# GitHub Actions

https://www.linkedin.com/learning/learning-github-actions-2/solution-develop-a-ci-cd-pipeline-for-a-python-script?autoSkip=true&dApp=200778144&resume=false&u=106815226

## Attributes

* name (workflow) -> name of the workflow. Should assist in describing what it does;
* on -> describes the event that will trigger the workflow -> <https://docs.github.com/en/actions/using-workflows/events-that-trigger-workflows#push>
* jobs -> defines a list of tasks for the workflow;
* runs-on -> the type of machine we’d like to use to run the job. Describes the operating system for the virtual environment where the job will run.
* Steps -> lists specific actions and commands. Each step has access to the file system in the virtual env but runs in its own distinct and separate process;
* uses -> tells the job a specific action to use. Actions are bundles of code used to perform a specific task or operation. Essentially, actions are docker images. Its main use is to tell the workflow how to find the action needed by the step;
* run -> used instead of uses. Executes a command or series of commands in a shell on the virtual env.
* name (steps) -> can be paired with uses and runs attributes;
* needs -> identifies one or more jobs that must complete successfully before a job will run;
* branches, tags, branches-ignore, tags-ignore -> conditional filtration to witch branches/tags the workflow should be applied or ignored to;
* with -> a map of the input params defined by the action. They are set as env vars and prefixed with INPUT\_ and converted to uppercase;
* env -> used to define a block of environment variables on Workflow, Jobs or Steps level;

## Workflow

* defines the GitHub event that triggers execution of any actions contained in the workflow;
* defines which actions will be run based on the event;

### Events

* a single trigger can be done as simple as on: push
* multiple triggers need to be enclosed like on: [push, pull\_request]

#### Conditions

* https://docs.github.com/en/actions/using-workflows/workflow-syntax-for-github-actions#onpushbranchestagsbranches-ignoretags-ignore
* On trigger we can limit activity to a specific branch/tag or ignore it;
* **Special characters are also allowed for a RegEx to match multiple similar**
* Achieved via the **branches** keyword in a given event
* Filter pattern cheat sheet -> https://docs.github.com/en/actions/using-workflows/workflow-syntax-for-github-actions#filter-pattern-cheat-sheet

Usage:

***on:***

***push:***

***branches:***

***- develop***

***- main***

***pull\_request:***

***branches:***

***- main***

### Jobs

* Workflows must have at least one job;
* Each job must have a unique identifier.
* Job identifiers must start with a letter or underscore;
* Jobs run in parallel by default;

#### Steps

* Tasks within a job;
* Each step is run as a process on our compute resources;
* This is where the commands and actions are executed;

##### Actions

* Specified with the **uses** keyword – execute an action in a new process on the operating system;
* Can contain multiple commands;

###### Syntax

|  |  |
| --- | --- |
| **Action location** | **Syntax** |
| Public repository | uses: {owner}/{repo}@{ref}  uses: octocat/super-cool-action@v1 |
| Same repository as the workflow | uses ./path/to/the/action  uses: ./.github/actions/my-local-action |
| Docker image registry | uses: docker://{image}:{tag}  uses: docker://hello-world:latest |

###### Common public actions

* checkout – checks out all the repositories code into the compute environment’s file system making files available to steps that run later in the job;

Usage***: uses: actions/checkout@v2***

##### Commands

* Specified with the **run** keyword – execute a single command using the default shell of the jobs operating system
  + Bash – Ubuntu, MacOS;
  + PowerShell – Windows (https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/get-date?view=powershell-7.3);

###### Syntax

|  |  |
| --- | --- |
| **Run** | **Syntax** |
| Single-line command | run: {command} {parameters} {arguments}  run: mv ./output ./archive |
| Multiline command | run: |  Command 1  Command 2  run: |  g++ -c -Wall -g Main.cpp  g++ -g -I Main.exe Main.o |

#### Dependencies

* For when a job produces a output that another job needs it as a input OR when a job should run after another has finished;
* Achieved via the **needs** attribute;

