHub uses the most specific environment variable. For example, an environment variable defined in a step will override job and workflow variables with the same name, while the step executes. A variable defined for a job will override a workflow variable with the same name, while the job executes.

Example

```
jobs:
  job1:
    env:
      FIRST_NAME: Mona
```

jobs.<job_id>.defaults

A `map` of default settings that will apply to all steps in the job. You can also set default settings for the entire workflow. For more information, see `defaults`.

When more than one default setting is defined with the same name, GitHub uses the most specific default setting. For example, a default setting defined in a job will override a default setting that has the same name defined in a workflow.

jobs.<job_id>.defaults.run

Provide default `shell` and `working-directory` to all `run` steps in the job. Context and expression are not allowed in this section.

You can provide default `shell` and `working-directory` options for all `run` steps in a job. You can also set default settings for `run` for the entire workflow. For more information, see `jobs.defaults.run`. You cannot use contexts or expressions in this keyword.

When more than one default setting is defined with the same name, GitHub uses the most specific default setting. For example, a default setting defined in a job will override a default setting that has the same name defined in a workflow.

Example

```
jobs:
  job1:
    runs-on: ubuntu-latest
    defaults:
      run:
        shell: bash
        working-directory: scripts
```

jobs.<job_id>.if

You can use the `if` conditional to prevent a job from running unless a condition is met. You can use any supported context and expression to create a conditional.

When you use expressions in an `if` conditional, you may omit the expression syntax (`${{ }}`) because GitHub automatically evaluates the `if` conditional as an expression. For more information, see "Context and expression syntax for GitHub Actions."

jobs.<job_id>.steps

A job contains a sequence of tasks called `steps`. Steps can run commands, run setup tasks, or run an action in your repository, a public repository, or an action published in a Docker registry. Not all steps run actions, but all actions run as a step. Each step runs in its own process in the runner environment and has access to the workspace and filesystem. Because steps run in their own process, changes to environment variables are not preserved between steps. GitHub provides built-in steps to set up and complete a job.

You can run an unlimited number of steps as long as you are within the workflow usage limits. For more information, see "Usage limits and billing" for GitHub-hosted runners and "About self-hosted runners" for self-hosted runner usage limits.

Example

```yaml
name: Greeting from Mona

on: push

jobs:
  my-job:
    name: My Job
    runs-on: ubuntu-latest
    steps:
    - name: Print a greeting
      env:
        MY_VAR: Hi there! My name is
        FIRST_NAME: Mona
        MIDDLE_NAME: The
        LAST_NAME: Octocat
      run: |
        echo $MY_VAR $FIRST_NAME $MIDDLE_NAME $LAST_NAME.
```

`jobs.<job_id>.steps.id`

A unique identifier for the step. You can use the `id` to reference the step in contexts. For more information, see "Context and expression syntax for GitHub Actions."

`jobs.<job_id>.steps.if`

You can use the `if` conditional to prevent a step from running unless a condition is met. You can use any supported context and expression to create a conditional.

When you use expressions in an `if` conditional, you may omit the expression syntax (`${{ }}`) because GitHub automatically evaluates the `if` conditional as an expression. For more information, see "Context and expression syntax for GitHub Actions."

Example using contexts

This step only runs when the event type is a `pull_request` and the event action is `unassigned`.

```
steps:
 - name: My first step
   if: ${{ github.event_name == 'pull_request' && github.event.action == '
   run: echo This event is a pull request that had an assignee removed.
```

Example using status check functions

The `my backup step` only runs when the previous step of a job fails. For more information, see "Context and expression syntax for GitHub Actions."

```
steps:
  - name: My first step
    uses: monacorp/action-name@main
  - name: My backup step
    if: ${{ failure() }}
    uses: actions/heroku@master
```

`jobs.<job_id>.steps.name`

A name for your step to display on GitHub.

`jobs.<job_id>.steps.uses`

Selects an action to run as part of a step in your job. An action is a reusable unit of code. You can use an action defined in the same repository as the workflow, a public repository, or in a published Docker container image.

We strongly recommend that you include the version of the action you are using by specifying a Git ref, SHA, or Docker tag number. If you don't specify a version, it could break your workflows or cause unexpected behavior when the action owner publishes an update.

