TECH BLOG

Home Discover & Search

Home > Discover & Search > A practical introduction to Github Actions and Workflows

A practical introduction to Github Actions and Workflows

JS Jörg Schultze-Lutter | Last modified: 13.04.2022 | 9 minutes read

Learn how you can use Github Actions and Workflows for automation purposes

Table of Contents

Introduction

Github Actions and Workflows can be used to automate processing steps for a Github repository so that, for example, defined processing steps can be performed automatically when checking in code, creating new releases, etc. You can also create your own Workflows from scratch.

In the following article, we will first look at one of Github's standard workflows and then explore further steps using a workflow I developed for deploying Python packages to the Python Package Index (PyPi) on a new release. The associated code sample of this workflow has been intentionally split into multiple processing steps so that you can learn how to pass parameters between Jobs and reuse existing code.

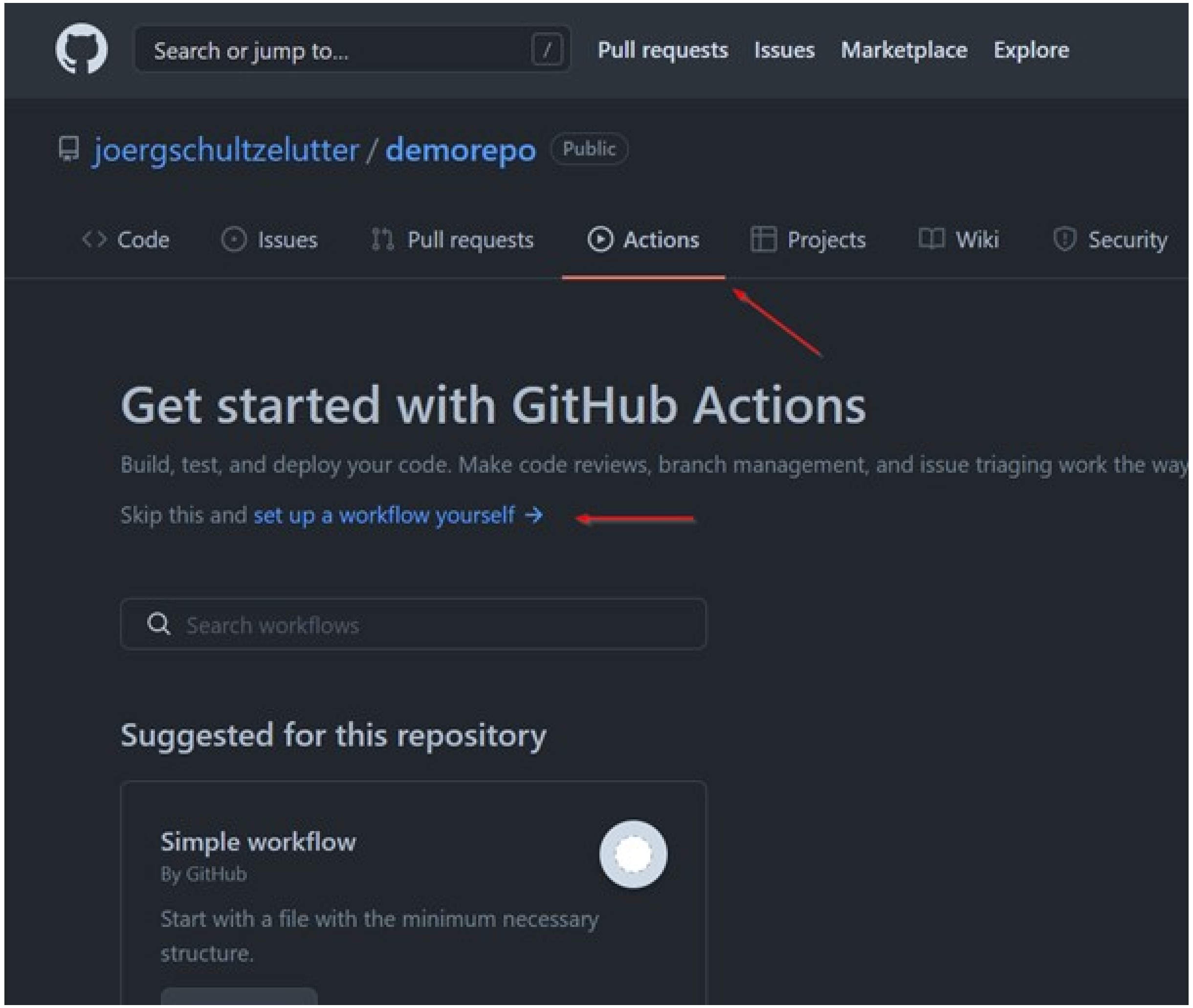
But let's start with the basics first and install our first Github Workflow which will perform a code security check for us whenever we apply changes to our repository.

