Working with the Container registry

In this article

You can store and manage Docker and OCI images in the Container registry, which uses the package namespace `https://ghcr.io`.

GitHub Packages is available with GitHub Free, GitHub Pro, GitHub Free for organizations, GitHub Team, GitHub Enterprise Cloud, GitHub Enterprise Server 3.0 or higher, and GitHub AE.

GitHub Packages is not available for private repositories owned by accounts using legacy per-repository plans. Also, accounts using legacy per-repository plans cannot access registries that support granular permissions, because these accounts are billed by repository. For the list of registries that support granular permissions, see "About permissions for GitHub Packages." For more information, see "GitHub's plans."

About the Container registry

The Container registry stores container images within your organization or personal account, and allows you to associate an image with a repository. You can choose whether to inherit permissions from a repository, or set granular permissions independently of a repository. You can also access public container images anonymously.

About Container registry support

The Container registry currently supports the following container image formats:

- Docker Image Manifest V2, Schema 2
- Open Container Initiative (OCI) Specifications

When installing or publishing a Docker image, the Container registry supports foreign layers, such as Windows images.

Authenticating to the Container registry

GitHub Packages only supports authentication using a personal access token (classic). For more

information, see "Managing your personal access tokens."

You need an access token to publish, install, and delete private, internal, and public packages.

You can use a personal access token (classic) to authenticate to GitHub Packages or the GitHub API. When you create a personal access token (classic), you can assign the token different scopes depending on your needs. For more information about packages-related scopes for a personal access token (classic), see "About permissions for GitHub Packages."

To authenticate to a GitHub Packages registry within a GitHub Actions workflow, you can use:

- `GITHUB_TOKEN` to publish packages associated with the workflow repository.
- a personal access token (classic) with at least `read:packages` scope to install packages associated with other private repositories (which `GITHUB_TOKEN` can't access).

Authenticating in a GitHub Actions workflow

This registry supports granular permissions. For registries that support granular permissions, if your GitHub Actions workflow is using a personal access token to authenticate to a registry, we highly recommend you update your workflow to use the `GITHUB_TOKEN`. For guidance on updating your workflows that authenticate to a registry with a personal access token, see "Publishing and installing a package with GitHub Actions."

Note: The ability for GitHub Actions workflows to delete and restore packages using the REST API is currently in public beta and subject to change.

You can use a `GITHUB_TOKEN` in a GitHub Actions workflow to delete or restore a package using the REST API, if the token has `admin` permission to the package. Repositories that publish packages using a workflow, and repositories that you have explicitly connected to packages, are automatically granted `admin` permission to packages in the repository.

For more information about the `GITHUB_TOKEN`, see "Automatic token authentication." For more information about the best practices when using a registry in actions, see "Security hardening for GitHub Actions."

You can also choose to give access permissions to packages independently for GitHub Codespaces and GitHub Actions. For more information, see "Configuring a package's access control and visibility" and "Configuring a package's access control and visibility."

Authenticating with a personal access token (classic)

GitHub Packages only supports authentication using a personal access token (classic). For more information, see "Managing your personal access tokens."

1. Create a new personal access token (classic) with the appropriate scopes for the tasks you want to accomplish. If your organization requires SSO, you must enable SSO for your new token.

   Note: By default, when you select the `write:packages` scope for your personal access token (classic) in the user interface, the `repo` scope will also be selected. The `repo` scope offers unnecessary and broad access, which we recommend you avoid using for GitHub Actions workflows in particular. For more information, see "Security hardening for GitHub Actions." As a workaround, you can select just the `write:packages` scope for your personal

access token (classic) in the user interface with this url: `https://github.com/settings/tokens/new?scopes=write:packages`.

- Select the `read:packages` scope to download container images and read their metadata.
- Select the `write:packages` scope to download and upload container images and read and write their metadata.
- Select the `delete:packages` scope to delete container images.

For more information, see "Managing your personal access tokens."