- Using the commit SHA of a released action version is the safest for stability and security.
- Using the specific major action version allows you to receive critical fixes and security patches while still maintaining compatibility. It also assures that your

workflow should still work.
- Using the default branch of an action may be convenient, but if someone releases a new major version with a breaking change, your workflow could break.

Some actions require inputs that you must set using the `with` keyword. Review the action's README file to determine the inputs required.

Actions are either JavaScript files or Docker containers. If the action you're using is a Docker container you must run the job in a Linux environment. For more details, see `runs-on`.

Example using versioned actions

```yaml
steps:
  # Reference a specific commit
  - uses: actions/setup-node@74bc508
  # Reference the major version of a release
  - uses: actions/setup-node@v1
  # Reference a minor version of a release
  - uses: actions/setup-node@v1.2
  # Reference a branch
  - uses: actions/setup-node@main
```

Example using a public action

```
{owner}/{repo}@{ref}
```

You can specific branch, ref, or SHA in a public GitHub repository.

```yaml
jobs:
  my_first_job:
    steps:
      - name: My first step
        # Uses the default branch of a public repository
        uses: actions/heroku@master
      - name: My second step
        # Uses a specific version tag of a public repository
        uses: actions/aws@v2.0.1
```

Example using a public action in a subdirectory

```
{owner}/{repo}/{path}@{ref}
```

A subdirectory in a public GitHub repository at a specific branch, ref, or SHA.

```yaml
jobs:
  my_first_job:
    steps:
      - name: My first step
        uses: actions/aws/ec2@main
```

Example using action in the same repository as the workflow

```
./path/to/dir
```

The path to the directory that contains the action in your workflow's repository. You must check out your repository before using the action.

```yaml
jobs:
  my_first_job:
    steps:
      - name: Check out repository
        uses: actions/checkout@v2
      - name: Use local my-action
        uses: ./.github/actions/my-action
```

Example using a Docker Hub action

```
docker://{image}:{tag}
```

A Docker image published on Docker Hub.

```yaml
jobs:
  my_first_job:
    steps:
      - name: My first step
        uses: docker://alpine:3.8
```

Example using a Docker public registry action

```
docker://{host}/{image}:{tag}
```

A Docker image in a public registry.

```
jobs:
  my_first_job:
    steps:
      - name: My first step
        uses: docker://gcr.io/cloud-builders/gradle
```

`jobs.<job_id>.steps.run`

Runs command-line programs using the operating system's shell. If you do not provide a `name`, the step name will default to the text specified in the `run` command.

Commands run using non-login shells by default. You can choose a different shell and customize the shell used to run commands. For more information, see "Using a specific shell."

Each `run` keyword represents a new process and shell in the runner environment. When you provide multi-line commands, each line runs in the same shell. For example:

- A single-line command:

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

- A multi-line command:

```
- name: Clean install dependencies and build
  run: |
    npm ci
    npm run build
```

Using the `working-directory` keyword, you can specify the working directory of where to run the command.

```
- name: Clean temp directory
  run: rm -rf *
  working-directory: ./temp
```

Using a specific shell

You can override the default shell settings in the runner's operating system using the `shell` keyword. You can use built-in `shell` keywords, or you can define a custom set of shell options.

Supported platform	**`shell`** parameter	Description	Command run internally
All	**`bash`**	The default shell on non-Windows platforms with a fallback to **`sh`**. When specifying a bash shell on Windows, the bash shell included with Git for Windows is used.	**`bash --noprofile --norc -eo pipefail {0}`**
All	**`pwsh`**	The PowerShell Core. GitHub appends the extension **`.ps1`** to your script name.	**`pwsh -command ". '{0}'"`**
All	**`python`**	Executes the python command.	**`python {0}`**
Linux / macOS	**`sh`**	The fallback behavior for non-Windows platforms if no shell is provided and **`bash`** is not found in the path.	**`sh -e {0}`**
Windows	**`cmd`**	GitHub appends the extension **`.cmd`** to your script name and substitutes for **`{0}`**.	**`%ComSpec% /D /E:ON /V:OFF /S /C "CALL "{0}""`**.
Windows	**`powershell`**	This is the default shell used on Windows. The Desktop PowerShell. GitHub appends the extension **`.ps1`** to your script name.	**`powershell -command ". '{0}'"`**.

