GitHub Actions Tutorial

This guide provides a brief overview of how to create a CI system for your project using GitHub Actions. While this document contains some of the basic steps needed to create a CI system in GitHub, we strongly recommend reading the official documentation provided by GitHub for additional details and information on the latest features available in GitHub Actions.

I. What are GitHub Actions?

GitHub Actions is a tool that can be used to automate the process of compiling, testing, and deploying code. Actions are event-driven, which means that they are triggered once an event has occurred. Examples of events are pushes and pull requests to a GitHub repository.

II. Components of a GitHub Action[1]

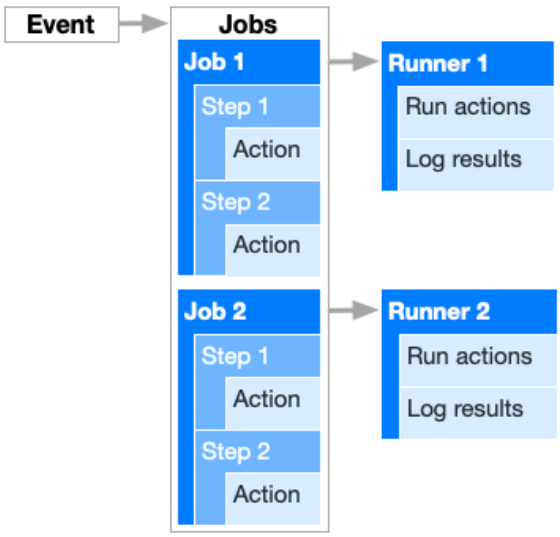

Figure 1: This diagram shows the different components of a workflow and how they relate to each other. Workflows are triggered by one or more events, and they are made of one or more jobs. Each job contains one or more steps which are individual tasks made of actions. As can be seen in the diagram each job is run in parallel in separate runners that execute each step of a job and report back the logs and results to GitHub.

[1] The below descriptions are based on GitHub's "Understanding GitHub Actions" documentation. For more information, we suggest reading the documentation here.

Workflows

Workflows are automated procedures that can be added to repositories. The simplest way to think of a “Workflow” is that it is a configuration script which defines a set of tasks to execute via GitHub Actions.

Workflows are comprised of one or more jobs that can be scheduled or triggered by an event. They can be used to automatically build, test, release, or even deploy a project via GitHub Actions.

Workflow files are written in YAML syntax and are stored in the .github/workflows folder within a project repository. Additionally, the workflow information can be accessed from the *Actions* tab on the homepage of a repository.

Importantly, workflows can be reused within and between repositories. That is, a new workflow can actually call and run an existing workflow (either from the same repository or another repository). This is helpful to maintain code readability and reduce unnecessary redundancy.

The GitHub community has a rich set of ready-to-use workflows that can be plugged into a new workflow to automate simple tasks (see examples in the GitHub Marketplace). Should you wish to create your own reusable workflow, more information can be found here.

Events

An event is a specific activity that triggers a workflow. These activities can be GitHub internal events (e.g., pushes or pull requests to a repository) or they can be external events (e.g., an API call). While we did not use external events to trigger workflows in our project, more information about them can be found in the GitHub documentation.

A workflow can be triggered by one or more events, which are specified within the workflow’s .yml file. A few examples of events include:

- Pushes
- Pull Requests
- Workflow Dispatch (manually clicking a button via the repository’s actions tab)

For many events, you can provide further specifications on exactly how the event should be triggered. For example, you can trigger the workflow to run only on pushes to a specific branch and pull requests to all branches within the repository.

Events can also be **scheduled** to trigger workflows at a specific time and date using POSIX Cron syntax. Scheduled workflows run on the last commit in the default branch and are restricted to run with at least 5 minutes of separation between them. More information on scheduled events can be found here.

Jobs

A Job is a set of steps that are executed on the same runner. Workflows can be made of one or more jobs. Workflows with multiple jobs run each job in parallel by default. However, if needed, jobs can be customized to run sequentially.

An important item to note about “jobs” is that they run on separate runners (machines). Therefore, sharing files between jobs requires special configuration, which is explained here.

