Creating a Docker container action

In this article

This guide shows you the minimal steps required to build a Docker container action.

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.

Introduction

In this guide, you'll learn about the basic components needed to create and use a packaged Docker container action. To focus this guide on the components needed to package the action, the functionality of the action's code is minimal. The action prints "Hello World" in the logs or "Hello [who-to-greet]" if you provide a custom name.

Once you complete this project, you should understand how to build your own Docker container action and test it in a workflow.

Self-hosted runners must use a Linux operating system and have Docker installed to run Docker container actions. For more information about the requirements of self-hosted runners, see "About self-hosted runners."

Warning: When creating workflows and actions, you should always consider whether your code might execute untrusted input from possible attackers. Certain contexts should be treated as untrusted input, as an attacker could insert their own malicious content. For more information, see "Security hardening for GitHub Actions."

Prerequisites

- You must create a repository on your GitHub Enterprise Server instance and clone it to your workstation. For more information, see "Creating a new repository" and "Cloning a repository."
- If your repository uses Git LFS, you must include the objects in archives of your repository. For more information, see "Managing Git LFS objects in archives of your

repository."

- You may find it helpful to have a basic understanding of GitHub Actions, environment variables and the Docker container filesystem. For more information, see "Variables" and "Using GitHub-hosted runners."

Creating a Dockerfile

In your new `hello-world-docker-action` directory, create a new `Dockerfile` file. Make sure that your filename is capitalized correctly (use a capital `D` but not a capital `f`) if you're having issues. For more information, see "Dockerfile support for GitHub Actions."

Dockerfile

Dockerfile

```
# Container image that runs your code
FROM alpine:3.10

# Copies your code file from your action repository to the filesystem path `/` of
the container
COPY entrypoint.sh /entrypoint.sh

# Code file to execute when the docker container starts up (`entrypoint.sh`)
ENTRYPOINT ["/entrypoint.sh"]
```

Creating an action metadata file

Create a new `action.yml` file in the `hello-world-docker-action` directory you created above. For more information, see "Metadata syntax for GitHub Actions."

action.yml

YAML

```yaml
# action.yml
name: 'Hello World'
description: 'Greet someone and record the time'
inputs:
  who-to-greet:  # id of input
    description: 'Who to greet'
    required: true
    default: 'World'
outputs:
  time: # id of output
    description: 'The time we greeted you'
runs:
  using: 'docker'
  image: 'Dockerfile'
  args:
    - ${{ inputs.who-to-greet }}
```

This metadata defines one `who-to-greet` input and one `time` output parameter. To pass inputs to the Docker container, you should declare the input using `inputs` and pass the input in the `args` keyword. Everything you include in `args` is passed to the container, but for better discoverability for users of your action, we recommended using inputs.

GitHub will build an image from your `Dockerfile`, and run commands in a new container using this image.

Writing the action code

You can choose any base Docker image and, therefore, any language for your action. The following shell script example uses the `who-to-greet` input variable to print "Hello [who-to-greet]" in the log file.

Next, the script gets the current time and sets it as an output variable that actions running later in a job can use. In order for GitHub to recognize output variables, you must write them to the `$GITHUB_OUTPUT` environment file: `echo "<output name>=<value>" >> $GITHUB_OUTPUT`. For more information, see "Workflow commands for GitHub Actions."

1. Create a new `entrypoint.sh` file in the `hello-world-docker-action` directory.
2. Add the following code to your `entrypoint.sh` file.

   entrypoint.sh

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

   echo "Hello $1"
   time=$(date)
   echo "time=$time" >> $GITHUB_OUTPUT
   ```

   If `entrypoint.sh` executes without any errors, the action's status is set to `success`. You can also explicitly set exit codes in your action's code to provide an action's status. For more information, see "Setting exit codes for actions."

3. Make your `entrypoint.sh` file executable. Git provides a way to explicitly change the permission mode of a file so that it doesn't get reset every time there is a clone/fork.

   ```shell
   git add entrypoint.sh
   git update-index --chmod=+x entrypoint.sh
   ```

4. Optionally, to check the permission mode of the file in the git index, run the following command.

   ```shell
   git ls-files --stage entrypoint.sh
   ```

   An output like `100755 e69de29bb2d1d6434b8b29ae775ad8c2e48c5391 0 entrypoint.sh` means the file has the executable permission. In this example, `755` denotes the executable permission.

Creating a README

To let people know how to use your action, you can create a README file. A README is most helpful when you plan to share your action publicly, but is also a great way to remind you or your team how to use the action.

In your `hello-world-docker-action` directory, create a `README.md` file that specifies the following information:

- A detailed description of what the action does.
- Required input and output arguments.
- Optional input and output arguments.
- Secrets the action uses.
- Environment variables the action uses.
- An example of how to use your action in a workflow.

README.md

Markdown

```
# Hello world docker action

This action prints "Hello World" or "Hello" + the name of a person to greet to
the log.