Example running a script using bash

```
steps:
  - name: Display the path
    run: echo $PATH
```

```
    shell: bash
```

Example running a script using Windows `cmd`

```
steps:
  - name: Display the path
    run: echo %PATH%
    shell: cmd
```

Example running a script using PowerShell Core

```
steps:
  - name: Display the path
    run: echo ${env:PATH}
    shell: pwsh
```

Example running a python script

```
steps:
  - name: Display the path
    run: |
      import os
      print(os.environ['PATH'])
    shell: python
```

Custom shell

You can set the `shell` value to a template string using `command […options] {0} [..more_options]`. GitHub interprets the first whitespace-delimited word of the string as the command, and inserts the file name for the temporary script at `{0}`.

Exit codes and error action preference

For built-in shell keywords, we provide the following defaults that are executed by GitHub-hosted runners. You should use these guidelines when running shell scripts.

- `bash` / `sh`:

    - Fail-fast behavior using `set -e o pipefail` : Default for `bash` and built-in `shell` . It is also the default when you don't provide an option on non-Windows platforms.
    - You can opt out of fail-fast and take full control by providing a template string to the shell options. For example, `bash {0}` .
    - sh-like shells exit with the exit code of the last command executed in a script, which is also the default behavior for actions. The runner will report the status of the step as fail/succeed based on this exit code.

- `powershell` / `pwsh`
    - Fail-fast behavior when possible. For `pwsh` and `powershell` built-in shell, we will prepend `$ErrorActionPreference = 'stop'` to script contents.
    - We append `if ((Test-Path -LiteralPath variable:\LASTEXITCODE)) { exit $LASTEXITCODE }` to powershell scripts so action statuses reflect the script's last exit code.
    - Users can always opt out by not using the built-in shell, and providing a custom shell option like: `pwsh -File {0}` , or `powershell -Command "& '{0}'"` , depending on need.

- `cmd`
    - There doesn't seem to be a way to fully opt into fail-fast behavior other than writing your script to check each error code and respond accordingly. Because we can't actually provide that behavior by default, you need to write this behavior into your script.
    - `cmd.exe` will exit with the error level of the last program it executed, and it will and return the error code to the runner. This behavior is internally consistent with the previous `sh` and `pwsh` default behavior and is the `cmd.exe` default, so this behavior remains intact.

`jobs.<job_id>.steps.with`

A `map` of the input parameters defined by the action. Each input parameter is a key/value pair. Input parameters are set as environment variables. The variable is prefixed with `INPUT_` and converted to upper case.

Example

Defines the three input parameters (`first_name` , `middle_name` , and `last_name`) defined by the `hello_world` action. These input variables will be accessible to the `hello-world` action as `INPUT_FIRST_NAME` , `INPUT_MIDDLE_NAME` , and `INPUT_LAST_NAME` environment variables.

```yaml
jobs:
  my_first_job:
    steps:
      - name: My first step
        uses: actions/hello_world@main
        with:
          first_name: Mona
          middle_name: The
          last_name: Octocat
```

`jobs.<job_id>.steps.with.args`

A `string` that defines the inputs for a Docker container. GitHub passes the `args` to the container's `ENTRYPOINT` when the container starts up. An `array of strings` is not supported by this parameter.

Example

```yaml
steps:
  - name: Explain why this job ran
    uses: monacorp/action-name@main
    with:
      entrypoint: /bin/echo
      args: The ${{ github.event_name }} event triggered this step.
```

The `args` are used in place of the `CMD` instruction in a `Dockerfile` . If you use `CMD` in your `Dockerfile` , use the guidelines ordered by preference:

1. Document required arguments in the action's README and omit them from the `CMD` instruction.
2. Use defaults that allow using the action without specifying any `args` .
3. If the action exposes a `--help` flag, or something similar, use that as the

default to make your action self-documenting.

`jobs.<job_id>.steps.with.entrypoint`

Overrides the Docker `ENTRYPOINT` in the `Dockerfile`, or sets it if one wasn't already specified. Unlike the Docker `ENTRYPOINT` instruction which has a shell and exec form, `entrypoint` keyword accepts only a single string defining the executable to be run.

Example

```yaml
steps:
  - name: Run a custom command
    uses: monacorp/action-name@main
    with:
      entrypoint: /a/different/executable
```

The `entrypoint` keyword is meant to use with Docker container actions, but you can also use it with JavaScript actions that don't define any inputs.

`jobs.<job_id>.steps.env`

Sets environment variables for steps to use in the runner environment. You can also set environment variables for the entire workflow or a job. For more information, see `env` and `jobs.<job_id>.env`.