Usage: ***needs: [job1, job2]***

### Limitations

* One repository can contain multiple workflows, but only 20 can be running at the same time;
* Actions cant trigger other workflows;

### Workflow example to print env vars in ubuntu and windows runners

name: 02\_First\_Workflow

on:

[push]

jobs:

job1:

name: First job

runs-on: ubuntu-latest

steps:

- name: Step one

uses: actions/checkout@v2

- name: Step two

run: env | sort

job2:

name: Second job

runs-on: windows-latest

steps:

- name: Step one

uses: actions/checkout@v2

- name: Step two

run: "Get-ChildItem Env: | Sort-Object Name"

## Using Actions

### From Marketplace

* Simple enough, we can just go to the market place and search for the needed item;

### From Repository

* Can be referenced from
  + The workflow repository;
    - Need to specify the path to the action file;
    - - uses: “./.github/action1”
  + A public repository;
    - Need to specify the repo’s owner user ID, the repo and a reference;
    - Reference can be a branch, tag or SHA;
    - If the action is not located in the root of the repository, then the path needs to be specified;
    - - uses: “user/repo@ref”
    - - uses: “octocat/my-cool-action@develop”
  + Docker image published to a repository;
    - Specifying the docker path to the image and tag
    - - uses: “docker://image:tag” -> docker hub
    - - uses: “docker://host/image:tag” -> public image repo other
    - - uses: “docker://python:3.9”

### Passing Arguments

* Steps use the **with** keyword to pass attributes;
* Creates a new block for mapping args to inputes
  + uses: {github account}/{action name}
  + with: key: value

Example:

steps:

* name: Checkout the code

uses: actions/checkout@v2

with:

repository: apache/tomcat

ref: master

path: ./tomcat

where

* repository is the apache/tomcat tells the checkout action to check out all the code from the apache tomcat repo on github. If not specified then the default is to check out the repository where the action is being run.
* Ref is the references to a commit for the checkout (branch, tag or SHA hash);
* Path is where to put the code relative to the working directory;

### Using environment variables

* Providing additional information to actions at runtime;
* Stored in memory in a virtual env, running our workflow;
* Commands and actions running as steps can access env vars to use the information that they hold;
* Defined via the **env** keyword on Workflow, Jobs or Steps level;
* Can be accessed via
  + Shell variable syntax – passed to the shell to be interpreted (the workflow passes the shell value and the shell does the interpretation)
    - Bash (Linux/macOS)
      * $VARIABLE\_NAME
    - PowerShell (Windows)
      * $Env:VARIABLE\_NAME
  + YAML syntax – variable is read from the workflow – meaning the variable is interpreted before it gets passed to any steps that use it. Because of that, they can be used to configure other parts of the workflow.
    - ${{ env.VARIABLE\_NAME }}
* Default Env Vars -> https://docs.github.com/en/actions/learn-github-actions/variables#defining-environment-variables-for-a-single-workflow

### Using secrets

* Special type of env var for securing sensitive information like API keys etc..
* Stored as encrypted values in GitHub repo settings;
* Can’t be viewed or edited. Must be deleted and recreated to change;
* Workflows are limited to 100 secrets;
* Defined in the repository settings > Secrets;
* Can be accessed via the secrets workflow context
  + ${{ secrets.SECRET\_NAME }}

### Using artifacts

* Data preserved from a workflow;
* Can be a file or collection of files;
* Can be used to pass data between workflow jobs;
* Can only be uploaded via
  + Actions/upload-artifact
* Can only be downloaded by the uploading workflow via
  + Actions/download-artifact

