Migrating from CircleCI to GitHub Actions

In this article

GitHub Actions and CircleCI share several similarities in configuration, which makes migration to GitHub Actions relatively straightforward.

Introduction

CircleCI and GitHub Actions both allow you to create workflows that automatically build, test, publish, release, and deploy code. CircleCI and GitHub Actions share some similarities in workflow configuration:

- Workflow configuration files are written in YAML and stored in the repository.
- Workflows include one or more jobs.
- Jobs include one or more steps or individual commands.
- Steps or tasks can be reused and shared with the community.

For more information, see "Understanding GitHub Actions."

Key differences

When migrating from CircleCI, consider the following differences:

- CircleCI's automatic test parallelism automatically groups tests according to user-specified rules or historical timing information. This functionality is not built into GitHub Actions.
- Actions that execute in Docker containers are sensitive to permissions problems since containers have a different mapping of users. You can avoid many of these problems by not using the `USER` instruction in your *Dockerfile*. For more information about the Docker filesystem on GitHub-hosted runners, see "Using GitHub-hosted runners."

Migrating workflows and jobs

CircleCI defines `workflows` in the *config.yml* file, which allows you to configure more than one workflow. GitHub requires one workflow file per workflow, and as a consequence, does not require you to declare `workflows`. You'll need to create a new workflow file for each workflow configured in *config.yml*.

Both CircleCI and GitHub Actions configure `jobs` in the configuration file using similar syntax. If you configure any dependencies between jobs using `requires` in your CircleCI workflow, you can use the equivalent GitHub Actions `needs` syntax. For more information, see "Workflow syntax for GitHub Actions."

Migrating orbs to actions

Both CircleCI and GitHub Actions provide a mechanism to reuse and share tasks in a workflow. CircleCI uses a concept called orbs, written in YAML, to provide tasks that people can reuse in a workflow. GitHub Actions has powerful and flexible reusable components called actions, which you build with either JavaScript files or Docker images. You can create actions by writing custom code that interacts with your repository in any way you'd like, including integrating with GitHub's APIs and any publicly available third-party API. For example, an action can publish npm modules, send SMS alerts when urgent issues are created, or deploy production-ready code. For more information, see "Creating actions."

CircleCI can reuse pieces of workflows with YAML anchors and aliases. GitHub Actions supports the most common need for reusability using matrices. For more information about matrices, see "Using a matrix for your jobs."

Using Docker images

Both CircleCI and GitHub Actions support running steps inside of a Docker image.

CircleCI provides a set of pre-built images with common dependencies. These images have the `USER` set to `circleci`, which causes permissions to conflict with GitHub Actions.

We recommend that you move away from CircleCI's pre-built images when you migrate to GitHub Actions. In many cases, you can use actions to install the additional dependencies you need.

For more information about the Docker filesystem, see "Using GitHub-hosted runners."

For more information about the tools and packages available on GitHub-hosted runner images, see "Using GitHub-hosted runners".

Using variables and secrets

CircleCI and GitHub Actions support setting variables in the configuration file and creating secrets using the CircleCI or GitHub UI.

For more information, see "Variables" and "Using secrets in GitHub Actions."

Caching

CircleCI and GitHub Actions provide a method to manually cache files in the configuration file.

Below is an example of the syntax for each system.

CircleCI syntax for caching

```
- restore_cache:
    keys:
      - v1-npm-deps-{{ checksum "package-lock.json" }}
      - v1-npm-deps-
```

GitHub Actions syntax for caching

```
- name: Cache node modules
  uses: actions/cache@v3
  with:
    path: ~/.npm
    key: v1-npm-deps-${{ hashFiles('**/package-lock.json') }}
    restore-keys: v1-npm-deps-
```

GitHub Actions does not have an equivalent of CircleCI's Docker Layer Caching (or DLC).

Persisting data between jobs

Both CircleCI and GitHub Actions provide mechanisms to persist data between jobs.

Below is an example in CircleCI and GitHub Actions configuration syntax.

CircleCI syntax for persisting data between jobs

```
- persist_to_workspace:
    root: workspace
    paths:
      - math-homework.txt

...

- attach_workspace:
    at: /tmp/workspace
```

GitHub Actions syntax for persisting data between jobs

```
- name: Upload math result for job 1
  uses: actions/upload-artifact@v3
  with:
    name: homework
    path: math-homework.txt

...

- name: Download math result for job 1
  uses: actions/download-artifact@v3
  with:
    name: homework
```

For more information, see "Storing workflow data as artifacts."

Using databases and service containers

Both systems enable you to include additional containers for databases, caching, or other dependencies.

In CircleCI, the first image listed in the *config.yaml* is the primary image used to run commands. GitHub Actions uses explicit sections: use `container` for the primary container, and list additional containers in `services`.

Below is an example in CircleCI and GitHub Actions configuration syntax.