When more than one environment variable is defined with the same name, GitHub uses the most specific environment variable. For example, an environment variable defined in a step will override job and workflow variables with the same name, while the step executes. A variable defined for a job will override a workflow variable with the same name, while the job executes.

Public actions may specify expected environment variables in the README file. If you are setting a secret in an environment variable, you must set secrets using the `secrets` context. For more information, see "Using environment variables" and "Context and expression syntax for GitHub Actions."

Example

```
steps:
  - name: My first action
    env:
      GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
      FIRST_NAME: Mona
      LAST_NAME: Octocat
```

`jobs.<job_id>.steps.continue-on-error`

Prevents a job from failing when a step fails. Set to `true` to allow a job to pass when this step fails.

`jobs.<job_id>.steps.timeout-minutes`

The maximum number of minutes to run the step before killing the process.

`jobs.<job_id>.timeout-minutes`

The maximum number of minutes to let a job run before GitHub automatically cancels it. Default: 360

`jobs.<job_id>.strategy`

A strategy creates a build matrix for your jobs. You can define different variations of an environment to run each job in.

`jobs.<job_id>.strategy.matrix`

You can define a matrix of different job configurations. A matrix allows you to create multiple jobs by performing variable substitution in a single job definition. For example, you can use a matrix to create jobs for more than one supported version of a programming language, operating system, or tool. A matrix reuses the job's configuration and creates a job for each matrix you configure.

- **Job matrix** - A job matrix can generate a maximum of 256 jobs per workflow run. This limit also applies to self-hosted runners.

Each option you define in the `matrix` has a key and value. The keys you define

become properties in the `matrix` context and you can reference the property in other areas of your workflow file. For example, if you define the key `os` that contains an array of operating systems, you can use the `matrix.os` property as the value of the `runs-on` keyword to create a job for each operating system. For more information, see "Context and expression syntax for GitHub Actions."

The order that you define a `matrix` matters. The first option you define will be the first job that runs in your workflow.

Example running with more than one version of Node.js

You can specify a matrix by supplying an array for the configuration options. For example, if the runner supports Node.js versions 6, 8, and 10, you could specify an array of those versions in the `matrix`.

This example creates a matrix of three jobs by setting the `node` key to an array of three Node.js versions. To use the matrix, the example sets the `matrix.node` context property as the value of the `setup-node` action's input parameter `node-version`. As a result, three jobs will run, each using a different Node.js version.

```yaml
strategy:
  matrix:
    node: [6, 8, 10]
steps:
  # Configures the node version used on GitHub-hosted runners
  - uses: actions/setup-node@v1
    with:
      # The Node.js version to configure
      node-version: ${{ matrix.node }}
```

The `setup-node` action is the recommended way to configure a Node.js version when using GitHub-hosted runners. For more information, see the `setup-node` action.

Example running with more than one operating system

You can create a matrix to run workflows on more than one runner operating system. You can also specify more than one matrix configuration. This example creates a matrix of 6 jobs:

- 2 operating systems specified in the `os` array
- 3 Node.js versions specified in the `node` array

When you define a matrix of operating systems, you must set the value of `runs-on` to the `matrix.os` context property you defined.

```yaml
runs-on: ${{ matrix.os }}
strategy:
  matrix:
    os: [ubuntu-16.04, ubuntu-18.04]
    node: [6, 8, 10]
steps:
  - uses: actions/setup-node@v1
    with:
      node-version: ${{ matrix.node }}
```

To find supported configuration options for GitHub-hosted runners, see "Virtual environments for GitHub-hosted runners."

Example including additional values into combinations

You can add additional configuration options to a build matrix job that already exists. For example, if you want to use a specific version of `npm` when the job that uses `windows-latest` and version 4 of `node` runs, you can use `include` to specify that additional option.

```yaml
runs-on: ${{ matrix.os }}
strategy:
  matrix:
    os: [macos-latest, windows-latest, ubuntu-18.04]
    node: [4, 6, 8, 10]
    include:
      # includes a new variable of npm with a value of 2
      # for the matrix leg matching the os and version
      - os: windows-latest
        node: 4
        npm: 2
```

