

Forgejo Actions | Basic concepts



- [Basic syntax](#)
 - [Expressions](#)
 - [Variables](#)
 - [Literals](#)
 - [Logical operators](#)
 - [Conditionals](#)
 - [Functions](#)
 - [Secrets](#)
 - [Custom variables](#)
 - [Name constraints](#)
 - [Precedence](#)
 - [Automatic token](#)
 - [Actions](#)
 - [Using actions](#)
 - [Creating actions](#)
-

This page explains the basic concepts you'll encounter when using Forgejo Actions.

All Forgejo Actions workflows are defined in workflow files. These are `.yaml` files stored in the `.forgejo/workflows` directory of a repository. Below is an example of a simple workflow file:

Basic syntax

```
on: [push]
jobs:
  print-content:
```

```
runs-on: docker

steps:
  - name: checkout code
    uses: actions/checkout@v6
  - name: list directory contents
    run: ls -la
```

The `on` key specifies which events the workflow should trigger on. In this case it triggers on any push operation. You can narrow this down to specific branches, or trigger on other events like issues or pull requests.

The `jobs` key contains the jobs that are part of this workflow. A `job` is one unit of work that will be handed out to a runner. A workflow may contain multiple jobs. By default, these will run independently from one another.

This workflow contains one job, called `print-content`. The `runs-on` key specifies the label of the runner the workflow will run on. This job will only run on runners with the `docker` label.

A job consists of one or more steps. The steps are executed in order. If one step fails, the job will clean up and no further steps will be run.

The first step, `checkout code`, `uses` an action to check out the contents of the repository into the working directory. Actions are reusable procedures for doing something in your CI workflow. [Read more about actions](#).

The second step, `list directory contents`, `run`s the provided command in the shell of the job. In this case it lists the contents of the working directory.

That concludes the basic syntax of a workflow file. For a complete definition of the syntax of a workflow file, see the [workflow reference](#).

Expressions

In a workflow file you may use expressions. These look like `${{ ... }}`. Expressions are evaluated by the runner when executing a workflow. In certain fields of the workflow file, like `if:`, the `${{ }}` is implicit and can be stripped.

“ **NOTE:** Implicit type conversions with booleans may be difficult to figure out when assigned to actions inputs or output values. It is easier for debugging to use explicit strings (e.g. 'yes' or 'no') instead of true/false or 'true'/'false'.

[Check out the expression example](#)

Variables

You can use variables to read information from the workflow context. There are many different variables available. For example, the expression `${{ forgejo.ref_name }}` would be substituted with the name of the branch the workflow is running for.

For a complete list of the values available in the workflow context, see the [workflow reference](#).

Literals

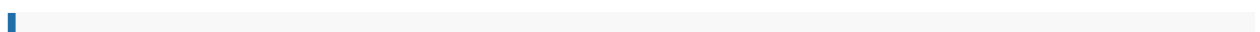
You may use the following types of literals in expressions:

- boolean: true or false
- null: null
- number: any number format supported by JSON
- string: enclosed in single quotes

Logical operators

You may use the following logical operators in expressions:

Operator	Description
<code>()</code>	Logical grouping
<code>[]</code>	Index
<code>.</code>	Property de-reference
<code>!</code>	Not
<code><</code>	Less than
<code><=</code>	Less than or equal
<code>></code>	Greater than
<code>>=</code>	Greater than or equal
<code>==</code>	Equal
<code>!=</code>	Not equal
<code>&&</code>	And
<code> </code>	Or



NOTE: String comparisons are case insensitive.

Conditionals

You may use these functions in `if:` conditionals on jobs and steps.

- `success()`. returns true when none of the previous jobs/steps have failed or been cancelled.
- `always()`. causes the job/step to always execute, and returns true, even when cancelled. If you want to run a job/step regardless of its success or failure, use the recommended alternative: `'!cancelled()'` (the expression must be enclosed by quotes to not be interpreted as YAML tag).
- `failure()`. returns true when any previous step/job has failed.

