Contexts

In this article

You can access context information in workflows and 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.

About contexts

Contexts are a way to access information about workflow runs, variables, runner environments, jobs, and steps. Each context is an object that contains properties, which can be strings or other objects.

Contexts, objects, and properties will vary significantly under different workflow run conditions. For example, the `matrix` context is only populated for jobs in a matrix.

You can access contexts using the expression syntax. For more information, see "Expressions."

```
${{ <context> }}
```

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."

Context name	Type	Description
`github`	`object`	Information about the workflow run. For more information, see `github` context.
`env`	`object`	Contains variables set in a

		workflow, job, or step. For more information, see `env` context.
`vars`	`object`	Contains variables set at the repository, organization, or environment levels. For more information, see `vars` context.
`job`	`object`	Information about the currently running job. For more information, see `job` context.
`jobs`	`object`	For reusable workflows only, contains outputs of jobs from the reusable workflow. For more information, see `jobs` context.
`steps`	`object`	Information about the steps that have been run in the current job. For more information, see `steps` context.
`runner`	`object`	Information about the runner that is running the current job. For more information, see `runner` context.
`secrets`	`object`	Contains the names and values of secrets that are available to a workflow run. For more information, see `secrets` context.
`strategy`	`object`	Information about the matrix execution strategy for the current job. For more information, see `strategy` context.
`matrix`	`object`	Contains the matrix properties defined in the workflow that apply to the current job. For more information, see `matrix` context.
`needs`	`object`	Contains the outputs of all jobs that are defined as a dependency of the current job. For more information, see `needs` context.
`inputs`	`object`	Contains the inputs of a reusable or manually triggered workflow. For more information, see `inputs` context.

As part of an expression, you can access context information using one of two syntaxes.

- Index syntax: `github['sha']`
- Property dereference syntax: `github.sha`

In order to use property dereference syntax, the property name must start with a letter or `_` and contain only alphanumeric characters, `-`, or `_`.

If you attempt to dereference a non-existent property, it will evaluate to an empty string.

Determining when to use contexts

GitHub Actions includes a collection of variables called *contexts* and a similar collection of variables called *default variables*. These variables are intended for use at different points in the workflow:

- **Default environment variables:** These environment variables exist only on the runner that is executing your job. For more information, see "Variables."
- **Contexts:** You can use most contexts at any point in your workflow, including when *default variables* would be unavailable. For example, you can use contexts with expressions to perform initial processing before the job is routed to a runner for execution; this allows you to use a context with the conditional `if` keyword to determine whether a step should run. Once the job is running, you can also retrieve context variables from the runner that is executing the job, such as `runner.os`. For details of where you can use various contexts within a workflow, see "Context availability."

The following example demonstrates how these different types of variables can be used together in a job:

```yaml
name: CI
on: push
jobs:
  prod-check:
    if: ${{ github.ref == 'refs/heads/main' }}
    runs-on: ubuntu-latest
    steps:
      - run: echo "Deploying to production server on branch $GITHUB_REF"
```

In this example, the `if` statement checks the `github.ref` context to determine the current branch name; if the name is `refs/heads/main`, then the subsequent steps are executed. The `if` check is processed by GitHub Actions, and the job is only sent to the runner if the result is `true`. Once the job is sent to the runner, the step is executed and refers to the `$GITHUB_REF` variable from the runner.

Context availability

Different contexts are available throughout a workflow run. For example, the `secrets` context may only be used at certain places within a job.

In addition, some functions may only be used in certain places. For example, the `hashFiles` function is not available everywhere.

The following table indicates where each context and special function can be used within a workflow. Unless listed below, a function can be used anywhere.

Workflow key	Context	Special functions
`run-name`	`github, inputs, vars`	None
`concurrency`	`github, inputs, vars`	None
`env`	`github, secrets, inputs, vars`	None
`jobs.<job_id>.concurrency`	`github, needs, strategy, matrix, inputs, vars`	None

`jobs.<job_id>.container`	`github, needs, strategy, matrix, vars, inputs`	None
`jobs.<job_id>.container.credentials`	`github, needs, strategy, matrix, env, vars, secrets, inputs`	None
`jobs.<job_id>.container.env.<env_id>`	`github, needs, strategy, matrix, job, runner, env, vars, secrets, inputs`	None
`jobs.<job_id>.container.image`	`github, needs, strategy, matrix, vars, inputs`	None
`jobs.<job_id>.continue-on-error`	`github, needs, strategy, vars, matrix, inputs`	None
`jobs.<job_id>.defaults.run`	`github, needs, strategy, matrix, env, vars, inputs`	None
`jobs.<job_id>.env`	`github, needs, strategy, matrix, vars, secrets, inputs`	None
`jobs.<job_id>.environment`	`github, needs, strategy, matrix, vars, inputs`	None
`jobs.<job_id>.environment.url`	`github, needs, strategy, matrix, job, runner, env, vars, steps, inputs`	None
`jobs.<job_id>.if`	`github, needs, vars, inputs`	`always, cancelled, success, failure`
`jobs.<job_id>.name`	`github, needs, strategy, matrix, vars, inputs`	None
`jobs.<job_id>.outputs.<output_id>`	`github, needs, strategy, matrix, job, runner, env, vars, secrets, steps, inputs`	None
`jobs.<job_id>.runs-on`	`github, needs, strategy, matrix, vars, inputs`	None
`jobs.<job_id>.secrets.<secrets_id>`	`github, needs, strategy, matrix, secrets, inputs, vars`	None
`jobs.<job_id>.services`	`github, needs, strategy, matrix, vars, inputs`	None
`jobs.<job_id>.services.<service_id>.credentials`	`github, needs, strategy, matrix, env, vars, secrets, inputs`	None
`jobs.<job_id>.services.<service_id>.env.<env_id>`	`github, needs, strategy, matrix, job, runner, env, vars, secrets, inputs`	None
`jobs.<job_id>.steps.continue-on-error`	`github, needs, strategy, matrix, job, runner, env, vars, secrets, steps, inputs`	`hashFiles`
`jobs.<job_id>.steps.env`	`github, needs, strategy, matrix, job, runner, env, vars,`	`hashFiles`