### Manage pull requests

* PR is possible to be checked. Tested and auto merge

## Setting CI/CD pipeline

* The main goal is to find and resolve problems early in the development cycle.
* Continuous Integration – devs work on their code in local env and commit their changes to shared repo. Their code can be combined with other;
* Continuous Delivery and Deployment – the integraded code from CI is compiled into artifacts and stored. To further improve the quality of the product, the artifacts can be used for additional testing before they are made available for deployment. And in a full continuous deployment cycle, artifacts are deployed to live envs.
* Example steps in the flow;
  + Push – triggered on each push
  + Lint check – lint the code and run predefined tests
    - Enforce coding standards (naming conv, formatting)
    - Improve code quality (unused imports, typos)
    - Catch errors in the design cycle;
    - Unit tests run
  + Build – build a docker image using a docker file in the repo
    - Compile code into binary package
    - Machine-readable format
    - Produce a artifact.
    - Exists beyond the life of the build step
    - Stored and tracked in registries
    - Authentication is required to upload artifacts
  + Store – we’ll upload the image to a package registry to make it available for deployment.
  + Test – we can test the image in a virtual env
  + Deploy – to a live env

## Custom Actions

* The objective
  + What will the action do? What problem will it solve?
  + Can the action be parameterized?
* The repo
  + Keeping the action in its own repo will help in maintaining and versioning the code;
* Docker file
  + All actions require a docker file to define the container env that runs the action;
* Script
  + Defines commands that runt he action
  + Interacts with env vars

### Docker review

* Docker files include all the information to build a docker image (refs to other images, specific commands to run during the build process);
* Format
  + # -> comment INSTRUCTION arguments
  + Commands
    - FROM -> sets the base image that serves as the starting point for our image. Each instruction after that is used to add layers to the base image. Its recommended to use official images from hub.docker.com. They include operating systems and programming languages.
    - RUN -> will execute a command in a new layer on top of the current image; Usefull for installing software into a base OS;
    - COPY -> copy files into the container image – takes files from the local env and copies them into the container image. That way, the file is available inside the container even if its used outside the repo;
    - ENTRYPOINT -> Used to make our container image into an executable that gets run when our custom action is used. Arguments are passed in as an array

### Runtime Environment Resources

* When a workflow is triggered, a virtual runtime env is created to provide the workflow with a variety of resources. This includes
  + Compute resources – includes
    - One virtual CPU
    - Memory up to 3.75 GB
    - Remote network connectivity – usefull for connecting to external resources like aPIs
  + Local file system - includes
    - Allows the action to read and write files locally – up to 100 GB
    - Paths to /github/home and /github/workspace
    - Check GITHUB\_ env vars
  + Env vars
  + Event payloads – check github docs for the events details

# GitHub Actions – Advanced

## Advanced workflows

### Service containers

* They are Docker containers that run as part of a job, using the **Services** context, we can add blocks for named services along with the container image the service will use and any other configuration it needs – env vars (**env context)**, network port connections under the **ports context**, and volumes under th**e volume context**. Any other settings needed can be passed in using the **options context**.
* Notes when using service containers
  + The job running the service must be Linux based (ubuntu-latest) if using self-hosted runners;
  + Self-hosted runners must use Linux and Docker;
  + Services will be run directly on the runner – must configure a map to a port on the runner;
  + Use localhost and a mapped port to connect to the service;
  + Example:

### Scheduled triggers

* Added via the **schedule** keyword in the event block;
* Using the cron format;
* Cron job must be quoted because the \* sign is special in yaml files;
* All schedules use UTC

### Composite actions

* Helps with repeatable steps;
* Usually the code is defined in a separate repo following a specific syntax (check git course uploads) and then the composite action is referenced wherever used;

### Manual approvals and environments

* Environment protection rules. For example
  + When we have automated a release deployment to a environment but we still want a manual approval of the action;
* GitHub uses environments to describe target for deployment. To protect environments, rules can be defined to identify which branches can deploy to a given environment (deployment branches).
* Environments can also contain secrets that are used at the repository.