Example including new combinations

You can use `include` to add new jobs to a build matrix. Any unmatched include configurations are added to the matrix. For example, if you want to use `node` version 12 to build on multiple operating systems, but wanted one extra experimental job using node version 13 on Ubuntu, you can use `include` to specify that additional job.

```yaml
runs-on: ${{ matrix.os }}
strategy:
  matrix:
    node: [12]
    os: [macos-latest, windows-latest, ubuntu-18.04]
    include:
      - node: 13
        os: ubuntu-18.04
        experimental: true
```

Example excluding configurations from a matrix

You can remove a specific configurations defined in the build matrix using the `exclude` option. Using `exclude` removes a job defined by the build matrix. The number of jobs is the cross product of the number of operating systems (`os`) included in the arrays you provide, minus any subtractions (`exclude`).

```yaml
runs-on: ${{ matrix.os }}
strategy:
  matrix:
    os: [macos-latest, windows-latest, ubuntu-18.04]
    node: [4, 6, 8, 10]
    exclude:
      # excludes node 4 on macOS
      - os: macos-latest
        node: 4
```

> **Note:** All `include` combinations are processed after `exclude` . This allows you to use `include` to add back combinations that were previously excluded.

`jobs.<job_id>.strategy.fail-fast`

When set to `true` , GitHub cancels all in-progress jobs if any `matrix` job fails. Default: `true`

`jobs.<job_id>.strategy.max-parallel`

The maximum number of jobs that can run simultaneously when using a `matrix` job

strategy. By default, GitHub will maximize the number of jobs run in parallel depending on the available runners on GitHub-hosted virtual machines.

```yaml
strategy:
  max-parallel: 2
```

jobs.<job_id>.continue-on-error

Prevents a workflow run from failing when a job fails. Set to `true` to allow a workflow run to pass when this job fails.

Example preventing a specific failing matrix job from failing a workflow run

You can allow specific jobs in a job matrix to fail without failing the workflow run. For example, if you wanted to only allow an experimental job with `node` set to `13` to fail without failing the workflow run.

```yaml
runs-on: ${{ matrix.os }}
continue-on-error: ${{ matrix.experimental }}
strategy:
  fail-fast: false
  matrix:
    node: [11, 12]
    os: [macos-latest, ubuntu-18.04]
    experimental: [false]
    include:
      - node: 13
        os: ubuntu-18.04
        experimental: true
```

jobs.<job_id>.container

A container to run any steps in a job that don't already specify a container. If you have steps that use both script and container actions, the container actions will run as sibling containers on the same network with the same volume mounts.

If you do not set a `container`, all steps will run directly on the host specified by `runs-on` unless a step refers to an action configured to run in a container.

Example

```yaml
jobs:
  my_job:
    container:
      image: node:10.16-jessie
      env:
        NODE_ENV: development
      ports:
        - 80
      volumes:
        - my_docker_volume:/volume_mount
      options: --cpus 1
```

When you only specify a container image, you can omit the `image` keyword.

```yaml
jobs:
  my_job:
    container: node:10.16-jessie
```

`jobs.<job_id>.container.image`

The Docker image to use as the container to run the action. The value can be the Docker Hub image name or a registry name.

`jobs.<job_id>.container.credentials`

If the image's container registry requires authentication to pull the image, you can use `credentials` to set a `map` of the `username` and `password`. The credentials are the same values that you would provide to the `docker login` command.

Example

```yaml
container:
  image: ghcr.io/owner/image
  credentials:
     username: ${{ github.actor }}
     password: ${{ secrets.ghcr_token }}
```

`jobs.<job_id>.container.env`

Sets a `map` of environment variables in the container.

`jobs.<job_id>.container.ports`

Sets an `array` of ports to expose on the container.

`jobs.<job_id>.container.volumes`

Sets an `array` of volumes for the container to use. You can use volumes to share data between services or other steps in a job. You can specify named Docker volumes, anonymous Docker volumes, or bind mounts on the host.

To specify a volume, you specify the source and destination path:

`<source>:<destinationPath>`.

The `<source>` is a volume name or an absolute path on the host machine, and `<destinationPath>` is an absolute path in the container.

Example

```
volumes:
  - my_docker_volume:/volume_mount
  - /data/my_data
  - /source/directory:/destination/directory
```

`jobs.<job_id>.container.options`

Additional Docker container resource options. For a list of options, see "`docker create` options."

`jobs.<job_id>.services`

> **Note:** If your workflows use Docker container actions or service containers, then you must use a Linux runner:

- If you are using GitHub-hosted runners, you must use the `ubuntu-latest` runner.
- If you are using self-hosted runners, you must use a Linux machine as your runner and Docker must be installed.