	`secrets, steps, inputs`	
`jobs.<job_id>.steps.if`	`github, needs, strategy, matrix, job, runner, env, vars, steps, inputs`	`always, cancelled, success, failure, hashFiles`
`jobs.<job_id>.steps.name`	`github, needs, strategy, matrix, job, runner, env, vars, secrets, steps, inputs`	`hashFiles`
`jobs.<job_id>.steps.run`	`github, needs, strategy, matrix, job, runner, env, vars, secrets, steps, inputs`	`hashFiles`
`jobs.<job_id>.steps.timeout-minutes`	`github, needs, strategy, matrix, job, runner, env, vars, secrets, steps, inputs`	`hashFiles`
`jobs.<job_id>.steps.with`	`github, needs, strategy, matrix, job, runner, env, vars, secrets, steps, inputs`	`hashFiles`
`jobs.<job_id>.steps.working-directory`	`github, needs, strategy, matrix, job, runner, env, vars, secrets, steps, inputs`	`hashFiles`
`jobs.<job_id>.strategy`	`github, needs, vars, inputs`	None
`jobs.<job_id>.timeout-minutes`	`github, needs, strategy, matrix, vars, inputs`	None
`jobs.<job_id>.with.<with_id>`	`github, needs, strategy, matrix, inputs, vars`	None
`on.workflow_call.inputs.<inputs_id>.default`	`github, inputs, vars`	None
`on.workflow_call.outputs.<output_id>.value`	`github, jobs, vars, inputs`	None

Example: printing context information to the log

You can print the contents of contexts to the log for debugging. The `toJSON` function is required to pretty-print JSON objects to the log.

> **Warning:** When using the whole `github` context, be mindful that it includes sensitive information such as `github.token`. GitHub masks secrets when they are printed to the console, but you should be cautious when exporting or printing the context.

YAML

```yaml
name: Context testing
on: push

jobs:
  dump_contexts_to_log:
    runs-on: ubuntu-latest
    steps:
      - name: Dump GitHub context
        env:
          GITHUB_CONTEXT: ${{ toJson(github) }}
```

```
        run: echo "$GITHUB_CONTEXT"
      - name: Dump job context
        env:
          JOB_CONTEXT: ${{ toJson(job) }}
        run: echo "$JOB_CONTEXT"
      - name: Dump steps context
        env:
          STEPS_CONTEXT: ${{ toJson(steps) }}
        run: echo "$STEPS_CONTEXT"
      - name: Dump runner context
        env:
          RUNNER_CONTEXT: ${{ toJson(runner) }}
        run: echo "$RUNNER_CONTEXT"
      - name: Dump strategy context
        env:
          STRATEGY_CONTEXT: ${{ toJson(strategy) }}
        run: echo "$STRATEGY_CONTEXT"
      - name: Dump matrix context
        env:
          MATRIX_CONTEXT: ${{ toJson(matrix) }}
        run: echo "$MATRIX_CONTEXT"
```

`github` context

The `github` context contains information about the workflow run and the event that triggered the run. You can also read most of the `github` context data in environment variables. For more information about environment variables, see "Variables."

Warning: When using the whole `github` context, be mindful that it includes sensitive information such as `github.token`. GitHub masks secrets when they are printed to the console, but you should be cautious when exporting or printing the context.

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."

Property name	Type	Description
`github`	`object`	The top-level context available during any job or step in a workflow. This object contains all the properties listed below.
`github.action`	`string`	The name of the action currently running, or the `id` of a step. GitHub removes special characters, and uses the name `__run` when the current step runs a script without an `id`. If you use the same action more than once in the same job, the name will include a suffix with the sequence number with underscore before it. For example, the first script you run will have the name `__run`, and the second script will be named `__run_2`. Similarly, the second invocation of `actions/checkout` will be `actionscheckout2`.