Functions

You may use these functions in expressions:

- `contains( search, item )`. Returns `true` if `search` contains `item`. If `search` is an array, this function returns `true` if the `item` is an element in the array. If `search` is a string, this function returns `true` if the `item` is a substring of `search`. This function is not case sensitive. Casts values to a string when comparing.
- `startsWith( searchString, searchValue )`. Returns `true` when `searchString` starts with `searchValue`. This function is not case sensitive. Casts values to a string.
- `endsWith( searchString, searchValue )`. Returns `true` if `searchString` ends with `searchValue`. This function is not case sensitive. Casts values to a string.
- `format( string, replaceValue0, replaceValue1, ..., replaceValueN )`. Replaces values in the `string`, with the variable `replaceValueN`. Variables in the `string` are specified using the `{N}` syntax, where `N` is an integer. You must specify at least one `replaceValue` and `string`. Escape curly braces using double braces.
- `join( array, optionalSeparator )`. The value for `array` can be an array or a string. All values in `array` are concatenated into a string. If you provide `optionalSeparator`, it is inserted between the concatenated values. Otherwise, the default separator `,` is used. Casts values to a string.
- `toJSON(value)`. Returns a pretty-print JSON representation of `value`.
- `fromJSON(value)`. Returns a JSON object or JSON data type for `value`. You can use this function to provide a JSON object as an evaluated expression or to convert environment variables from a string.
- `hashFiles(pattern[, ...])`. Returns a hash (e.g. `39b35c59b92fdb3024d09a4f2f960d123ba59c3d531569f5195b65fbf0a40161`) based on the concatenated content of all the files matching the [patterns](#) given in argument. Returns an empty string if the files are not found. It is a JavaScript function executed in the container

running the action and requires [Node.js](#) to be installed.

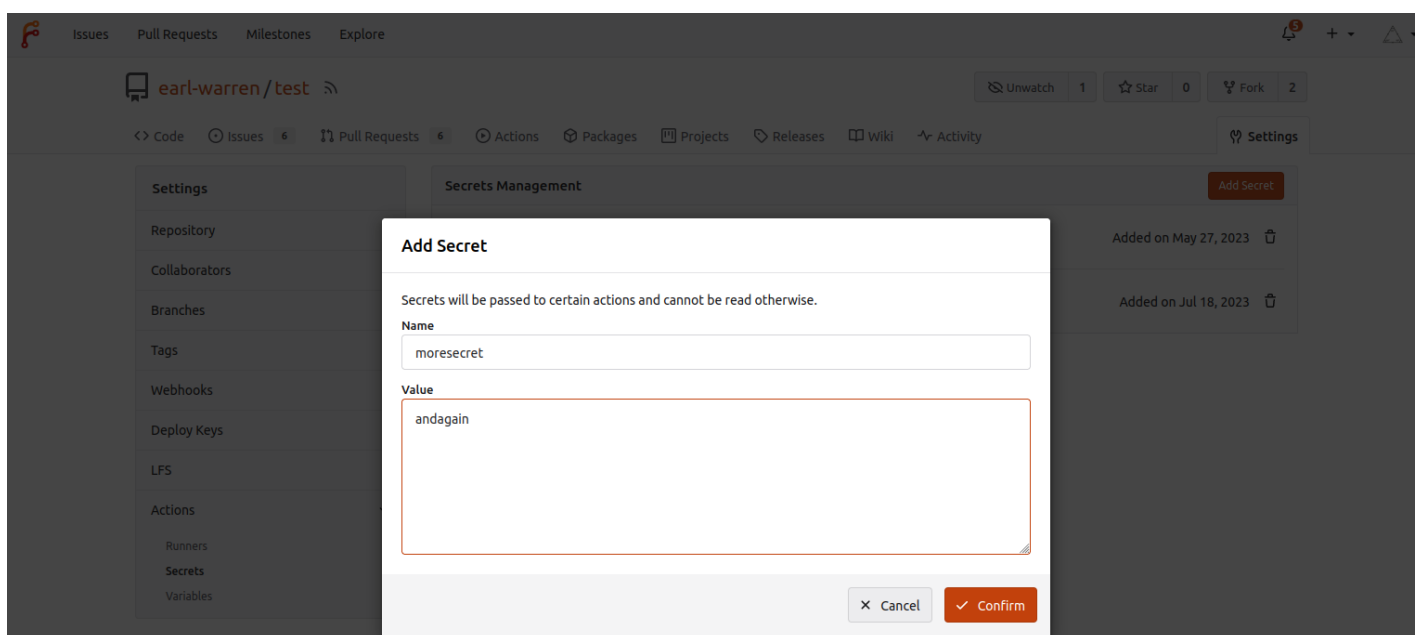
Some of these functions cast values to strings in order to compare them. The following conversions are used:

Input type	Output result
<code>null</code>	<code>''</code> (empty string)
booleans	<code>'true'</code> or <code>'false'</code>
numbers	decimal format, exponential for large numbers
arrays	arrays are never converted to strings
objects	objects are never converted to strings

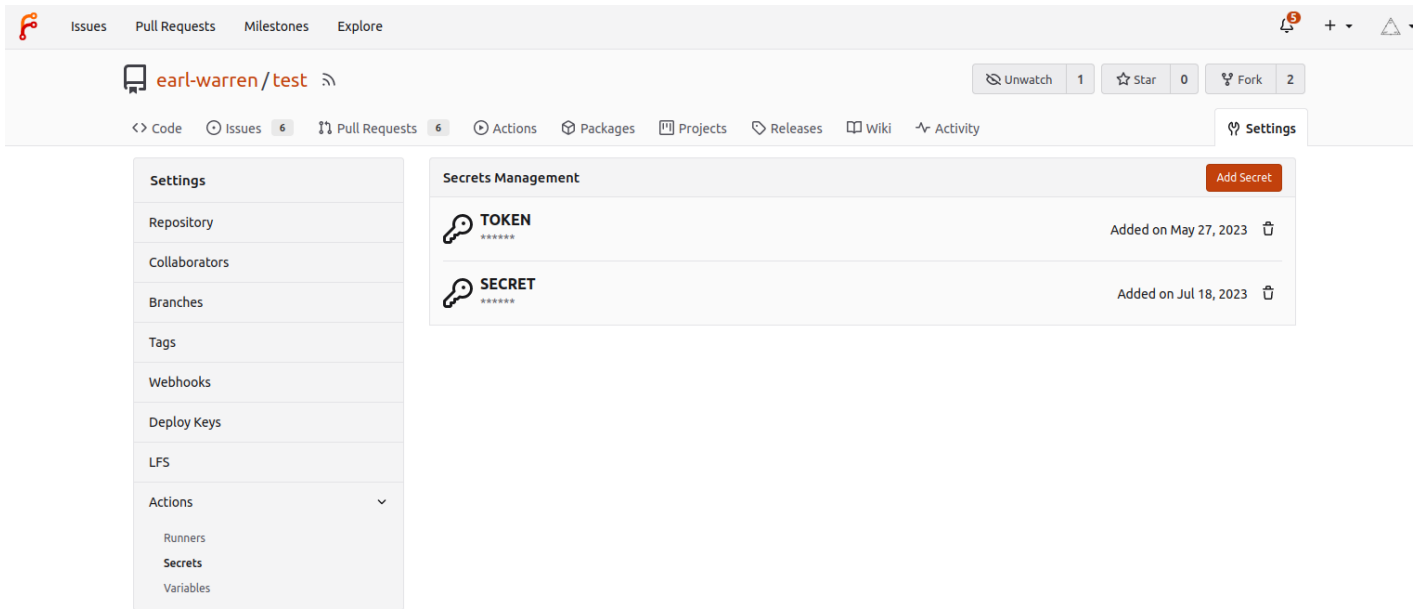
Secrets

A repository, a user or an organization can hold secrets, a set of key/value pairs that are stored encrypted in the Forgejo database and revealed to the workflows as `${{ secrets.KEY }}`. They can be defined from the web interface:

- in `/org/{org}/settings/actions/secrets` to be available in all the repositories that belong to the organization
- in `/user/settings/actions/secrets` to be available in all the repositories that belong to the logged in user
- in `/owner/{repo}/settings/actions/secrets` to be available to the `workflows` of a single repository



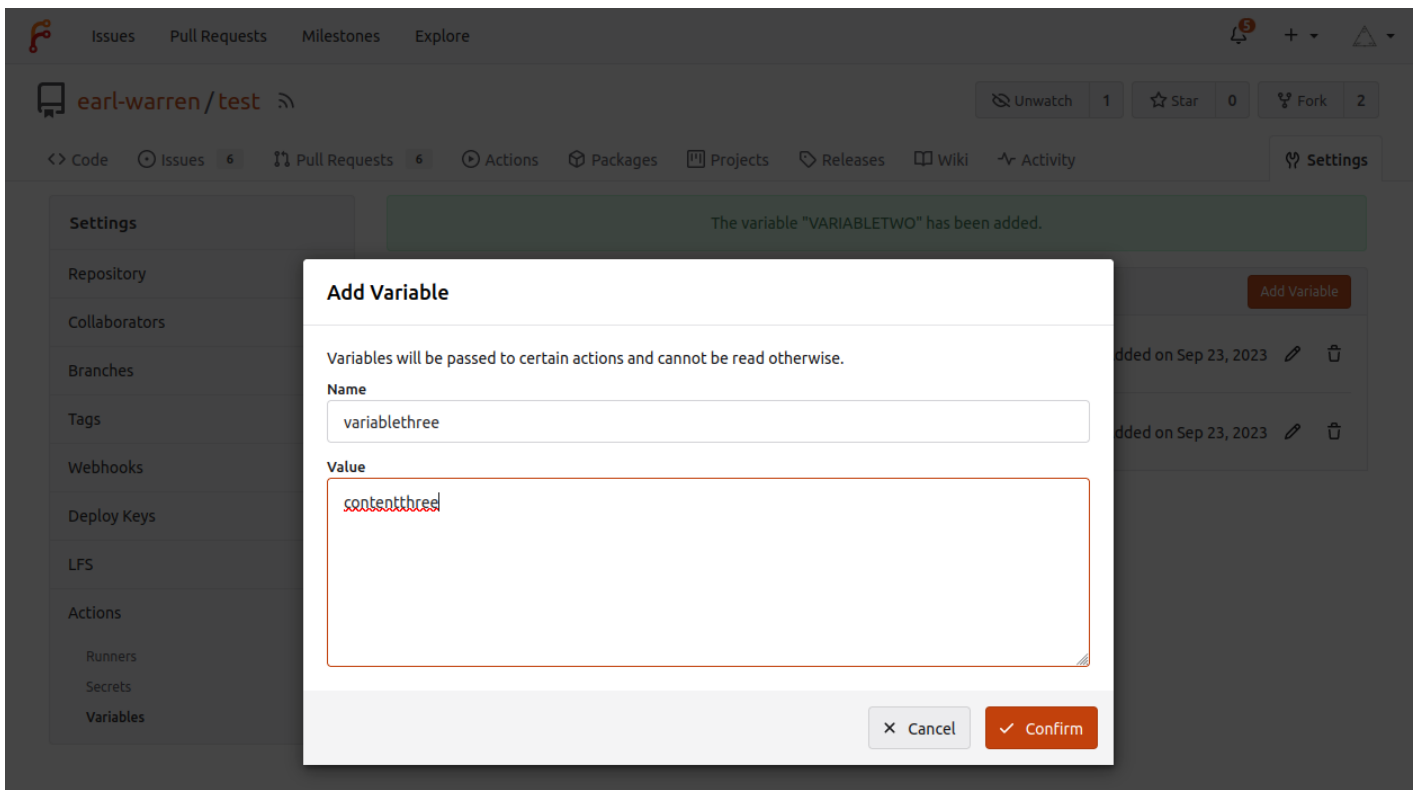
Once the secret is added, its value cannot be changed or displayed.