Setting up our first Github Workflow

To set up a Github Workflow, the repository must first be in 'public' status. Workflows as well as Github Actions are of course available for 'private' repositories, but in this case you have to pay for the execution of the actions.

In the following example I assume that our repository' visibility is 'public'.

The **Github Actions** menu of the repository is accessed via the ***Actions*** tab. Here the developer can either select one of the predefined Workflows or create his own Workflow.



Get started with GitHub Actions

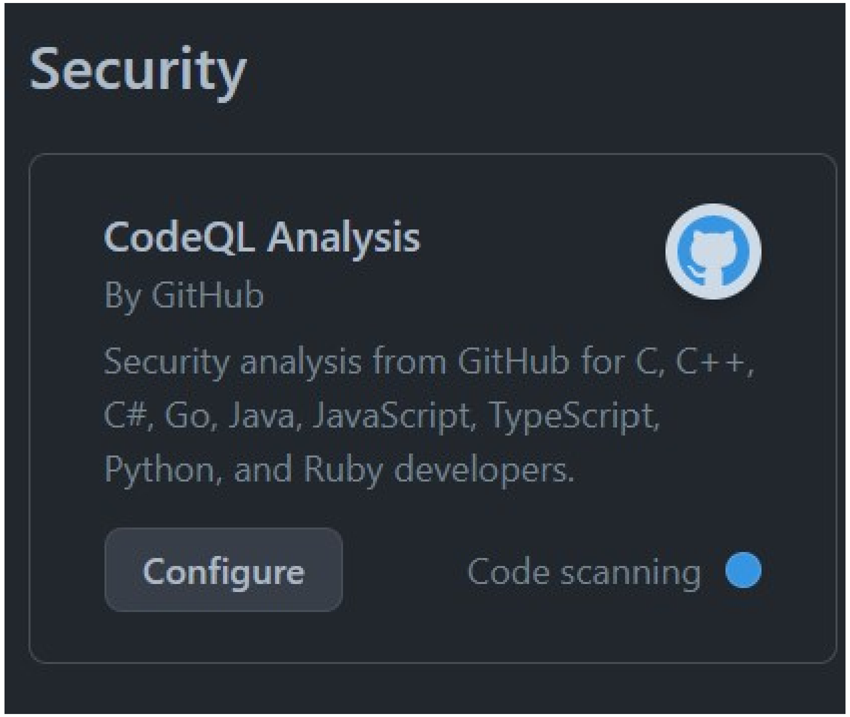
Build, test, and deploy your code. Make code reviews, branch management, and issue triaging work the way

Skip this and set up a workflow yourself →

Suggested for this repository

Simple workflow
By GitHub
Start with a file with the minimum necessary structure.

For our first example, we select the **CodeQL Analysis** Workflow and click the Configure button.