`github.action_path`	`string`	The path where an action is located. This property is only supported in composite actions. You can use this path to access files located in the same repository as the action, for example by changing directories to the path: `cd ${{ github.action_path }}` .
`github.action_ref`	`string`	For a step executing an action, this is the ref of the action being executed. For example, `v2` . Do not use in the `run` context. To make this context work with composite actions, reference it within the `env` context of the composite action.
`github.action_repository`	`string`	For a step executing an action, this is the owner and repository name of the action. For example, `actions/checkout` . Do not use in the `run` context. To make this context work with composite actions, reference it within the `env` context of the composite action.
`github.action_status`	`string`	For a composite action, the current result of the composite action.
`github.actor`	`string`	The username of the user that triggered the initial workflow run. If the workflow run is a re-run, this value may differ from `github.triggering_actor` . Any workflow re-runs will use the privileges of `github.actor` , even if the actor initiating the re-run (`github.triggering_actor`) has different privileges.
`github.actor_id`	`string`	The account ID of the person or app that triggered the initial workflow run. For example, `1234567` . Note that this is different from the actor username.
`github.api_url`	`string`	The URL of the GitHub REST API.
`github.base_ref`	`string`	The `base_ref` or target branch of the pull request in a workflow run. This property is only available when the event that triggers a workflow run is either `pull_request` or `pull_request_target` .

`github.env`	`string`	Path on the runner to the file that sets environment variables from workflow commands. This file is unique to the current step and is a different file for each step in a job. For more information, see "Workflow commands for GitHub Actions."
`github.event`	`object`	The full event webhook payload. You can access individual properties of the event using this context. This object is identical to the webhook payload of the event that triggered the workflow run, and is different for each event. The webhooks for each GitHub Actions event is linked in "Events that trigger workflows." For example, for a workflow run triggered by the `push` event, this object contains the contents of the push webhook payload.
`github.event_name`	`string`	The name of the event that triggered the workflow run.
`github.event_path`	`string`	The path to the file on the runner that contains the full event webhook payload.
`github.graphql_url`	`string`	The URL of the GitHub GraphQL API.
`github.head_ref`	`string`	The `head_ref` or source branch of the pull request in a workflow run. This property is only available when the event that triggers a workflow run is either `pull_request` or `pull_request_target`.
`github.job`	`string`	The `job_id` of the current job. Note: This context property is set by the Actions runner, and is only available within the execution `steps` of a job. Otherwise, the value of this property will be `null`.
`github.path`	`string`	Path on the runner to the file that sets system `PATH` variables from workflow commands. This file is unique to the current step and is a different file for each step in a job. For more information, see "Workflow commands for GitHub Actions."
`github.ref`	`string`	The fully-formed ref of the branch or tag that triggered the workflow run. For workflows

		triggered by `push`, this is the branch or tag ref that was pushed. For workflows triggered by `pull_request`, this is the pull request merge branch. For workflows triggered by `release`, this is the release tag created. For other triggers, this is the branch or tag ref that triggered the workflow run. This is only set if a branch or tag is available for the event type. The ref given is fully-formed, meaning that for branches the format is `refs/heads/<branch_name>`, for pull requests it is `refs/pull/<pr_number>/merge`, and for tags it is `refs/tags/<tag_name>`. For example, `refs/heads/feature-branch-1`.
`github.ref_name`	`string`	The short ref name of the branch or tag that triggered the workflow run. This value matches the branch or tag name shown on GitHub. For example, `feature-branch-1`.
`github.ref_protected`	`boolean`	`true` if branch protections are configured for the ref that triggered the workflow run.
`github.ref_type`	`string`	The type of ref that triggered the workflow run. Valid values are `branch` or `tag`.
`github.repository`	`string`	The owner and repository name. For example, `octocat/Hello-World`.
`github.repository_id`	`string`	The ID of the repository. For example, `123456789`. Note that this is different from the repository name.
`github.repository_owner`	`string`	The repository owner's username. For example, `octocat`.
`github.repository_owner_id`	`string`	The repository owner's account ID. For example, `1234567`. Note that this is different from the owner's name.
`github.repositoryUrl`	`string`	The Git URL to the repository. For example, `git://github.com/octocat/hello-world.git`.
`github.retention_days`	`string`	The number of days that workflow run logs and artifacts are kept.

`github.run_id`	`string`	A unique number for each workflow run within a repository. This number does not change if you re-run the workflow run.
`github.run_number`	`string`	A unique number for each run of a particular workflow in a repository. This number begins at 1 for the workflow's first run, and increments with each new run. This number does not change if you re-run the workflow run.
`github.run_attempt`	`string`	A unique number for each attempt of a particular workflow run in a repository. This number begins at 1 for the workflow run's first attempt, and increments with each re-run.
`github.secret_source`	`string`	The source of a secret used in a workflow. Possible values are `None`, `Actions`, or `Dependabot`.
`github.server_url`	`string`	The URL of the GitHub server. For example: `https://github.com`.
`github.sha`	`string`	The commit SHA that triggered the workflow. The value of this commit SHA depends on the event that triggered the workflow. For more information, see "Events that trigger workflows." For example, `ffac537e6cbbf934b08745a378932722df287a53`.
`github.token`	`string`	A token to authenticate on behalf of the GitHub App installed on your repository. This is functionally equivalent to the `GITHUB_TOKEN` secret. For more information, see "Automatic token authentication." Note: This context property is set by the Actions runner, and is only available within the execution `steps` of a job. Otherwise, the value of this property will be `null`.
`github.triggering_actor`	`string`	The username of the user that initiated the workflow run. If the workflow run is a re-run, this value may differ from `github.actor`. Any workflow re-runs will use the privileges of `github.actor`, even if the actor initiating the re-run (`github.triggering_actor`) has