2. Save your personal access token (classic). We recommend saving your token as an environment variable.

```
export CR_PAT=YOUR_TOKEN
```

3. Using the CLI for your container type, sign in to the Container registry service at `ghcr.io`.

```
$ echo $CR_PAT | docker login ghcr.io -u USERNAME --password-stdin
> Login Succeeded
```

Pushing container images

This example pushes the latest version of `IMAGE_NAME`.

```
docker push ghcr.io/NAMESPACE/IMAGE_NAME:latest
```

Replace `NAMESPACE` with the name of the personal account or organization to which you want the image to be scoped.

This example pushes the `2.5` version of the image.

```
docker push ghcr.io/NAMESPACE/IMAGE_NAME:2.5
```

When you first publish a package, the default visibility is private. To change the visibility or set access permissions, see "Configuring a package's access control and visibility." You can link a published package to a repository using the user interface or command line. For more information, see "Connecting a repository to a package."

When you push a container image from the command line, the image is not linked to a repository by default. This is the case even if you tag the image with a namespace that matches the name of the repository, such as `ghcr.io/octocat/my-repo:latest`.

The easiest way to connect a repository to a container package is to publish the package from a workflow using `${{secrets.GITHUB_TOKEN}}`, as the repository that contains the workflow is linked automatically. Note that the `GITHUB_TOKEN` will not have permission to push the package if you have previously pushed a package to the same namespace, but have not connected the package to the repository.

To connect a repository when publishing an image from the command line, and to ensure your `GITHUB_TOKEN` has appropriate permissions when using a GitHub Actions workflow, we recommend adding the label `org.opencontainers.image.source` to your `Dockerfile`. For more information, see “Labelling container images” in this article and “Publishing and installing a package with GitHub Actions.”

Pulling container images

Pull by digest

To ensure you're always using the same image, you can specify the exact container image version you want to pull by the `digest` SHA value.

1. To find the digest SHA value, use `docker inspect` or `docker pull` and copy the SHA value after `Digest:`

   ```
   docker inspect ghcr.io/NAMESPACE/IMAGE_NAME
   ```

   Replace `NAMESPACE` with the name of the personal account or organization to which the image is scoped.

2. Remove image locally as needed.

   ```
   docker rmi  ghcr.io/NAMESPACE/IMAGE_NAME:latest
   ```

3. Pull the container image with `@YOUR_SHA_VALUE` after the image name.

   ```
   docker pull
   ghcr.io/NAMESPACE/IMAGE_NAME@sha256:82jf9a84u29hiasldj289498uhois8498hjs29hkuhs
   ```

Pull by name

```
docker pull ghcr.io/NAMESPACE/IMAGE_NAME
```

Replace `NAMESPACE` with the name of the personal account or organization to which the image is scoped.

Pull by name and version

Docker CLI example showing an image pulled by its name and the `1.14.1` version tag:

```
$ docker pull ghcr.io/NAMESPACE/IMAGE_NAME:1.14.1
> 5e35bd43cf78: Pull complete
> 0c48c2209aab: Pull complete
> fd45dd1aad5a: Pull complete
> db6eb50c2d36: Pull complete
> Digest: sha256:ae3b135f133155b3824d8b1f62959ff8a72e9cf9e884d88db7895d8544010d8e
> Status: Downloaded newer image for ghcr.io/NAMESPACE/IMAGE_NAME/release:1.14.1
> ghcr.io/NAMESPACE/IMAGE_NAME/release:1.14.1
```

Replace `NAMESPACE` with the name of the personal account or organization to which the image is scoped.

Pull by name and latest version

```
$ docker pull ghcr.io/NAMESPACE/IMAGE_NAME:latest
> latest: Pulling from NAMESPACE/IMAGE_NAME
> Digest: sha256:b3d3e366b55f9a54599220198b3db5da8f53592acbbb7dc7e4e9878762fc5344
```

```
> Status: Downloaded newer image for ghcr.io/NAMESPACE/IMAGE_NAME:latest
> ghcr.io/NAMESPACE/IMAGE_NAME:latest
```

Replace `NAMESPACE` with the name of the personal account or organization to which the image is scoped.