Steps

Steps are individual tasks within a job that can run specific commands. A step can either be an action or a set of shell commands. All steps within a job are run on the same runner and are normally run sequentially. This allows the subsequent steps within an action to reuse any files/data created by a previous step.

Importantly, by default, if one step in a job fails, all subsequent steps will not be run. However, if you want some steps to always run, even if a previous step fails, this default setting can be changed by adding “if: always()” within a step (see this Stack Overflow thread for more details).

Actions

Actions are the smallest building block of workflows. One can create their own actions or reuse those made available by the GitHub community.

The simplest way to think of an action is as a reusable unit of code. As an example, the *actions/checkout@v2* is a reusable action provided by the GitHub community that handles cloning a repository as part of a workflow. For more examples of how an action can be used within a step of a job, see the documentation here.

Runners

Runners are servers that have the GitHub Actions runner application installed. Runners listen for available jobs, run one job at a time, and report the progress, logs, and results back to the GitHub UI.

Workflows can use GitHub hosted runners (which are VMs that run in Azure) or self-hosted runners (explained in the Runners section).

III. Adding Workflows to Your Project[2]

There are two different types of workflows that one can add to a GitHub project:

- The first one is using workflow templates that are available in the *Actions* tab. One can select a template and modify it according to the needs and environment of the project.
- The second way (and the one we used for our project) is setting up a new, custom workflow.

To set up a new workflow, go to the Actions tab and select either one of the templates provided or “Set up a workflow yourself” (to create a custom workflow). This will create a YAML file (.yml) under the .github/workflows folder. An example workflow file can be found here.

Simple Workflow Example

```yaml
name: learn-github-actions
on: [push]
jobs:
  check-bats-version:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - uses: actions/setup-node@v2
        with:
          node-version: '14'
      - run: npm install -g bats
      - run: bats -v
```

name:

- This is the name of the workflow (and will be used in the Actions tab to identify this specific workflow)

on:

- The “on” keyword specifies the events that will trigger the workflow. Here one can also specify the branch, path, or tag which the workflow should be run. More information about including multiple events with configurations in the YAML file can be found here

jobs:

- This keyword is used to specify the different jobs within a workflow. In the example above there is one job named “check-bats-version”. There can be multiple jobs

runs-on:

- This keyword configures the type of runner on which the job will run. In this case, the workflow will run on a Git-Hub hosted runner with the latest Ubuntu OS. More information on runners is explained in the Runners section.

steps:

[2] See https://docs.github.com/en/actions/quickstart

- This section groups all the steps (commands/actions) that run sequentially in a specific job. In this case, the job calls two pre-built two actions (checkout@v2 and setup-node@v2) and runs two shell commands.

uses:

- This keyword is used to call any action that is needed for the job to run. The first one 'actions/checkout@v2' is a public community action that checks out your repository and downloads it to the runner.. The 'actions/setup-node@v2' downloads the node software package on the runner, which is needed to use the 'npm' command.

run:

- This keyword is used to execute shell commands on the runner. These can be commands to install a package (like npm install -g bats), run a program (./name_of_the_script), or even compile a program (via make), etc.
- While not shown in the above example, multiple commands can be run within a single "run" block with the following syntax:

```
run: |
    command1
    command2
    command3
```

IV. Managing Workflows

Viewing workflow activity

- To see what workflows have been created and run, navigate to your repository and click *Actions* in the top bar.
- Here you can view all your workflows that have been executed, with a history of each past run.
- Each workflow run can be clicked to view more detailed information about the specific run, including runtime for each job, which jobs/steps passed or failed, and all output that has been printed to console.
- By clicking the "gear symbol" in the top right corner of a specific run you can also view the raw, unformatted logs from the run.
- On this page, workflows may also be run manually or re-run. If needed, previous runs of a workflow can also be deleted from the history.

Enabling/Disabling Workflows

Workflows can be manually enabled or disabled. A disabled workflow will no longer run (even when it should be triggered by an event). To disable a workflow, simply click on the specific workflow in the left column of the *Actions* tab in your repository, then click the three dots in the upper right corner of the workflow, and select disable workflow. The workflow may be re-enabled from this same page at any point.