		different privileges.
`github.workflow`	`string`	The name of the workflow. If the workflow file doesn't specify a `name`, the value of this property is the full path of the workflow file in the repository.
`github.workflow_ref`	`string`	The ref path to the workflow. For example, `octocat/hello-world/.github/workflows/my-workflow.yml@refs/heads/my_branch`.
`github.workflow_sha`	`string`	The commit SHA for the workflow file.
`github.workspace`	`string`	The default working directory on the runner for steps, and the default location of your repository when using the `checkout` action.

Example contents of the `github` context

The following example context is from a workflow run triggered by the `push` event. The `event` object in this example has been truncated because it is identical to the contents of the `push` webhook payload.

> **Note:** This context is an example only. The contents of a context depends on the workflow that you are running. Contexts, objects, and properties will vary significantly under different workflow run conditions.


```
{
  "token": "***",
  "job": "dump_contexts_to_log",
  "ref": "refs/heads/my_branch",
  "sha": "c27d339ee6075c1f744c5d4b200f7901aad2c369",
  "repository": "octocat/hello-world",
  "repository_owner": "octocat",
  "repositoryUrl": "git://github.com/octocat/hello-world.git",
  "run_id": "1536140711",
  "run_number": "314",
  "retention_days": "90",
  "run_attempt": "1",
  "actor": "octocat",
  "workflow": "Context testing",
  "head_ref": "",
  "base_ref": "",
  "event_name": "push",
  "event": {
    ...
  },
  "server_url": "https://github.com",
  "api_url": "https://api.github.com",
  "graphql_url": "https://api.github.com/graphql",
  "ref_name": "my_branch",
  "ref_protected": false,
  "ref_type": "branch",
  "secret_source": "Actions",
  "workspace": "/home/runner/work/hello-world/hello-world",
  "action": "github_step",
  "event_path": "/home/runner/work/_temp/_github_workflow/event.json",
```

```
  "action_repository": "",
  "action_ref": "",
  "path": "/home/runner/work/_temp/_runner_file_commands/add_path_b037e7b5-1c88-
48e2-bf78-eaaab5e02602",
  "env": "/home/runner/work/_temp/_runner_file_commands/set_env_b037e7b5-1c88-
48e2-bf78-eaaab5e02602"
}
```


Example usage of the `github` context

This example workflow uses the `github.event_name` context to run a job only if the workflow run was triggered by the `pull_request` event.

YAML

```yaml
name: Run CI
on: [push, pull_request]

jobs:
  normal_ci:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Run normal CI
        run: ./run-tests

  pull_request_ci:
    runs-on: ubuntu-latest
    if: ${{ github.event_name == 'pull_request' }}
    steps:
      - uses: actions/checkout@v4
      - name: Run PR CI
        run: ./run-additional-pr-ci
```

`env` context

The `env` context contains variables that have been set in a workflow, job, or step. It does not contain variables inherited by the runner process. For more information about setting variables in your workflow, see "Workflow syntax for GitHub Actions."

You can retrieve the values of variables stored in `env` context and use these values in your workflow file. You can use the `env` context in any key in a workflow step except for the `id` and `uses` keys. For more information on the step syntax, see "Workflow syntax for GitHub Actions."

If you want to use the value of a variable inside a runner, use the runner operating system's normal method for reading environment variables.

Property name	Type	Description
`env`	`object`	This context changes for each step in a job. You can access this context from any step in a job. This object contains the properties listed below.
`env.<env_name>`	`string`	The value of a specific environment variable.

Example contents of the `env` context

The contents of the `env` context is a mapping of variable names to their values. The context's contents can change depending on where it is used in the workflow run. In this example, the `env` context contains two variables.

```
{
  "first_name": "Mona",
  "super_duper_var": "totally_awesome"
}
```

Example usage of the `env` context

This example workflow shows variables being set in the `env` context at the workflow, job, and step levels. The `${{ env.VARIABLE-NAME }}` syntax is then used to retrieve variable values within individual steps in the workflow.

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

YAML

```yaml
name: Hi Mascot
on: push
env:
  mascot: Mona
  super_duper_var: totally_awesome

jobs:
  windows_job:
    runs-on: windows-latest
    steps:
      - run: echo 'Hi ${{ env.mascot }}'  # Hi Mona
      - run: echo 'Hi ${{ env.mascot }}'  # Hi Octocat
        env:
          mascot: Octocat
  linux_job:
    runs-on: ubuntu-latest
    env:
      mascot: Tux
    steps:
      - run: echo 'Hi ${{ env.mascot }}'  # Hi Tux
```

`vars` context

Note: Configuration variables for GitHub Actions are in beta and subject to change.