## Inputs

## `who-to-greet`

**Required** The name of the person to greet. Default `"World"`.

## Outputs

## `time`

The time we greeted you.

## Example usage

uses: actions/hello-world-docker-action@v2
with:
  who-to-greet: 'Mona the Octocat'
```

Commit, tag, and push your action to GitHub Enterprise Server

From your terminal, commit your `action.yml`, `entrypoint.sh`, `Dockerfile`, and `README.md` files.

It's best practice to also add a version tag for releases of your action. For more information on versioning your action, see "About custom actions."

Shell

```
git add action.yml entrypoint.sh Dockerfile README.md
git commit -m "My first action is ready"
git tag -a -m "My first action release" v1
git push --follow-tags
```

Testing out your action in a workflow

Now you're ready to test your action out in a workflow.

- When an action is in a private repository, you can control who can access it. For more information, see "Managing GitHub Actions settings for a repository."

- When an action is in an internal repository, you can control who can access it. For more information, see "Managing GitHub Actions settings for a repository."
- Public actions can be used by workflows in any repository.

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.

Example using a public action

The following workflow code uses the completed *hello world* action in the public `actions/hello-world-docker-action` repository. Copy the following workflow example code into a `.github/workflows/main.yml` file, but replace the `actions/hello-world-docker-action` with your repository and action name. You can also replace the `who-to-greet` input with your name.

.github/workflows/main.yml

YAML

```yaml
on: [push]

jobs:
  hello_world_job:
    runs-on: ubuntu-latest
    name: A job to say hello
    steps:
      - name: Hello world action step
        id: hello
        uses: actions/hello-world-docker-action@v2
        with:
          who-to-greet: 'Mona the Octocat'
      # Use the output from the `hello` step
      - name: Get the output time
        run: echo "The time was ${{ steps.hello.outputs.time }}"
```

Example using a private action

Copy the following example workflow code into a `.github/workflows/main.yml` file in your action's repository. You can also replace the `who-to-greet` input with your name.

.github/workflows/main.yml

YAML

```yaml
on: [push]

jobs:
  hello_world_job:
    runs-on: ubuntu-latest
    name: A job to say hello
    steps:
      # To use this repository's private action,
      # you must check out the repository
      - name: Checkout
        uses: actions/checkout@v4
      - name: Hello world action step
        uses: ./ # Uses an action in the root directory
        id: hello
        with:
          who-to-greet: 'Mona the Octocat'
      # Use the output from the `hello` step
```

```
      - name: Get the output time
        run: echo "The time was ${{ steps.hello.outputs.time }}"
```

From your repository, click the **Actions** tab, and select the latest workflow run. Under **Jobs** or in the visualization graph, click **A job to say hello**.

Click **Hello world action step**, and you should see "Hello Mona the Octocat" or the name you used for the `who-to-greet` input printed in the log. To see the timestamp, click **Get the output time**.

Accessing files created by a container action

When a container action runs, it will automatically map the default working directory (`GITHUB_WORKSPACE`) on the runner with the `/github/workspace` directory on the container. Any files added to this directory on the container will be available to any subsequent steps in the same job. For example, if you have a container action that builds your project, and you would like to upload the build output as an artifact, you can use the following steps.

workflow.yml

YAML

```yaml
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout
        uses: actions/checkout@v4

      # Output build artifacts to /github/workspace on the container.
      - name: Containerized Build
        uses: ./.github/actions/my-container-action

      - name: Upload Build Artifacts
        uses: actions/upload-artifact@v3
        with:
          name: workspace_artifacts
          path: ${{ github.workspace }}
```

For more information about uploading build output as an artifact, see "Storing workflow data as artifacts."

Example Docker container actions on GitHub.com

You can find many examples of Docker container actions on GitHub.com.

- github/issue-metrics
- microsoft/infersharpaction
- microsoft/ps-docs