V. Runners

GitHub-Hosted Runners

A GitHub-hosted runner is a VM created and managed by GitHub (within Azure) to run the workflows. A GitHub-hosted runner provides a new, clean VM for every workflow run.

The specific type of machine on which a workflow runs can be specified in the workflow using “runs on: “ and will provide various preinstalled software, passwordless sudo, and consistent file system setups.

A complete list of GitHub-hosted runners may be found here. If more software is needed, details of how they can be installed using your workflow can be found here: Customizing GitHub-hosted runners. Usage limits for GitHub-hosted runners can also be found here: Usage limits, billing, and administration.

Self-Hosted Runners

Instead of using GitHub-provided machines, one can install the runner application on their own machine to communicate with GitHub to receive workflows. In this way, the machine that the workflow runs on can use specific hardware, OS, and other software that is set-up by the user on this machine. Setup instructions for self-hosted runners can be found here.

While self-hosted runners provide more flexibility, they can also pose certain security risks, especially for public repositories. In addition, they also require the user to perform any maintenance themselves (e.g., deleting files after a workflow runs).

VI. Simple Tips for Using GitHub Actions

Below, we provide a few simple tips for using GitHub Actions based on our collective experience in using the system to develop a CI for the Linux Unikernel project.

Printing to Console

When using GitHub-hosted runners, not having direct access to the machine can make debugging very difficult (especially for errors related to incorrect file paths, non-existent tools, system configurations, etc.). To help debug such problems, we suggest making use of the UI and logs provided for each workflow run in the *Actions* tab which reports.

Within the workflow YAML file, one can add print statements (such as executing commands such as `pwd`, `cat`, or even printing to standard out directly from a program) that will then be displayed on the UI. You can also redirect the output of any task or program to a file and then print the file to the console to view detailed logs. Then, in the Actions tab, one can select a specific workflow run and look at the output for specific jobs and steps to understand where things might not be behaving as expected.

Cloning Repositories

When cloning repositories, we highly recommend using the Checkout V2 action. This action is enabled with the following syntax: `uses: actions/checkout@v2`. One can also specify different options for this checkout action (such as checking out a specific branch/commit or cloning a repository into a specific subfolder). More information on the checkout action usage can be found in the Action's README.md file.

There is one important item that should be noted with regards to using the checkout Action (or cloning in general) within a workflow. By default, the first first repository that is checked out will be set as the "GITHUB_WORKSPACE" environment variable, which becomes the top-level directory in the runner. Each "step" within a workflow automatically begins in the "GITHUB_WORKSPACE," so to run commands in any subdirectories, you need to first "cd" into that subdirectory. Moreover, to get the latest changes from a pull request to a specific repository, you must checkout this repository **first** before checking out any other repositories that you may need.

Now, say you want the latest changes from a "pull request" to a specific repository (repository A) but then need to checkout another repository (repository B) which must be the parent directory of repository A in your workflow. To do so, you can first checkout repository A, then checkout repository B, and "move" (e.g., using the mv command in Linux) repository A into a subfolder of repository B. While this may seem like something that should never be needed, it can happen in practice! In the work done by our team, we had to execute these exact steps in order to get the latest changes added to one repository, but then use this repository as a subdirectory in our build system. To see this example in practice, you can view the "Copy Linux dir into ukl dir" step in this reusable workflow we created for the unikernelLinux CI (note: this workflow is called after first cloning the linux repository, which contains the changes we want to test in our action).

GitHub Secrets

Many workflows need access to sensitive information (API keys, passwords, Access Tokens, etc.) which you may not want to put into your workflow file. To solve this problem, GitHub workflows have access to repository secrets (see how to set up a repository secret here).

After a secret is set up in a repository, the secret can be accessed within a workflow (as an environment variable) using the ${{ secrets.<secret_name> }} syntax.

In our case, in order to clone private repositories within our workflow, we had to first set up an access token in the repository where we set up the workflow (see this documentation for creating access tokens) and then store that token as a repository secret to be used by our workflows. Alternatively, one can take advantage of the GITHUB_TOKEN secret (which is automatically enabled in your workflow) to make authenticated GitHub API requests (using the permissions enabled in the repository in which the workflow is run).