The `vars` context contains custom configuration variables set at the organization, repository, and environment levels. For more information about defining configuration variables for use in multiple workflows, see "Variables".

Example contents of the `vars` context

The contents of the `vars` context is a mapping of configuration variable names to their values.

```
{
```

```
  "mascot": "Mona"
}
```

Example usage of the `vars` context

This example workflow shows how configuration variables set at the repository, environment, or organization levels are automatically available using the `vars` context.

If a configuration variable has not been set, the return value of a context referencing the variable will be an empty string.

The following example shows using configuration variables with the `vars` context across a workflow. Each of the following configuration variables have been defined at the repository, organization, or environment levels.

YAML

```yaml
on:
  workflow_dispatch:
env:
  # Setting an environment variable with the value of a configuration variable
  env_var: ${{ vars.ENV_CONTEXT_VAR }}

jobs:
  display-variables:
    name: ${{ vars.JOB_NAME }}
    # You can use configuration variables with the `vars` context for dynamic
jobs
    if: ${{ vars.USE_VARIABLES == 'true' }}
    runs-on: ${{ vars.RUNNER }}
    environment: ${{ vars.ENVIRONMENT_STAGE }}
    steps:
    - name: Use variables
      run: |
        echo "repository variable : $REPOSITORY_VAR"
        echo "organization variable : $ORGANIZATION_VAR"
        echo "overridden variable : $OVERRIDE_VAR"
        echo "variable from shell environment : $env_var"
      env:
        REPOSITORY_VAR: ${{ vars.REPOSITORY_VAR }}
        ORGANIZATION_VAR: ${{ vars.ORGANIZATION_VAR }}
        OVERRIDE_VAR: ${{ vars.OVERRIDE_VAR }}

    - name: ${{ vars.HELLO_WORLD_STEP }}
      if: ${{ vars.HELLO_WORLD_ENABLED == 'true' }}
      uses: actions/hello-world-javascript-action@main
      with:
        who-to-greet: ${{ vars.GREET_NAME }}
```

`job` context

The `job` context contains information about the currently running job.

Property name	Type	Description
`job`	`object`	This context changes for each job in a workflow run. You can access this context from any step in a job. This object contains all the properties listed below.
`job.container`	`object`	Information about the job's

		container. For more information about containers, see "Workflow syntax for GitHub Actions."
`job.container.id`	`string`	The ID of the container.
`job.container.network`	`string`	The ID of the container network. The runner creates the network used by all containers in a job.
`job.services`	`object`	The service containers created for a job. For more information about service containers, see "Workflow syntax for GitHub Actions."
`job.services.<service_id>.id`	`string`	The ID of the service container.
`job.services.<service_id>.network`	`string`	The ID of the service container network. The runner creates the network used by all containers in a job.
`job.services.<service_id>.ports`	`object`	The exposed ports of the service container.
`job.status`	`string`	The current status of the job. Possible values are `success`, `failure`, or `cancelled`.

Example contents of the `job` context

This example `job` context uses a PostgreSQL service container with mapped ports. If there are no containers or service containers used in a job, the `job` context only contains the `status` property.

```
{
  "status": "success",
  "container": {
    "network": "github_network_53269bd575974817b43f4733536b200c"
  },
  "services": {
    "postgres": {
      "id": "60972d9aa486605e66b0dad4abb638dc3d9116f566579e418166eedb8abb9105",
      "ports": {
        "5432": "49153"
      },
      "network": "github_network_53269bd575974817b43f4733536b200c"
    }
  }
}
```

Example usage of the `job` context

This example workflow configures a PostgreSQL service container, and automatically maps port 5432 in the service container to a randomly chosen available port on the host. The `job` context is used to access the number of the port that was assigned on the host.

```
YAML
```

YAML

```yaml
name: PostgreSQL Service Example
on: push
jobs:
  postgres-job:
    runs-on: ubuntu-latest
    services:
      postgres:
        image: postgres
        env:
          POSTGRES_PASSWORD: postgres
        options: --health-cmd pg_isready --health-interval 10s --health-timeout
5s --health-retries 5
        ports:
          # Maps TCP port 5432 in the service container to a randomly chosen
available port on the host.
          - 5432

    steps:
      - uses: actions/checkout@v4
      - run: pg_isready -h localhost -p ${{ job.services.postgres.ports[5432] }}
      - run: ./run-tests
```

`jobs` context

The `jobs` context is only available in reusable workflows, and can only be used to set outputs for a reusable workflow. For more information, see "Reusing workflows."

Property name	Type	Description
`jobs`	`object`	This is only available in reusable workflows, and can only be used to set outputs for a reusable workflow. This object contains all the properties listed below.
`jobs.<job_id>.result`	`string`	The result of a job in the reusable workflow. Possible values are `success`, `failure`, `cancelled`, or `skipped`.
`jobs.<job_id>.outputs`	`object`	The set of outputs of a job in a reusable workflow.
`jobs.<job_id>.outputs.<output_name>`	`string`	The value of a specific output for a job in a reusable workflow.