CircleCI syntax for using databases and service containers

```
---
version: 2.1

jobs:

  ruby-26:
    docker:
      - image: circleci/ruby:2.6.3-node-browsers-legacy
        environment:
          PGHOST: localhost
          PGUSER: administrate
          RAILS_ENV: test
      - image: postgres:10.1-alpine
        environment:
          POSTGRES_USER: administrate
          POSTGRES_DB: ruby26
          POSTGRES_PASSWORD: ""

    working_directory: ~/administrate

    steps:
      - checkout

      # Bundle install dependencies
      - run: bundle install --path vendor/bundle

      # Wait for DB
      - run: dockerize -wait tcp://localhost:5432 -timeout 1m

      # Setup the environment
      - run: cp .sample.env .env

      # Setup the database
      - run: bundle exec rake db:setup

      # Run the tests
      - run: bundle exec rake

workflows:
  version: 2
  build:
    jobs:
      - ruby-26
...

- attach_workspace:
    at: /tmp/workspace
```

GitHub Actions syntax for using databases and service containers

```
name: Containers

on: [push]
```

```yaml
jobs:
  build:

    runs-on: ubuntu-latest
    container: circleci/ruby:2.6.3-node-browsers-legacy

    env:
      PGHOST: postgres
      PGUSER: administrate
      RAILS_ENV: test

    services:
      postgres:
        image: postgres:10.1-alpine
        env:
          POSTGRES_USER: administrate
          POSTGRES_DB: ruby25
          POSTGRES_PASSWORD: ""
        ports:
          - 5432:5432
        # Add a health check
        options: --health-cmd pg_isready --health-interval 10s --health-timeout
5s --health-retries 5

    steps:
      # This Docker file changes sets USER to circleci instead of using the
default user, so we need to update file permissions for this image to work on GH
Actions.
      # See https://docs.github.com/actions/using-github-hosted-runners/about-
github-hosted-runners#docker-container-filesystem

      - name: Setup file system permissions
        run: sudo chmod -R 777 $GITHUB_WORKSPACE /github /__w/_temp
      - uses: actions/checkout@v4
      - name: Install dependencies
        run: bundle install --path vendor/bundle
      - name: Setup environment configuration
        run: cp .sample.env .env
      - name: Setup database
        run: bundle exec rake db:setup
      - name: Run tests
        run: bundle exec rake
```

For more information, see "About service containers."

Complete Example

Below is a real-world example. The left shows the actual CircleCI *config.yml* for the thoughtbot/administrator repository. The right shows the GitHub Actions equivalent.

Complete example for CircleCI

```yaml
---
version: 2.1

commands:
  shared_steps:
    steps:
      - checkout

      # Restore Cached Dependencies
      - restore_cache:
          name: Restore bundle cache
          key: administrate-{{ checksum "Gemfile.lock" }}
```

```
      # Bundle install dependencies
      - run: bundle install --path vendor/bundle

      # Cache Dependencies
      - save_cache:
          name: Store bundle cache
          key: administrate-{{ checksum "Gemfile.lock" }}
          paths:
            - vendor/bundle

      # Wait for DB
      - run: dockerize -wait tcp://localhost:5432 -timeout 1m

      # Setup the environment
      - run: cp .sample.env .env

      # Setup the database
      - run: bundle exec rake db:setup

      # Run the tests
      - run: bundle exec rake

default_job: &default_job
  working_directory: ~/administrate
  steps:
    - shared_steps
    # Run the tests against multiple versions of Rails
    - run: bundle exec appraisal install
    - run: bundle exec appraisal rake

jobs:
  ruby-25:
    <<: *default_job
    docker:
      - image: circleci/ruby:2.5.0-node-browsers
        environment:
          PGHOST: localhost
          PGUSER: administrate
          RAILS_ENV: test
      - image: postgres:10.1-alpine
        environment:
          POSTGRES_USER: administrate
          POSTGRES_DB: ruby25
          POSTGRES_PASSWORD: ""

  ruby-26:
    <<: *default_job
    docker:
      - image: circleci/ruby:2.6.3-node-browsers-legacy
        environment:
          PGHOST: localhost
          PGUSER: administrate
          RAILS_ENV: test
      - image: postgres:10.1-alpine
        environment:
          POSTGRES_USER: administrate
          POSTGRES_DB: ruby26
          POSTGRES_PASSWORD: ""

workflows:
  version: 2
  multiple-rubies:
    jobs:
      - ruby-26
      - ruby-25
```

Complete example for GitHub Actions

```yaml
# This workflow uses actions that are not certified by GitHub.
# They are provided by a third-party and are governed by
# separate terms of service, privacy policy, and support
# documentation.

# GitHub recommends pinning actions to a commit SHA.
# To get a newer version, you will need to update the SHA.
# You can also reference a tag or branch, but the action may change without
warning.

name: Containers

on: [push]

jobs:
  build:

    strategy:
      matrix:
        ruby: ['2.5', '2.6.3']

    runs-on: ubuntu-latest

    env:
      PGHOST: localhost
      PGUSER: administrate
      RAILS_ENV: test

    services:
      postgres:
        image: postgres:10.1-alpine
        env:
          POSTGRES_USER: administrate
          POSTGRES_DB: ruby25
          POSTGRES_PASSWORD: ""
        ports:
          - 5432:5432
        # Add a health check
        options: --health-cmd pg_isready --health-interval 10s --health-timeout
5s --health-retries 5

    steps:
      - uses: actions/checkout@v4
      - name: Setup Ruby
        uses: eregon/use-ruby-action@ec02537da5712d66d4d50a0f33b7eb52773b5ed1
        with:
          ruby-version: ${{ matrix.ruby }}
      - name: Cache dependencies
        uses: actions/cache@v3
        with:
          path: vendor/bundle
          key: administrate-${{ matrix.image }}-${{ hashFiles('Gemfile.lock') }}
      - name: Install postgres headers
        run: |
          sudo apt-get update
          sudo apt-get install libpq-dev
      - name: Install dependencies
        run: bundle install --path vendor/bundle
      - name: Setup environment configuration
        run: cp .sample.env .env
      - name: Setup database
        run: bundle exec rake db:setup
      - name: Run tests
        run: bundle exec rake
      - name: Install appraisal
        run: bundle exec appraisal install
      - name: Run appraisal
        run: bundle exec appraisal rake
```