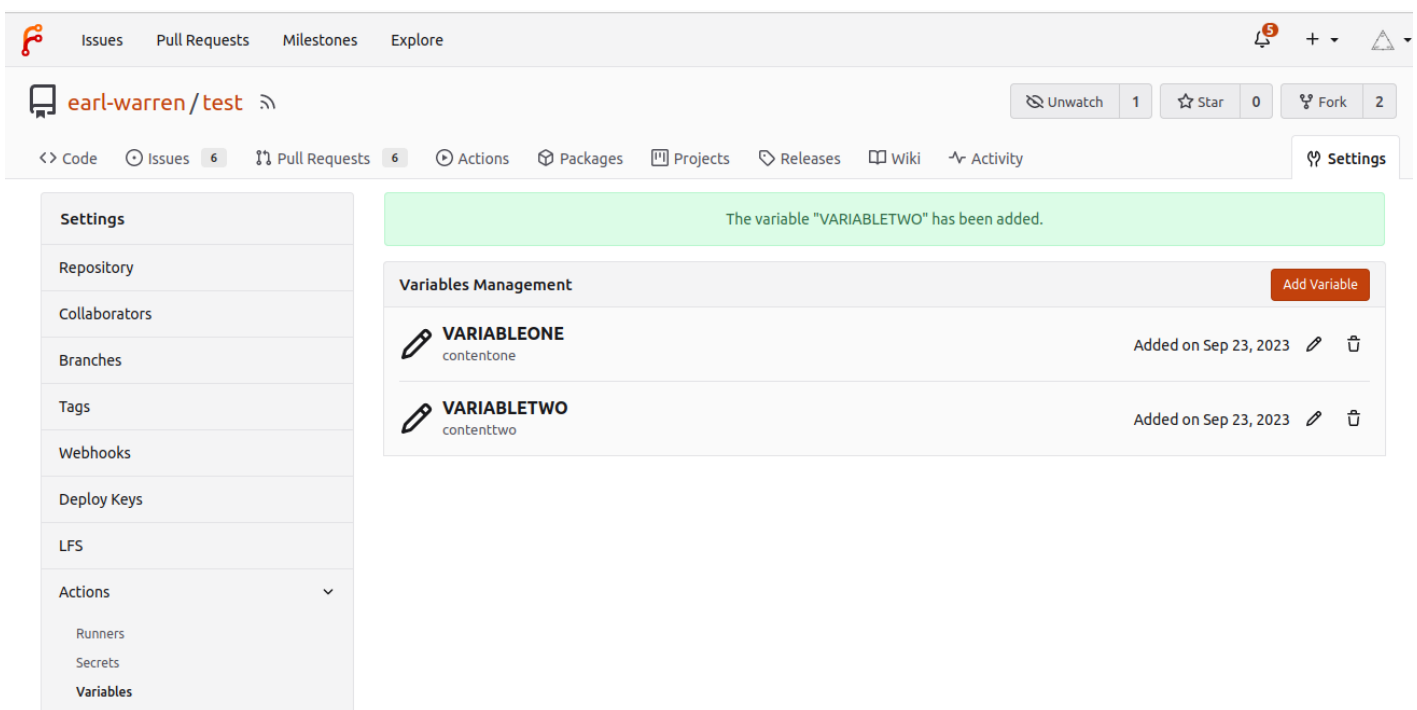
Custom variables

A repository, a user or an organization can hold variables, a set of key/value pairs that are stored in the Forgejo database and available to the workflows as `${{ vars.KEY }}`. They can be defined from the web interface:

- in `/org/{org}/settings/actions/variables` to be available in all the repositories that belong to the organization
- in `/user/settings/actions/variables` to be available in all the repositories that belong to the logged in user
- in `/owner/{repo}/settings/actions/variables` to be available to the `workflows` of a single repository



After a variable is added, its value can be modified.