Example contents of the `jobs` context

This example `jobs` context contains the result and outputs of a job from a reusable workflow run.

```
{
  "example_job": {
    "result": "success",
    "outputs": {
      "output1": "hello",
      "output2": "world"
    }
  }
```

```
}
```

Example usage of the `jobs` context

This example reusable workflow uses the `jobs` context to set outputs for the reusable workflow. Note how the outputs flow up from the steps, to the job, then to the `workflow_call` trigger. For more information, see "Reusing workflows."

YAML

```yaml
name: Reusable workflow

on:
  workflow_call:
    # Map the workflow outputs to job outputs
    outputs:
      firstword:
        description: "The first output string"
        value: ${{ jobs.example_job.outputs.output1 }}
      secondword:
        description: "The second output string"
        value: ${{ jobs.example_job.outputs.output2 }}

jobs:
  example_job:
    name: Generate output
    runs-on: ubuntu-latest
    # Map the job outputs to step outputs
    outputs:
      output1: ${{ steps.step1.outputs.firstword }}
      output2: ${{ steps.step2.outputs.secondword }}
    steps:
      - id: step1
        run: echo "firstword=hello" >> $GITHUB_OUTPUT
      - id: step2
        run: echo "secondword=world" >> $GITHUB_OUTPUT
```

`steps` context

The `steps` context contains information about the steps in the current job that have an `id` specified and have already run.

Property name	Type	Description
`steps`	`object`	This context changes for each step in a job. You can access this context from any step in a job. This object contains all the properties listed below.
`steps.<step_id>.outputs`	`object`	The set of outputs defined for the step. For more information, see "Metadata syntax for GitHub Actions."
`steps.<step_id>.conclusion`	`string`	The result of a completed step after `continue-on-error` is applied. Possible values are `success`, `failure`, `cancelled`, or `skipped`. When a `continue-on-error` step fails, the `outcome` is `failure`, but the

		final `conclusion` is `success`.
`steps.<step_id>.outcome`	`string`	The result of a completed step before `continue-on-error` is applied. Possible values are `success`, `failure`, `cancelled`, or `skipped`. When a `continue-on-error` step fails, the `outcome` is `failure`, but the final `conclusion` is `success`.
`steps.<step_id>.outputs.<output_name>`	`string`	The value of a specific output.

Example contents of the `steps` context

This example `steps` context shows two previous steps that had an `id` specified. The first step had the `id` named `checkout`, the second `generate_number`. The `generate_number` step had an output named `random_number`.

```
{
  "checkout": {
    "outputs": {},
    "outcome": "success",
    "conclusion": "success"
  },
  "generate_number": {
    "outputs": {
      "random_number": "1"
    },
    "outcome": "success",
    "conclusion": "success"
  }
}
```

Example usage of the `steps` context

This example workflow generates a random number as an output in one step, and a later step uses the `steps` context to read the value of that output.

YAML

```yaml
name: Generate random failure
on: push
jobs:
  randomly-failing-job:
    runs-on: ubuntu-latest
    steps:
      - id: checkout
        uses: actions/checkout@v4
      - name: Generate 0 or 1
        id: generate_number
        run:  echo "random_number=$(($RANDOM % 2))" >> $GITHUB_OUTPUT
      - name: Pass or fail
        run: |
          if [[ ${{ steps.generate_number.outputs.random_number }} == 0 ]]; then
exit 0; else exit 1; fi
```

`runner` context

The `runner` context contains information about the runner that is executing the current job.

Property name	Type	Description
`runner`	`object`	This context changes for each job in a workflow run. This object contains all the properties listed below.
`runner.name`	`string`	The name of the runner executing the job. This name may not be unique in a workflow run as runners at the repository and organization levels could use the same name.
`runner.os`	`string`	The operating system of the runner executing the job. Possible values are `Linux`, `Windows`, or `macOS`.
`runner.arch`	`string`	The architecture of the runner executing the job. Possible values are `X86`, `X64`, `ARM`, or `ARM64`.
`runner.temp`	`string`	The path to a temporary directory on the runner. This directory is emptied at the beginning and end of each job. Note that files will not be removed if the runner's user account does not have permission to delete them.
`runner.tool_cache`	`string`	The path to the directory containing preinstalled tools for GitHub-hosted runners. For more information, see "Using GitHub-hosted runners".
`runner.debug`	`string`	This is set only if debug logging is enabled, and always has the value of `1`. It can be useful as an indicator to enable additional debugging or verbose logging in your own job steps.

Example contents of the `runner` context

The following example context is from a Linux GitHub-hosted runner.

```
{
  "os": "Linux",
  "arch": "X64",
  "name": "GitHub Actions 2",
  "tool_cache": "/opt/hostedtoolcache",
  "temp": "/home/runner/work/_temp"
}
```

Example usage of the `runner` context

This example workflow uses the `runner` context to set the path to the temporary directory to write logs, and if the workflow fails, it uploads those logs as artifact.