Building container images

This example builds the `hello_docker` image:

```
docker build -t hello_docker .
```

Tagging container images

1. Find the ID for the Docker image you want to tag.

   ```
   $ docker images
   > REPOSITORY                                         TAG
   IMAGE ID            CREATED             SIZE
   > ghcr.io/my-org/hello_docker         latest            38f737a91f39
   47 hours ago        91.7MB
   > hello-world                                        latest
   fce289e99eb9        16 months ago       1.84kB
   ```

2. Tag your Docker image using the image ID and your desired image name and hosting destination.

   ```
   docker tag 38f737a91f39 ghcr.io/NAMESPACE/NEW_IMAGE_NAME:latest
   ```

Replace `NAMESPACE` with the name of the personal account or organization to which you want the image to be scoped.

Labelling container images

You can use pre-defined annotation keys to add metadata including a description, a license, and a source repository to your container image. Values for supported keys will appear on the package page for the image.

For most images, you can use Docker labels to add the annotation keys to an image. For more information, see LABEL in the official Docker documentation and Pre-Defined Annotation Keys in the `opencontainers/image-spec` repository.

For multi-arch images, you can add a description to the image by adding the appropriate annotation key to the `annotations` field in the image's manifest. For more information, see "Adding a description to multi-arch images."

The following annotation keys are supported in the Container registry.

Key	Description
`org.opencontainers.image.source`	The URL of the repository associated with the package. For more information, see "Connecting a repository to a package."
`org.opencontainers.image.description`	A text-only description limited to 512

	characters. This description will appear on the package page, below the name of the package.
`org.opencontainers.image.licenses`	An SPDX license identifier such as "MIT," limited to 256 characters. The license will appear on the package page, in the "Details" sidebar. For more information, see SPDX License List.

To add a key as a Docker label, we recommend using the `LABEL` instruction in your `Dockerfile`. For example, if you're the user `octocat` and you own `my-repo`, and your image is distributed under the terms of the MIT license, you would add the following lines to your `Dockerfile`:

```
LABEL org.opencontainers.image.source=https://github.com/octocat/my-repo
LABEL org.opencontainers.image.description="My container image"
LABEL org.opencontainers.image.licenses=MIT
```

Note: If you publish a package that is linked to a repository, the package automatically inherits the access permissions of the linked repository, and GitHub Actions workflows in the linked repository automatically get access to the package, unless your organization has disabled automatic inheritance of access permissions. For more information, see "Configuring a package's access control and visibility."

Alternatively, you can add labels to an image at buildtime with the `docker build` command.

```
$ docker build \
 --label "org.opencontainers.image.source=https://github.com/octocat/my-repo" \
 --label "org.opencontainers.image.description=My container image" \
 --label "org.opencontainers.image.licenses=MIT"
```

Adding a description to multi-arch images

A multi-arch image is an image that supports multiple architectures. It works by referencing a list of images, each supporting a different architecture, within a single manifest.

The description that appears on the package page for a multi-arch image is obtained from the `annotations` field in the image's manifest. Like Docker labels, annotations provide a way to associate metadata with an image, and support pre-defined annotation keys. For more information, see Annotations in the `opencontainers/image-spec` repository.

To provide a description for a multi-arch image, set a value for the `org.opencontainers.image.description` key in the `annotations` field of the manifest, as follows.

```
"annotations": {
  "org.opencontainers.image.description": "My multi-arch image"
}
```

For example, the following GitHub Actions workflow step builds and pushes a multi-arch image. The `outputs` parameter sets the description for the image.

```
# This workflow uses actions that are not certified by GitHub.
# They are provided by a third-party and are governed by
# separate terms of service, privacy policy, and support
# documentation.
```

```
- name: Build and push Docker image
  uses: docker/build-push-action@f2a1d5e99d037542a71f64918e516c093c6f3fc4
  with:
    context: .
    file: ./Dockerfile
    platforms: ${{ matrix.platforms }}
    push: true
    outputs: type=image,name=target,annotation-
index.org.opencontainers.image.description=My multi-arch image
```