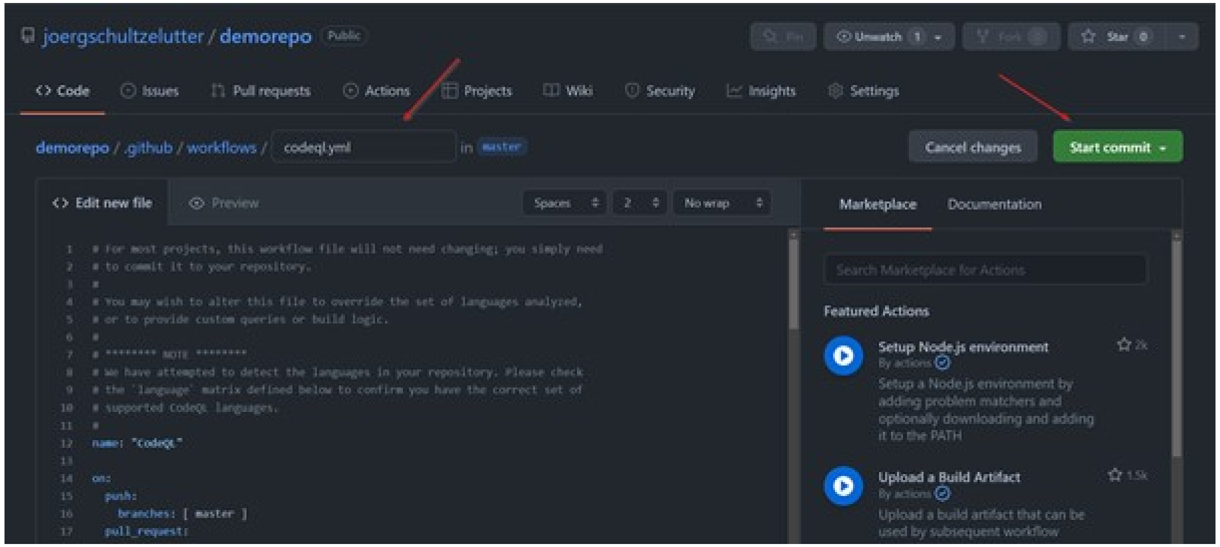
Security

CodeQL Analysis

By GitHub

Security analysis from GitHub for C, C++, C#, Go, Java, JavaScript, TypeScript, Python, and Ruby developers.

In the next window we get access to the code associated with this Workflow, which we may want to amend and/or save to a different file name. For our example, we do not apply any changes and save that Workflow under its default name ***codeql.yml*** using **Start commit** followed by ***Commit new file*** button.



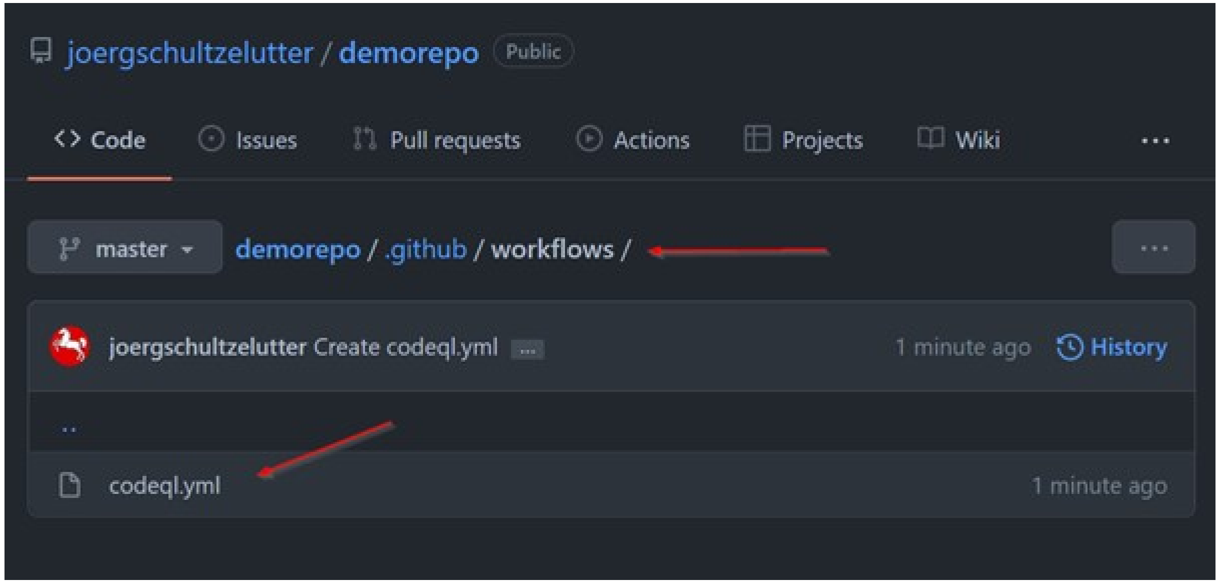
Setup a Node.js environment by adding problem matchers and optionally downloading and adding it to the PATH

