Continuous integration with GitHub Actions, GitLab pipelines and ci.inria.fr

Thierry Martinez (SED)

Willow software developper meetup

27 July 2022

Tentative for a cartography of continuous integration platforms

GitHub Actions

(launched on 16 October 2018)

- A platform for GitHub projects, providing GitHub-hosted runners for Linux, MacOS and Windows.
- File-based workflow specification: `.github/workflows/*.yml`. A command-line tool, `act` is available for running workflows locally (or from other continuous integration platforms): https://github.com/nektos/act
- Very easy to extend (new reusable actions can be defined in git repositories), Linux runners can run docker containers, user-provided runners can be used.

GitLab Pipelines

(introduced in GitLab 8.8.0, on 22 May 2016)

- Continuous integration system integrated in GitLab. gitlab.inria.fr provides shared docker runners, ci.inria.fr can host user-maintained VMs, and self-hosted GitLab runners can be registered.
- File-based workflow specification: `.gitlab.yml`. A command-line tool, `gitlab-ci-local` is available for running workflows locally (or from other continuous integration platforms): https://github.com/firecow/gitlab-ci-local
- The supported YAML syntax is richer than GitHub Actions (support for anchors, extends and file inclusion to reuse parts of code), but there is no built-in supports for reusable actions comparable with what GitHub Actions proposes.

ci.inria.fr

- A cloud of virtual machines dedicated for continuous integration: VMs are created and maintained by the users. Linux, Windows and MacOS VMs are supported, but MacOS VMs are quite fragile for the moment (but we are working on it!).
- Provides yet another continuous integration system, Jenkins, and a part of it can be configured by a file-based workflow specification: `Jenkinsfile`, difficult to run locally (except by installing and maintaining a local instance of Jenkins...).
- Very extensible through plugins but fragile and hard to maintain. Most plugins should be configured via the web interface; there is an API covering many operations, but it is not well documented and there is no easy tool to call it (excepting `curl`!).

A primer on GitHub Action

Repository initialization

- We will use GitHub command line: https://cli.github.com/
- GitHub Actions run every workflow specified in files `.github/workflows/*.yml` in a GitHub repository.
- `gh repo create github-actions-primer --public --clone`
- Put some contents in `github-actions-primer/.github/workflows/main.yml`

```yaml
name: main
on: [push]
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Preparing the environment
        run: |
          <enter some shell commands>
```

See Choosing GitHub hosted runners for a list of available platforms for `runs-on` entry. Note that `ubuntu-latest` is currently Ubuntu 20.04. There is `ubuntu-22.04` available in beta.

- Use `gh run list` to check the status of workflow runs on the command-line.

Run workflow locally

```
$ act
```

For platforms not supported out-of-box, one can provide a Docker image: for instance, to support `ubuntu-22.04`

```
act -P ubuntu-22.04=local-ubuntu-22.04
```

where `local-ubuntu-22.04` is a tag for an image built with a `Dockerfile` such as

```
FROM ubuntu:22.04
RUN apt-get update
RUN apt-get upgrade --yes
RUN apt-get install --yes sudo curl psmisc
```

> ⚠️ Using versions of Ubuntu 21.10 in Docker images requires Docker 20.10.9 (issue with syscall `clone3`).

> ⚠️ GitHub-hosted runners reduce interactions much more than `act` knows to do locally: think about adding options `--yes` and passing `DEBIAN_FRONTEND=noninteractive` in `apt-get` environment...

Running jobs in a container

Build environments can be prepared once for all in a Docker image to reduce build times:

- `docker build -t ghcr.io/‹user›/‹image name› .`
- create a personal access token with scope `write:packages`, save it in a file
- `docker login ghcr.io -u ‹user› --password-stdin < ‹token path›`
- `docker push ghcr.io/‹user›/‹image name›`
- create a personal access token with scope `read:packages`, store it in a secret (using `gh secret set`)
- reference the container in the job

```
    container:
      image: ghcr.io/‹user›/‹image name›
      credentials:
        username: ${{ github.actor }}
        password: ${{ secrets.‹secret name› }}
```

- to run the workflow locally, use `act --secret-file ‹file name›`

Use a job to build the environment

- Store a personal access token] with scope `write:packages` in a secret.
- Check out the repository! Add the following action

```
      - name: Checkout
        uses: actions/checkout@v3
```

- Steps for `docker build` and `docker push`.

> ⚠️ `checkout` action wipes out the current directory!
> Should be run before any actions writing useful things in it (local setup, etc.).

Run a job only if a file has changed

- Checkout with the input `fetch-depth: 2` to get the two last commits (by default, only the last commit is checked out, *i.e.* `git fetch --depth=1`)
- Use `git diff --quiet --exit-code HEAD^ HEAD -- ‹path›` to check if a file changed.

> ⚠️ Commands should succeed (with return code 0). Use `if-then-else-fi` to control the result of `git diff`.

- Can be done in another job, using job outputs and conditions.
- We only want to build the image if `Dockerfile` has changed, but the main job should be run even if the build job has been skipped: use `always()` and check `needs.‹job_id›.result` for `success` or `skipped`.

Using artifacts and deploy release

- Storing workflow data as artifacts: `actions/upload-artifact@v3` with inputs `name` and `path`, `actions/download-artifact@v3` with input `name`.
- softprops/action-gh-release with input `files`

Some notes on Windows runner

- `windows-latest` runners come with `choco` and some pre-installed tools, such as `7z`.

Adding self-hosted runners

- In Project Settings › Actions › Runners, button *New self-hosted runner*. Follow the instructions. Tags match the values of `runs-on:` field.
- `./run.sh` can be run in `tmux` or as a service.

Matrix job

- Use `strategy.matrix` to build the same job with different combinations of parameters.
- Set `strategy.fail-fast: false` to continue the build of other combinations when a combination failed.

Creating custom actions

- 3 types of actions can be created: Docker, Javascript and Composite
- Docker actions can only be used on Linux runners
- Composite actions combines multiple workflow steps in a single action

Describing an action

- An action is described by a single `action.yml` file
- One can define the inputs, outputs end environnement variables of an action
- If the action is designed to be reusable and public, use a dedicated public repository for the action
  See [`publishing on GitHub Markerplace`]:
  https://docs.github.com/en/actions/creating-actions/publishing-actions-in-github-marketplace
- If the action is local to a repository, place the yml file in
  `.github/actions/‹action name›/action.yml`
- Local actions are used in a workflow as follows:

```
    - name: Run docker custom action
      uses: ./.github/actions/local-action
      ...
```

Example: a Docker action

-
-
- The Docker action can use a `Dockerfile` => GitHub will build the image when the action is run
-

Example: a Javascript action

-
- local javascript actions has the drawback of requiring `node_modules/` to be there => clutters the repository
-
-