Finding and customizing actions

Get started with GitHub Actions
2 of 6 in learning path

Next: Essential features of GitHub Actions

In this article

Actions are the building blocks that power your workflow. A workflow can contain actions created by the community, or you can create your own actions directly within your application's repository. This guide will show you how to discover, use, and customize actions.

Note: GitHub-hosted runners are not currently supported on GitHub Enterprise Server. You can see more information about planned future support on the GitHub public roadmap.

Overview

The actions you use in your workflow can be defined in:

- The same repository as your workflow file
- An internal repository within the same enterprise account that is configured to allow access to workflows
- Any public repository
- A published Docker container image on Docker Hub

GitHub Marketplace is a central location for you to find actions created by the GitHub community.

Note: GitHub Actions on your GitHub Enterprise Server instance may have limited access to actions on GitHub.com or GitHub Marketplace. For more information, see "Managing access to actions from GitHub.com" and contact your GitHub Enterprise site administrator.

Adding an action from the same repository

If an action is defined in the same repository where your workflow file uses the action, you can reference the action with either the `{owner}/{repo}@{ref}` or `./path/to/dir` syntax in your workflow file.

Example repository file structure:

```
|-- hello-world (repository)
|   |__ .github
|       └── workflows
|           └── my-first-workflow.yml
|       └── actions
|           |__ hello-world-action
|               └── action.yml
```

Example workflow file:

```
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      # This step checks out a copy of your repository.
      - uses: actions/checkout@v4
      # This step references the directory that contains the action.
      - uses: ./.github/actions/hello-world-action
```

The `action.yml` file is used to provide metadata for the action. Learn about the content of this file in "Metadata syntax for GitHub Actions."

Adding an action from a different repository

If an action is defined in a different repository than your workflow file, you can reference the action with the `{owner}/{repo}@{ref}` syntax in your workflow file.

The action must be stored in a public repository or an internal repository that is configured to allow access to workflows. For more information, see "Sharing actions and workflows with your enterprise."

```
jobs:
  my_first_job:
    steps:
      - name: My first step
        uses: actions/setup-node@v3
```

Referencing a container on Docker Hub

If an action is defined in a published Docker container image on Docker Hub, you must reference the action with the `docker://{image}:{tag}` syntax in your workflow file. To protect your code and data, we strongly recommend you verify the integrity of the Docker container image from Docker Hub before using it in your workflow.

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

For some examples of Docker actions, see the Docker-image.yml workflow and "Creating a Docker container action."

Using release management for your custom actions

The creators of a community action have the option to use tags, branches, or SHA values to manage releases of the action. Similar to any dependency, you should indicate the version of the action you'd like to use based on your comfort with automatically accepting updates to the action.

You will designate the version of the action in your workflow file. Check the action's documentation for information on their approach to release management, and to see which tag, branch, or SHA value to use.

Note: We recommend that you use a SHA value when using third-party actions. However, it's important to note Dependabot will only create Dependabot alerts for vulnerable GitHub Actions that use semantic versioning. For more information, see "Security hardening for GitHub Actions" and "About Dependabot alerts."

Using tags

Tags are useful for letting you decide when to switch between major and minor versions, but these are more ephemeral and can be moved or deleted by the maintainer. This example demonstrates how to target an action that's been tagged as `v1.0.1`:

```
steps:
  - uses: actions/javascript-action@v1.0.1
```

Using SHAs

If you need more reliable versioning, you should use the SHA value associated with the version of the action. SHAs are immutable and therefore more reliable than tags or branches. However, this approach means you will not automatically receive updates for an action, including important bug fixes and security updates. You must use a commit's full SHA value, and not an abbreviated value. When selecting a SHA, you should verify it is from the action's repository and not a repository fork. This example targets an action's SHA:

```
steps:
  - uses: actions/javascript-action@a824008085750b8e136effc585c3cd6082bd575f
```

Using branches

Specifying a target branch for the action means it will always run the version currently on that branch. This approach can create problems if an update to the branch includes breaking changes. This example targets a branch named `@main`:

```
steps:
  - uses: actions/javascript-action@main
```

For more information, see "About custom actions."

Using inputs and outputs with an action

An action often accepts or requires inputs and generates outputs that you can use. For example, an action might require you to specify a path to a file, the name of a label, or other data it will use as part of the action processing.

To see the inputs and outputs of an action, check the `action.yml` or `action.yaml` in the root directory of the repository.

In this example `action.yml`, the `inputs` keyword defines a required input called `file-path`, and includes a default value that will be used if none is specified. The `outputs` keyword defines an output called `results-file`, which tells you where to locate the results.

```
name: "Example"
description: "Receives file and generates output"
inputs:
  file-path: # id of input
    description: "Path to test script"
    required: true
    default: "test-file.js"
outputs:
  results-file: # id of output
    description: "Path to results file"
```

Next steps

To continue learning about GitHub Actions, see "Essential features of GitHub Actions."

Previous
Understanding GitHub Actions

Next
Essential features of GitHub Actions