Used to host service containers for a job in a workflow. Service containers are useful for creating databases or cache services like Redis. The runner automatically creates a Docker network and manages the life cycle of the service containers.

If you configure your job to run in a container, or your step uses container actions, you don't need to map ports to access the service or action. Docker automatically exposes all ports between containers on the same Docker user-defined bridge network. You can directly reference the service container by its hostname. The hostname is automatically mapped to the label name you configure for the service in the workflow.

If you configure the job to run directly on the runner machine and your step doesn't use a container action, you must map any required Docker service container ports to the Docker host (the runner machine). You can access the service container using localhost and the mapped port.

For more information about the differences between networking service containers, see "About service containers."

Example using localhost

This example creates two services: nginx and redis. When you specify the Docker host port but not the container port, the container port is randomly assigned to a free port. GitHub sets the assigned container port in the `${{job.services.<service_name>.ports}}` context. In this example, you can access the service container ports using the `${{ job.services.nginx.ports['8080'] }}` and `${{ job.services.redis.ports['6379'] }}` contexts.

```
services:
  nginx:
    image: nginx
    # Map port 8080 on the Docker host to port 80 on the nginx container
    ports:
      - 8080:80
  redis:
    image: redis
    # Map TCP port 6379 on Docker host to a random free port on the Redis
```

```
    ports:
      - 6379/tcp
```

`jobs.<job_id>.services.<service_id>.image`

The Docker image to use as the service container to run the action. The value can be the Docker Hub image name or a registry name.

`jobs.<job_id>.services.<service_id>.credentials`

If the image's container registry requires authentication to pull the image, you can use `credentials` to set a `map` of the `username` and `password`. The credentials are the same values that you would provide to the `docker login` command.

Example

```
services:
  myservice1:
    image: ghcr.io/owner/myservice1
    credentials:
      username: ${{ github.actor }}
      password: ${{ secrets.ghcr_token }}
  myservice2:
    image: dockerhub_org/myservice2
    credentials:
      username: ${{ secrets.DOCKER_USER }}
      password: ${{ secrets.DOCKER_PASSWORD }}
```

`jobs.<job_id>.services.<service_id>.env`

Sets a `map` of environment variables in the service container.

`jobs.<job_id>.services.<service_id>.ports`

Sets an `array` of ports to expose on the service container.

`jobs.<job_id>.services.<service_id>.volumes`

Sets an `array` of volumes for the service container to use. You can use volumes to share data between services or other steps in a job. You can specify named Docker volumes, anonymous Docker volumes, or bind mounts on the host.

To specify a volume, you specify the source and destination path:

`<source>:<destinationPath>`.

The `<source>` is a volume name or an absolute path on the host machine, and `<destinationPath>` is an absolute path in the container.

Example

```
volumes:
  - my_docker_volume:/volume_mount
  - /data/my_data
  - /source/directory:/destination/directory
```

`jobs.<job_id>.services.<service_id>.options`

Additional Docker container resource options. For a list of options, see "`docker create` options."

Filter pattern cheat sheet

You can use special characters in path, branch, and tag filters.

- `*`: Matches zero or more characters, but does not match the `/` character. For example, `Octo*` matches `Octocat`.
- `**`: Matches zero or more of any character.
- `?`: Matches zero or one single character. For example, `Octoc?t` matches `Octocat`.
- `+`: Matches one or more of the preceding character.
- `[]` Matches one character listed in the brackets or included in ranges. Ranges can only include `a-z`, `A-Z`, and `0-9`. For example, the range `[0-9a-f]` matches any digits or lowercase letter. For example, `[CB]at` matches `Cat` or `Bat` and `[1-2]00` matches `100` and `200`.
- `!`: At the start of a pattern makes it negate previous positive patterns. It has no

special meaning if not the first character.

The characters `*`, `[`, and `!` are special characters in YAML. If you start a pattern with `*`, `[`, or `!`, you must enclose the pattern in quotes.

```yaml
# Valid
- '**/README.md'

# Invalid - creates a parse error that
# prevents your workflow from running.
- **/README.md
```

For more information about branch, tag, and path filter syntax, see "`on.<push|pull_request>.<branches|tags>`" and "`on.<push|pull_request>.paths`."