YAML

```yaml
name: Build
on: push

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Build with logs
        run: |
          mkdir ${{ runner.temp }}/build_logs
          ./build.sh --log-path ${{ runner.temp }}/build_logs
      - name: Upload logs on fail
        if: ${{ failure() }}
        uses: actions/upload-artifact@v3
        with:
          name: Build failure logs
          path: ${{ runner.temp }}/build_logs
```

`secrets` context

The `secrets` context contains the names and values of secrets that are available to a workflow run. The `secrets` context is not available for composite actions due to security reasons. If you want to pass a secret to a composite action, you need to do it explicitly as an input. For more information about secrets, see "Using secrets in GitHub Actions."

`GITHUB_TOKEN` is a secret that is automatically created for every workflow run, and is always included in the `secrets` context. For more information, see "Automatic token authentication."

Warning: If a secret was used in the job, GitHub automatically redacts secrets printed to the log. You should avoid printing secrets to the log intentionally.

Property name	Type	Description
`secrets`	`object`	This context is the same for each job in a workflow run. You can access this context from any step in a job. This object contains all the properties listed below.
`secrets.GITHUB_TOKEN`	`string`	Automatically created token for each workflow run. For more information, see "Automatic token authentication."
`secrets.<secret_name>`	`string`	The value of a specific secret.

Example contents of the `secrets` context

The following example contents of the `secrets` context shows the automatic

`GITHUB_TOKEN`, as well as two other secrets available to the workflow run.

```
{
  "github_token": "***",
  "NPM_TOKEN": "***",
  "SUPERSECRET": "***"
}
```

Example usage of the `secrets` context

This example workflow uses the labeler action, which requires the `GITHUB_TOKEN` as the value for the `repo-token` input parameter:

YAML

```yaml
name: Pull request labeler
on: [ pull_request_target ]

jobs:
  triage:
    runs-on: ubuntu-latest
    permissions:
      contents: read
      pull-requests: write
    steps:
      - uses: actions/labeler@v4
        with:
          repo-token: ${{ secrets.GITHUB_TOKEN }}
```

`strategy` context

For workflows with a matrix, the `strategy` context contains information about the matrix execution strategy for the current job.

Property name	Type	Description
`strategy`	`object`	This context changes for each job in a workflow run. You can access this context from any job or step in a workflow. This object contains all the properties listed below.
`strategy.fail-fast`	`boolean`	When this evaluates to `true`, all in-progress jobs are canceled if any job in a matrix fails. For more information, see "Workflow syntax for GitHub Actions."
`strategy.job-index`	`number`	The index of the current job in the matrix. **Note:** This number is a zero-based number. The first job's index in the matrix is `0`.
`strategy.job-total`	`number`	The total number of jobs in the matrix. **Note:** This number **is not** a zero-based number. For example, for a matrix with four jobs, the value of `job-total` is

		`4`.
`strategy.max-parallel`	`number`	The maximum number of jobs that can run simultaneously when using a `matrix` job strategy. For more information, see "Workflow syntax for GitHub Actions."

Example contents of the `strategy` context

The following example contents of the `strategy` context is from a matrix with four jobs, and is taken from the final job. Note the difference between the zero-based `job-index` number, and `job-total` which is not zero-based.

```
{
  "fail-fast": true,
  "job-index": 3,
  "job-total": 4,
  "max-parallel": 4
}
```

Example usage of the `strategy` context

This example workflow uses the `strategy.job-index` property to set a unique name for a log file for each job in a matrix.

YAML

```yaml
name: Test matrix
on: push

jobs:
  test:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        test-group: [1, 2]
        node: [14, 16]
    steps:
      - uses: actions/checkout@v4
      - run: npm test > test-job-${{ strategy.job-index }}.txt
      - name: Upload logs
        uses: actions/upload-artifact@v3
        with:
          name: Build log for job ${{ strategy.job-index }}
          path: test-job-${{ strategy.job-index }}.txt
```

`matrix` context

For workflows with a matrix, the `matrix` context contains the matrix properties defined in the workflow file that apply to the current job. For example, if you configure a matrix with the `os` and `node` keys, the `matrix` context object includes the `os` and `node` properties with the values that are being used for the current job.

There are no standard properties in the `matrix` context, only those which are defined in the workflow file.

Property name	Type	Description

`matrix`	`object`	This context is only available for jobs in a matrix, and changes for each job in a workflow run. You can access this context from any job or step in a workflow. This object contains the properties listed below.
`matrix.<property_name>`	`string`	The value of a matrix property.

Example contents of the `matrix` context

The following example contents of the `matrix` context is from a job in a matrix that has the `os` and `node` matrix properties defined in the workflow. The job is executing the matrix combination of an `ubuntu-latest` OS and Node.js version `16`.

```
{
  "os": "ubuntu-latest",
  "node": 16
}
```

Example usage of the `matrix` context

This example workflow creates a matrix with `os` and `node` keys. It uses the `matrix.os` property to set the runner type for each job, and uses the `matrix.node` property to set the Node.js version for each job.

