GitHub

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. Each different YAML file corresponds to a different workflow.

```
name: My Workflow
on:
  push:
    branches:
      - 'releases/*'
      - '!releases/**-alpha'
env:
  message: 'conversation'
  my_token: ${{ secrets.GITHUB_TOKEN }}
jobs:
  my_build:
    runs-on: ubuntu-latest
    steps:
      - name: Checking out our code
        uses: actions/checkout@master
      - name: Say something
        run: |
          echo "A little less ${message}"
          echo "A little more action"
  my_job:
    needs: my_build
    container:
      image: node:10.16-jessie
      env:
        NODE_ENV: development
      ports:
        - 80
      volumes:
        - my_docker_volume:/volume_mount
      options: --cpus 1
    services:
      redis:
        image: redis
        ports:
          - 6379/tcp
```

Workflow `name`

The name of your workflow will be displayed on your repository's actions page.

Workflow, Job or Step `env`

A map of environment variables which can be set at different scopes. Several environment variables are available by default (`GITHUB_EVENT_NAME`, `GITHUB_EVENT_PATH`, `GITHUB_SHA`, `GITHUB_REF`, `HOME`...) as well as a secret, `GITHUB_TOKEN`, which you can leverage for API calls or git commands through the `secrets` context.

`on` Event

The type 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 (with the `!` prefix) git references the workflow will run on, while `paths` specifies which files must have been modified in order to run the workflow.
- If your rules are only made of exclusions, you can use `branches-ignore`, `tags-ignore` and `paths-ignore`. The `-ignore` form and its inclusive version cannot be mixed.
- The `types` keyword enables you to narrow down activities (`opened`, `created`, `edited`...) causing the workflow to run. The list of available activities depends on the event.
- 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 unique `job_id` (`my_build` or `my_job`). Jobs run in parallel by default unless queued with the `needs` attribute. Each job runs in a fresh instance of the virtual environment specified by `runs-on`.

Job `name`

The name of the job displayed on GitHub.

`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`

The type of virtual host machine to run the job on. Can be either a GitHub or self-hosted runner. Jobs can also run in user-specified containers (see: `container`). Available GitHub-hosted virtual machine types are `ubuntu-latest`, `windows-latest`, `macOS-latest` plus some other specific versions for each operating system, in the form of `ubuntu-xx.xx`, `macOS-xx.xx` or `windows-xxxx`. To specify a self-hosted runner for your job, configure `runs-on` in your workflow file with self-hosted runner labels. Example: `[self-hosted, linux]`.

`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`.

`timeout-minutes`

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

`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.

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`

Specify 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. Including the version of the action you are using by specifying a Git ref, branch, SHA, or Docker tag is strongly recommended:

- `uses: {owner}/{repo}@{ref}` for actions in a public repository
- `uses: {owner}/{repo}/{path}@{ref}` for actions in a subdirectory of a public repository
- `uses: ./path/to/dir` for actions in a a subdirectory of the same repository
- `uses: docker://{image}:{tag}` for actions on Docker Hub
- `uses: docker://{host}/{image}:{tag}` for actions in a public registry

`with`

A map of the input parameters defined by the action in its `action.yml` file. When the acion is container based, 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`.

`if`

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

`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. Multiple commands can be run in a single shell instance using the `|` (pipe) operator.

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.