Creating Custom GitHub Actions

CS-615 - Professor Schaumann

Tyler Kaminski

14 April 2022

With the advent of containers and Agile development, a new role for tech professionals has formed as the union of software engineer and systems administrator: the DevOps Engineer. Like the systems administrators who have come before, DevOps Engineers must be skilled automation specialists, creating scripts and automation frameworks to build, test, and debug production software. As Continuous Integration and Continuous Delivery (CI/CD) have become essential to the software development process and the DevOps workflow, CI/CD frameworks have emerged such as GitHub Actions, Travis CI, and Jenkins. In this meetup, Sam Wronski, a cloud advocate at Microsoft, described how to get set up GitHub Actions and how to extend and create new ones to suit whatever purpose a DevOps engineer might have when working with a source code repository. Given my interest in automation, especially in the context of systems administration, I thought this meetup would be helpful for me to learn how automation is currently used in industry by DevOps Engineers to improve the efficiency of the teams they work within.

Actions are written in `yaml` and every action configuration file requires at least the `name` of what the action does, the `description` of the action's results, and a `runs` list to signify what commands should be run within the action. By default, actions will also run after every commit and merged pull request. GitHub Actions also allow for the ability to be run under cron jobs, with cron being a job scheduler that allows actions to be run at certain times of day or at certain intervals, such as once a day at 3 in the morning. GitHub allows actions to be run inside a docker container if specialized testing and building frameworks are used that don't use normal shell-based interaction, such as if a testing framework is written for direct access from a language like Go or TypeScript. If an action does not require a container to execute its building and testing steps, the action's steps can be written as basic one-line shell commands, executed sequentially. A feature also present in GitHub Actions are composite actions, which allows for the ability to execute multiple shell commands to be strung together into one action and create a dependency chain of commands that must all succeed for the action to pass. Given the growth of Javascript as an essential computing language, especially in the context of web development, Microsoft also included support for writing GitHub Actions

directly in Javascript, using NodeJS directly without requiring a shell wrapper, which can simplify workflows for DevOps engineers on web development projects. If an engineer desires to take the containerized approach to setting up CI/CD with GitHub Actions, GitHub's servers will generate docker images entirely in the cloud given solely a `Dockerfile`, populated with the functionality the developer writes. For projects that require inputs that should be kept outside of the repository, such as secrets, API keys, or database credentials, the `inputs` keyword exists for specifying such requirements. These inputs can then be specified within an Action using the `with` keyword.

GitHub Actions also allow for the use of external actions created by other users of GitHub or existing in other repos or directories, which can be included with the `uses` configuration keyword. Since all GitHub Actions are plaintext and open source, any engineer can look into the actions used by a project and if a project depends on other actions, those can be examined as well, with no binaries involved in the action workflow unless a project specifies it in a custom action. This also does open up a security vulnerability, as each GitHub Action dependency becomes one source of possible exploits that may do malicious activities such as mining cryptocurrencies or extracting secrets instead of performing the intended functionality. Some shortcomings also exist when using GitHub Actions in certain niche ways, such as how the NodeJS extension will require a considerable amount of Node dependencies which must exist within the project directory and how Docker containers cannot execute other containers within them.

Below is an example GitHub Action for executing a basic shell composite action, which exist within the git repository of the project we want to automate.

In our `.github/workflows/*.yaml` file:

```yaml
name: "CS615 Action"
on:
  # Run on all repo changes
  push:

jobs:
  # Specify a name for the specific job you want to run and the environment to
  # run it in.
  run-cs615-action:
    runs-on: ubuntu-latest
    name: Run CS615 Action
    steps:
    # First, check out the current repo commit.
    - name: Check out repository
      uses: actions/checkout@v2
```

```
    # Next, run our custom action with an input parameter.
    - name: Composite Shell Action
      uses: ./sh
      # For an actual secure setup, this secret should be stored and managed
      # using GitHub's secrets API.
      with:
        secret: "SuperSecretKey"
```

And then in our `sh/action.yaml` file:

```
name: "CS615 Action"
description: "Echo our secret"
# Import our desired inputs
inputs:
  secret:
    description: "Secret key"
    required: true
runs:
  # Define our composite action, which will be run using bash.
  using: "composite"
  steps:
    - run: echo "My key is ${{ inputs.secret }}"
      shell: bash
```

Upon each new push to the repo, this action will print out the given secret to GitHub's Action log. Such an example can easily be extended to run programs which require API keys as parameters which have a CLI interface.

If you ever want to use GitHub Actions in your own projects, I suggest checking out https://docs.github.com/en/github-ae@latest/actions/creating-actions/metadata-syntax-for-github-actions, which contains tons of useful documentation for what all the fields in an action do and what kinds of uses exist for actions.

As Agile development techniques continue to see more widespread use in industry, I believe systems administrators must be comfortable with managing CI/CD frameworks and extending their functionality to give development teams concrete metrics for test conformance and build success. Especially as industry moves its traditional Systems Administrator functions over to roles such as DevOps and Site Reliability Engineers, systems administrators will need to adapt to new responsibilities such as supervising source control and automating actions that interact with source code. It is for these reasons that every person interested in systems administration should familiarize themselves

with setting up and managing CI/CD pipelines, whether it be using a service like GitHub Actions, Gitlab CI/CD, or Travis CI.

While I have some experience setting up CI/CD pipelines in frameworks such as GitHub Actions and Travis, I felt this meetup gave me more insight into how to tailor existing actions to specifically work the way I have envisioned and how to do so in a more optimized and simple fashion than how I have set them up in the past. The meetup also introduced me to some useful resources for learning how to set up pipelines from scratch, which will help me a lot the next time I need to set up a pipeline and specifically create it within the requirements of the projects I'm working on.