Patterns to match branches and tags

Pattern	Description	Example matches
`feature/*`	The **`*`** wildcard matches any character, but does not match slash (`/`).	**`feature/my-branch`** **`feature/your-branch`**
`feature/**`	The **`**`** wildcard matches any character including slash (`/`) in branch and tag names.	**`feature/beta-a/my-branch`** **`feature/your-branch`** **`feature/mona/the/octocat`**
`main` **`releases/mona-the-octcat`**	Matches the exact name of a branch or tag name.	**`main`** **`releases/mona-the-octocat`**
`'*'`	Matches all branch and tag names that don't contain a slash (`/`). The **`*`** character is a special character in YAML. When you start a pattern with **`*`**, you must use quotes.	**`main`** **`releases`**

Pattern	Description	Example matches
`'**'`	Matches all branch and tag names. This is the default behavior when you don't use a **`branches`** or **`tags`** filter.	-**`all/the/branches`** -**`every/tag`**
`'*feature'`	The **`*`** character is a special character in YAML. When you start a pattern with **`*`**, you must use quotes.	-**`mona-feature`** -**`feature`** -**`ver-10-feature`**
`v2*`	Matches branch and tag names that start with **`v2`**.	-**`v2`** -**`v2.0`** -**`v2.9`**
`v[12].[0-9]+.[0-9]+`	Matches all semantic versioning tags with major version 1 or 2	-**`v1.10.1`** -**`v2.0.0`**

Patterns to match file paths

Path patterns must match the whole path, and start from the repository's root.

Pattern	Description of matches	Example matches
`'*'`	The **`*`** wildcard matches any character, but does not match slash (**`/`**). The **`*`** character is a special character in YAML. When you start a pattern with **`*`**, you must use quotes.	-**`README.md`** -**`server.rb`**
`'*.jsx?'`	The **`?`** character matches zero or one of the preceding character.	-**`page.js`** -**`page.jsx`**
`'**'`	The **`**`** wildcard matches any character including slash (**`/`**). This is the default behavior when you don't use a **`path`** filter.	-**`all/the/files.md`**
`'*.js'`	The **`*`** wildcard matches any character, but does not match slash (**`/`**). Matches all **`.js`** files at the root of the repository.	-**`app.js`** -**`index.js`**
`'**.js'`	Matches all **`.js`** files in the repository.	-**`index.js`** -**`js/index.js`**

Pattern	Description of matches	Example matches
		-**src/js/app.js**
docs/*	All files within the root of the **docs** directory, at the root of the repository.	-**docs/README.md** -**docs/file.txt**
docs/**	Any files in the **/docs** directory at the root of the repository.	-**docs/README.md** -**docs/mona** **/octocat.txt**
docs//*.md**	A file with a **.md** suffix anywhere in the **docs** directory.	-**docs/README.md** -**docs/mona/hello-world.md** -**docs/a/markdown/file.md**
'/docs/**'**	Any files in a **docs** directory anywhere in the repository.	-**/docs/hello.md** -**dir/docs/my-file.txt** -**space/docs/plan/space.doc**
'/README.md'**	A README.md file anywhere in the repository.	-**README.md** -**js/README.md**
'/*src/**'**	Any file in a folder with a **src** suffix anywhere in the repository.	-**a/src/app.js** -**my-src/code/js/app.js**
'/*-post.md'**	A file with the suffix **-post.md** anywhere in the repository.	-**my-post.md** -**path/their-post.md**
'/migrate-*.sql'**	A file with the prefix **migrate-** and suffix **.sql** anywhere in the repository.	-**migrate-10909.sql** -**db/migrate-v1.0.sql** -**db/sept/migrate-v1.sql**
-***.md** -**!README.md**	Using an exclamation mark (**!**) in front of a pattern negates it. When a file matches a	-**hello.md** *Does not match*

Pattern	Description of matches	Example matches
	pattern and also matches a negative pattern defined later in the file, the file will not be included.	-**`README.md`** -**`docs/hello.md`**
-**`*.md`** -**`!README.md`** -**`README*`**	Patterns are checked sequentially. A pattern that negates a previous pattern will re-include file paths.	-**`hello.md`** -**`README.md`** -**`README.doc`**

Did this doc help you?

Help us make these docs great!

All GitHub docs are open source. See something that's wrong or unclear? Submit a pull request.

Make a contribution

Or, learn how to contribute.