Actions Cheat Sheet

GitHub Actions give you the flexibility to build automated software development lifecycle workflows. You can write individual tasks, called actions, and combine them to create custom workflows in your repository. GitHub Actions are automated processes allowing you to build, test, package, release, or deploy any code project on GitHub, but you can also use them to automate any step of your workflow: merging pull requests, assigning labels, triaging issues to name a few.

!

!

Workflow Syntax

Workflow files use YAML syntax, and must have either a .yml or .yaml file extension. You must store workflow files in the `.github/workflows/` directory of your repository.

```
name: Node CI
on: push
  branches:
    - 'releases/*'
jobs:
  my_build:
    runs-on: ubuntu-latest
    needs: my-other-job
    steps:
    - uses: actions/checkout@master
    - name: Say something
      run: echo "A little bit more action"
```

Workflow `name`

The name of your workflow. GitHub displays the names of your workflows on your repository's actions page.

`on` Event

The name of the GitHub event that triggers the workflow. You can provide a single event string, an array of events, or an event configuration map that restricts the execution of a workflow:

- When using the `push` and `pull_request` events, `branches` and `tags` allow to select or exclude references the workflow will run on, while `paths` specifies which files must have been modified in order to run the workflow.
- The `types` keyword enables you to narrow down activity that causes the workflow to run: `created`, `edited`, `deleted`… A workflow trigger can also be scheduled:

```
on:
  schedule:
  - cron: '*/15 * * * *'
```

`jobs` Collection

A workflow run is made up of one or more jobs identified by a `job_id`. Jobs run in parallel by default. Each job runs in a fresh instance of the virtual environment specified by `runs-on`.

`job_id`

The name of the job displayed on GitHub. For example `my_build` or `build`. This string is unique with the `jobs` object.

!

`needs`

Identifies any job that must complete successfully before this job will run. It can be a string or array of strings. If a job fails, all jobs that need it are skipped unless the jobs use a conditional statement that causes the job to continue.

`runs-on`

Inside of the `jobs` collection, this is the type of virtual host machine to run the job on. Available virtual machine types are:

- `ubuntu-latest`, `ubuntu-18.04`, or `ubuntu-16.04`
- `windows-latest`, `windows-2019`, or `windows-2016`
- `macOS-latest` or `macOS-10.14`

`container`

Instead of running directly on a host selected with `runs-on`, a container can run any steps in a job that don't already specify a container. If you have steps that use both script and container actions, the container actions will run as sibling containers on the same network with the same volume mounts. This object has the following attributes: `image`, `env`, `ports`, `volume` and `options`.

```
...
jobs:
  my_job:
    container:
      image: node:10.16-jessie
      env:
        NODE_ENV: development
      ports:
        - 80
      volumes:
        - my_docker_volume:/volume_mount
      options: --cpus 1
```

`services`

Additional containers to host services for a job in a workflow. These are useful for creating databases or cache services. The runner on the virtual machine will automatically create a network and manage the lifecycle of the service containers. Each service is a named object in the `services` collection (`redis` or `nginx` for example) and can receive the same parameters than the `container` object.

`timeout-minutes`

The maximum number of minutes to let a workflow run before GitHub automatically cancels it. Default: 360

!

GitHub

Actions Cheat Sheet

!

!

Job `steps`

A job contains a sequence of tasks called `steps`. Steps can run commands, run setup tasks, or run an action from your repository, a public repository, or an action published in a Docker registry. Each step runs in its own process in the virtual environment and has access to the workspace and filesystem.

Step `name`

Specify the label to be displayed for this step in GitHub. It's not required but does improve readability in the logs.

`uses`

Select an action to run as part of a step in your job. You can use an action defined in the same repository as the workflow, a public repository elsewhere on GitHub, or in a published Docker container image.

`with`

A map of the input parameters defined by the action. Each input parameter is a key/value pair. Input parameters are prefixed with `INPUT_`, converted to upper case and set as environment variables: the `first_name` parameter will be exposed as the `INPUT_FIRST_NAME` environment variable. Special parameter names are: - `args`, a string that defines the inputs passed to a Docker container's `ENTRYPOINT`. It is used in place of the `CMD` instruction in a `Dockerfile`. - `entrypoint`, a string that defines or overrides the executable to run as the Docker container's `ENTRYPOINT`.

`env`

Sets environment variables for steps to use in the virtual environment. If you are setting a secret in an environment variable, you must use the `secrets` context.

```
...
steps:
  - name: A cool action
    uses: actions/bin/sh@master
    with:
      entrypoint: /bin/echo
      args: The ${{ github.event_name }} did it
    env:
      last_name: ${{secrets.LASTNAME}}
```

Default environment variables:

- `HOME`: Path to the GitHub home directory used to store user data.
- `GITHUB_WORKFLOW`: Name of the workflow.
- `GITHUB_ACTION`: Name of the action.
- `GITHUB_ACTOR`: Name of the person or app that initiated the workflow.
- `GITHUB_REPOSITORY`: Owner and repository name.
- `GITHUB_EVENT_NAME`: Name of the triggering webhook event.
- `GITHUB_EVENT_PATH`: Path of the file with the webhook event payload.
- `GITHUB_WORKSPACE`: GitHub workspace directory path.
- `GITHUB_SHA`: Commit SHA that triggered the workflow.
- `GITHUB_REF`: Branch or tag ref that triggered the workflow.
- `GITHUB_HEAD_REF`: Branch of the head repository (forks only).
- `GITHUB_BASE_REF`: Branch of the base repository (forks only).

Default secrets:

- `GITHUB_TOKEN`: Token you can leverage for API calls or git commands.

!

`run`

Instead of running an existing action, a command line program can be run using the operating system's shell. Each run keyword represents a new process and shell in the virtual environment. A specific shell can be selected with the `shell` attribute.

`if`

Prevents a step from running unless a condition is met. The value is an expression without the `${{ … }}` block

Job `strategy`

A build matrix strategy is a set of different configurations of the virtual environment. The job' set of steps will be executed on each of these configurations. The following exemple specifies 3 nodejs versions on 2 operating systems:

```
runs-on: ${{ matrix.os }}
strategy:
  matrix:
    os: [ubuntu-16.04, ubuntu-18.04]
    node: [6, 8, 10]
steps:
  - uses: actions/setup-node@v1
    with:
      node-version: ${{ matrix.node }}
```

`fail-fast`

When set to `true` (default value), GitHub cancels all in-progress jobs if any of the matrix job fails.

Context and expressions

Expressions can be used to programmatically set variables in workflow files and access contexts. An expression can be any combination of literal values, references to a context, or functions. You can combine literals, context references, and functions using operators. With the exception of the `if` key, expressions are written in a `${{ … }}` block.

Contexts

Contexts are a way to access runtime information. The following objects are available: `github`, `job`, `steps`, `runner`, `secrets`, `strategy`, `matrix`.

Functions

Functions `contains`, `startsWith`, `endsWith` with arguments `(searchString, searchValue)`: return true if searchString respectively contains, starts or ends with searchValue. These functions are not case sensitive. Casts values to a string.

`format(string, replaceValue0, …, replaceValueN)`: replaces values (specified using the {N} syntax, where N is an integer) in the `string`, with the variable `replaceValueN`.

`join(element, optionalElem)`: all values in `element` (an array or a string) are concatenated into a string. `optionalElem` is appended to the end of `element`.

`toJSON(value)`: returns a pretty-print JSON representation of `value`.

`success()`, `always()`, `failure()` and `cancelled()`: these status check functions can be used as expressions in `if` conditionals.