### Caching between workflow runs

* Helps in downloading and storing dependencies locally so that we don’t download them every time.
* In GitHub actions, workflows often start in a new virtual environment, so we lose the advantage of caching. If not handled with care, the build times can be slow without caching.
* There are 2 approaches for caching:
* The following prog lang have setup actions that can provide caching.

|  |  |  |
| --- | --- | --- |
| **Package manager** | **Setup.\* action for caching** | **Cache activation** |
| Npm, yarn, pnpm | Setup-node | uses: actions/setup-node@v2  with:  node-version: ‘14’  cache: ‘npm‘ |
| Pip, pipenv | Setup-python | uses: actions/setup-python@v2  with:  python-version: ‘3.9’  cache: ‘pip‘ |
| Gradle, maven | Setup-java | uses: actions/setup-java@v2  with:  java-version: ‘11’  cache: ‘maven‘ |
| Ruby, gems | Setup-ruby | uses: actions/setup-ruby@v1  with:  bundler-cache: true |

* If not the above, then we can do it by:
* If we want to cache objects not managed by the setup action, we can use the **actions/cache@v2** action.
* Good practice for generating a key is to use a value from hashing a file. The build in **hashFiles(‘FILE\_NAME’)** is a good option. If the file changes, then a new one is created. Can use the following as well:
  + OS -> ${{ runner.os }}
  + Workflow var -> ${{ env.cache-name }} – we should config it
* Limits
  + Caches are limited to the active branch in a workflow
  + There is no limit on the numbers of caches
  + Total space is 10gb else oldest gets removed
  + Not used for 7 days will remove cache
* Name: My cache

Uses: actions/cache@v2

With:

Key: ‘the cache-key -> ***used to detriment if a cache should be restored or created***

Path: /file/system/location’ -> ***where the cache will be***

## Matrix strategy

* When we want to build and test with different parameters (different versions of software – 3 different versions of node js). Using a matrix strategy, we provide all the necessary parameters as an array to 1 job, and github would create all the possible combination for us.
* Overview
  + We can use 1 job to create a variety of different configurations.
  + Matrix is defined by the **strategy** keyword, followed by the **matrix** keyword:

.

.

jobs:

build:

strategy:

fail-fast: false

matrix:

version: [14, 16, 17]

platform: [ubuntu-latest, macos-latest, windows-latest]

experimental: [false]

include:

* platform: ubuntu-latest

version: 18

experimental: true

exclude:

* platform: windows-latest

version: 16

continue-on-error: ${{ matrix.experimental }}

runs-on: ${{ matrix.platform }}

steps:

* uses: actions/setup-node@2

with:

node-version: ${{ matrix.version }}

.

.

* + We’ll be able to reference these values in the workflow by using the **matrix** context, followed by the key;
  + Can be referenced from anywhere in the workflow but are especially usefull in the runs-on and the uses step;
  + Matrix will use each of the values we specify to create a separate instance of our job. – in this example – 3 different jobs will be created using the same configuration but with different values for the version input;
  + Options
    - include/exclude
      * key-value combinations
      * Can both be used in the same matrix
      * Includes are processed after excludes
    - Fail-fast
      * Allows github to cancel all jobs that are in progress on the first failed job;
      * Default value is **true**;
      * Can be used with the continue-on-error setting (in the example, fail-fast will not apply for our experimental jobs (include)

## Publishing packages

### Package registry overview

* GitHub package service – is a hosting service that allows us to host packages privately or publicly and use them as dependencies in our projects.
* Permissions and visibility are inherit from the repository;
* Authentication is required to create and access software packages
  + Using GITHUB\_TOKEN in GitHubActions workflows to authenticate the package management tools;
  + For access from local system or other app, we need to setup a personal access token with correct access permissions;
* Container images can be configured to allow anonymous public access;
* Container image permissions can be customized to accounts and organizations;

### Publishing container images

* Provides a native container registry. This allows us to tightly integrate GitHub Actions to build and publish images using code repo events like commits and releases;
* The process:
  + Event and checkout code
  + Log into the registry to host the image

A screenshot of a computer

Description automatically generated

* + Gather image metadata to tag the imageA screen shot of a computer code

    Description automatically generated
  + Build and publishA computer screen shot of a computer code