YAML

```yaml
name: Test matrix
on: push

jobs:
  build:
    runs-on: ${{ matrix.os }}
    strategy:
      matrix:
        os: [ubuntu-latest, windows-latest]
        node: [14, 16]
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v3
        with:
          node-version: ${{ matrix.node }}
      - name: Install dependencies
        run: npm ci
      - name: Run tests
        run: npm test
```

`needs` context

The `needs` context contains outputs from all jobs that are defined as a direct dependency of the current job. Note that this doesn't include implicitly dependent jobs (for example, dependent jobs of a dependent job). For more information on defining job dependencies, see "Workflow syntax for GitHub Actions."

Property name	Type	Description
`needs`	`object`	This context is only populated

`needs`	`object`	This context is only populated for workflow runs that have dependent jobs, and changes for each job in a workflow run. You can access this context from any job or step in a workflow. This object contains all the properties listed below.
`needs.<job_id>`	`object`	A single job that the current job depends on.
`needs.<job_id>.outputs`	`object`	The set of outputs of a job that the current job depends on.
`needs.<job_id>.outputs.<output name>`	`string`	The value of a specific output for a job that the current job depends on.
`needs.<job_id>.result`	`string`	The result of a job that the current job depends on. Possible values are `success`, `failure`, `cancelled`, or `skipped`.

Example contents of the `needs` context

The following example contents of the `needs` context shows information for two jobs that the current job depends on.

```
{
  "build": {
    "result": "success",
    "outputs": {
      "build_id": "ABC123"
    }
  },
  "deploy": {
    "result": "failure",
    "outputs": {}
  }
}
```

Example usage of the `needs` context

This example workflow has three jobs: a `build` job that does a build, a `deploy` job that requires the `build` job, and a `debug` job that requires both the `build` and `deploy` jobs and runs only if there is a failure in the workflow. The `deploy` job also uses the `needs` context to access an output from the `build` job.

YAML

```yaml
name: Build and deploy
on: push

jobs:
  build:
    runs-on: ubuntu-latest
    outputs:
      build_id: ${{ steps.build_step.outputs.build_id }}
    steps:
      - uses: actions/checkout@v4
      - name: Build
```

```yaml
        id: build_step
        run: |
          ./build
          echo "build_id=$BUILD_ID" >> $GITHUB_OUTPUT
  deploy:
    needs: build
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - run: ./deploy --build ${{ needs.build.outputs.build_id }}
  debug:
    needs: [build, deploy]
    runs-on: ubuntu-latest
    if: ${{ failure() }}
    steps:
      - uses: actions/checkout@v4
      - run: ./debug
```

`inputs` context

The `inputs` context contains input properties passed to an action, to a reusable workflow, or to a manually triggered workflow. For reusable workflows, the input names and types are defined in the `workflow_call` event configuration of a reusable workflow, and the input values are passed from `jobs.<job_id>.with` in an external workflow that calls the reusable workflow. For manually triggered workflows, the inputs are defined in the `workflow_dispatch` event configuration of a workflow.

The properties in the `inputs` context are defined in the workflow file. They are only available in a reusable workflow or in a workflow triggered by the `workflow_dispatch` event

Property name	Type	Description
`inputs`	`object`	This context is only available in a reusable workflow or in a workflow triggered by the `workflow_dispatch` event. You can access this context from any job or step in a workflow. This object contains the properties listed below.
`inputs.<name>`	`string` or `number` or `boolean` or `choice`	Each input value passed from an external workflow.

Example contents of the `inputs` context

The following example contents of the `inputs` context is from a workflow that has defined the `build_id`, `deploy_target`, and `perform_deploy` inputs.

```json
{
  "build_id": 123456768,
  "deploy_target": "deployment_sys_1a",
  "perform_deploy": true
}
```

Example usage of the `inputs` context in a reusable workflow

This example reusable workflow uses the `inputs` context to get the values of the `build_id`, `deploy_target`, and `perform_deploy` inputs that were passed to the reusable

workflow from the caller workflow.

YAML

```yaml
name: Reusable deploy workflow
on:
  workflow_call:
    inputs:
      build_id:
        required: true
        type: number
      deploy_target:
        required: true
        type: string
      perform_deploy:
        required: true
        type: boolean

jobs:
  deploy:
    runs-on: ubuntu-latest
    if: ${{ inputs.perform_deploy }}
    steps:
      - name: Deploy build to target
        run: deploy --build ${{ inputs.build_id }} --target ${{
inputs.deploy_target }}
```

Example usage of the `inputs` context in a manually triggered workflow

This example workflow triggered by a `workflow_dispatch` event uses the `inputs` context to get the values of the `build_id`, `deploy_target`, and `perform_deploy` inputs that were passed to the workflow.

YAML

```yaml
on:
  workflow_dispatch:
    inputs:
      build_id:
        required: true
        type: string
      deploy_target:
        required: true
        type: string
      perform_deploy:
        required: true
        type: boolean

jobs:
  deploy:
    runs-on: ubuntu-latest
    if: ${{ inputs.perform_deploy }}
    steps:
      - name: Deploy build to target
        run: deploy --build ${{ inputs.build_id }} --target ${{
inputs.deploy_target }}
```