Name constraints

The following rules apply to variable names:

- Variable names can only contain alphanumeric characters (`[a-z]`, `[A-Z]`, `[0-9]`) or underscores (`[_]`). Spaces are not allowed.

- Variable names must not start with the `FORGEJO_`, `GITHUB_` or `GITEA_` prefix.
- Variable names must not start with a number.
- Variable names are case-insensitive.
- Variable names must be unique at the level they are created at.
- Variable names must not be `CI`.

Precedence

A variable found in the settings of the owner of a repository (organization or user) has precedence over the same variable found in a repository.

Automatic token

At the start of each workflow, a unique authentication token is automatically created and destroyed when it completes. It can be used to read the repositories associated with the workflow, even when they are private. It is available:

- in the environment of each step as `FORGEJO_TOKEN`
- as `forgejo.token`
- as `env.FORGEJO_TOKEN`
- as `secrets.FORGEJO_TOKEN`

To help with reusing actions and workflows originally developed for GitHub Actions, they are also available under the following names:

- `GITHUB_TOKEN` == `FORGEJO_TOKEN`
- `github.token` == `forgejo.token`
- `env.GITHUB_TOKEN` == `env.FORGEJO_TOKEN`
- `secrets.GITHUB_TOKEN` == `secrets.FORGEJO_TOKEN`

This token can only be used for interactions with the repository of the project and any attempt to use it on other repositories, even for creating an issue, will return a 404 error.

This token also has write permission to the repository and can be used to push commits or use API endpoints such as creating a label or merge a pull request.

In order to avoid infinite recursion, no workflow will be triggered as a side effect of a change authored with this token. For instance, if a branch is pushed to the repository and there exists a workflow that is triggered on push events, it will not fire.

A workflow triggered by a `pull_request` or `pull_request_target` event from a forked repository is an exception: in that case the token does not have write permissions to the repository. The pull request could contain an untested or malicious workflow.

NOTE: In the case of a `pull_request_target` the risk is mitigated and the automatic token may be given write permissions in future versions of Forgejo.

Actions

An `action` is a reusable procedure for accomplishing something in your CI workflow.

Using actions

To use an action, specify the `uses:` key in a step like this:

```
- name: Checkout the repository
  uses: actions/checkout@v6
  with:
    ref: 'main'
```

The `uses` key contains the name of the repository the action is stored in. You can specify it in shorthand like above, in which case it will be prefixed with the `DEFAULT_ACTIONS_URL`. You can also specify the full url: `uses: https://codeberg.org/username/yourcustomaction`.

The `with` key contains parameters to be passed to the action. Which parameters are available depends on which action you are using. You can find in the documentation of the action you are using.

Creating actions

An action is a repository that contains the equivalent of a function in any programming language. It comes in two flavors, depending on the file found at the root of the repository:

- **action.yml:** describes the inputs and outputs of the action and the implementation. See [this example](#).
- **Dockerfile:** if no `action.yml` file is found, it is used to create an image with `docker build` and run a container from it to carry out the action. See [this example](#) and [the workflow that uses it](#). Note that files written outside of the **workspace** will be lost when the **step** using such an action terminates.

Just as any other program or function, an action has pre-requisites to successfully be installed and run. When looking at reusing an existing action, this is an important consideration. For instance

[setup-go](#) depends on NodeJS during installation.

To read more about creating your own actions, see the [actions guide](#).

👤 #3

👤 cocobook-sync 👤 2026-10-28 16:37:47 UTC

👤 cocobook-sync 👤 2026-10-29 01:09:29 UTC