    Description automatically generated
* Using images
  + Public images
    - Using ‘docker pull’ with the registry, username, image name and tag;
  + Private images
    - Authenticate with a personal access token before pulling the image;
  + In GitHub Action
    - In a ‘run’ step – **run: docker run {REGISTRY\_NAME}/{GITHUB\_ACCOUNT/HOSTING}/{IMAGE\_NAME}:{BRANCH/TAG}**
    - In a ‘uses’ step – **uses: {GITHUB\_ACCOUNT/HOSTING}/{IMAGE\_NAME}@}:{BRANCH/TAG}**

### Publishing software packages (GitHub Packages REGISTRY)

* Configuration for the package registry
  + Required data
    - References the repository
    - Package metadata
      * Version number
      * Authors name
      * Description
      * License

Depending on the language:

A screenshot of a computer

Description automatically generated

* Authenticate with the registry
* Build the package
* Publish the package to the registryA screenshot of a computer

  Description automatically generated

Using packages

* Use config files to set up GitHub Packages as a registry;
* Software packages follow the permissions and visibility of the repo that hosts their code;
* Public repos need an authentication as a github user

## Self-hosted runners

### Intro

* Allows us to configure our jobs compute environment to the needed specifications. We install the OS, any software that us required and then connect it directly to the GitHub repo;

### GitHub-hosted runners vs self-hosted runners

|  |  |
| --- | --- |
| GitHub-hosted runners | Self-hosted runners |
| 2-3 core CPU | As configured |
| 7 or 14 GB of RAM | As configured |
| 14 GB of SSD disk space | As configured |
| Ubuntu Linux, Windows, macOS (regularly updated and patched) | Red Hat Enterprise Linux, CentOS, Fedora, Windows (diff ver), etc.. (we are responsible to update and patch) |
| Preinstalled runtimes, package managers, SDKs, tools (regularly updated) | Tools and dependencies need to be installed (we are responsible to update) |
| Workflows are run in new compute env each time | May persist after run. Helps with caching, configuration problems may be hard to debug |
|  | Queued jobs are limited to 24 hours |
|  | Workflow runs are limited to 72 hours |
|  | Idle runners are removed after 30 days if not used |
|  | Should be used with private repos ONLY !! |
|  | Malicious code can persist across workflows |
|  | Limit access to trusted collaborators |
|  | Security hardening for GitHub Actions – must read |

### Configure a self-hosted runner

* Can be associated with
  + Repositories
  + Organizations
  + Enterprises

## JavaScript Actions

### Overview

A white background with blue text

Description automatically generated

We would need a

* Repository for the code (easier versioning, maintenance and usage)

### Develop

* The following are required
  + Action metadata file
    - Stored at the root of the repo as yaml file – action.yml
    - It represents the interface which github actions workflow will use to connect to our action
    - It describes
      * Inputs
      * Outputs
      * Runtime
      * Author
      * Desc

A screen shot of a computer

Description automatically generated

* + README file
    - Required for marketplace
    - Detailed desc
    - Inputs, outputs
    - Requirements
    - Example usage
  + Project file updates
    - .gitignore – remove dist (allows compiled files to be added to the repo)
    - Package.json

A screen shot of a computer program

Description automatically generated

### Use actions/core in JS actions

* The GitHub actions toolkit – a set of packages to make creating actions easier
  + @actions/core
    - Functionalities for working with GitHub actions front end and internals of the github actions framework;
    - Used for reading inputs and setting outputs

A screenshot of a computer program

Description automatically generated

* + - Createn annotations that are presented on the GHA web interface

A group of black text

Description automatically generated

* + - Set exit codes

### Use actions/github in JS actions

* @action/github
  + Expose the context passed from the workflow to the action as JSON

A screenshot of a computer

Description automatically generated

* + Provides an authenticated octokit client – allows JavaScript actions to access the github AP

A computer code with black text

Description automatically generated

* Com bining contexts and octokit

A screen shot of a computer code

Description automatically generated