Upload a build artifact that can be used by subsequent workflow

```
# For most projects, this workflow file will not need changing; you simply need
# to commit it to your repository.
#
# You may wish to alter this file to override the set of languages analyzed,
# or to provide custom queries or build logic.
#
# ******** NOTE ********
# We have attempted to detect the languages in your repository. Please check
# the `language` matrix defined below to confirm you have the correct set of
# supported CodeQL languages.
#
name: "CodeQL"

on:
  push:
    branches: [ master ]
  pull_request:
```

Our new Workflow is automatically activated after saving with the trigger predefined in the YAML source code (e.g. ***push*** or ***pull_request***; we will discuss these options in the next chapter).

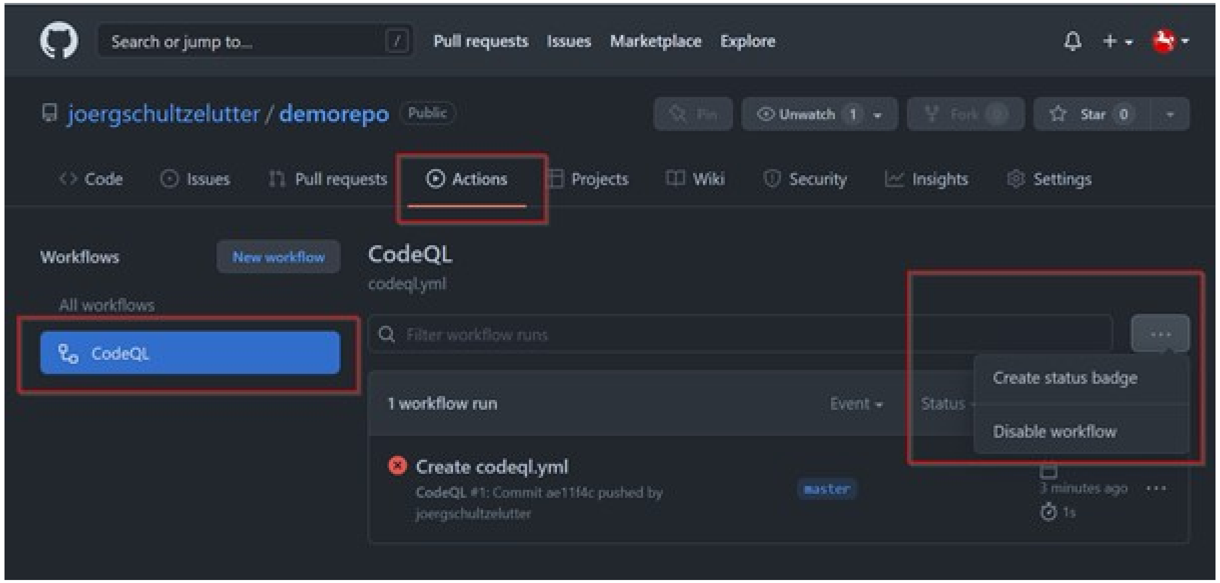

joergschultzelutter / demorepo Public
Code
Issues
Pull requests
Actions
Projects
Wiki
...
master
demorepo / .github / workflows /
...
joergschultzelutter Create codeql.yml ...
1 minute ago
History
..
codeql.yml
1 minute ago

Using the ***Actions*** menu of the repository, we can see the state of the Workflow, results for future Workflow runs, and disable the Workflow if needed. Based on the trigger configuration for this CodeQL Workflow, a security analysis of the code is now automatically performed for each push or pull request of new code into this repository.

We have successfully installed and activated our first Github Workflow.

Workflow Structure

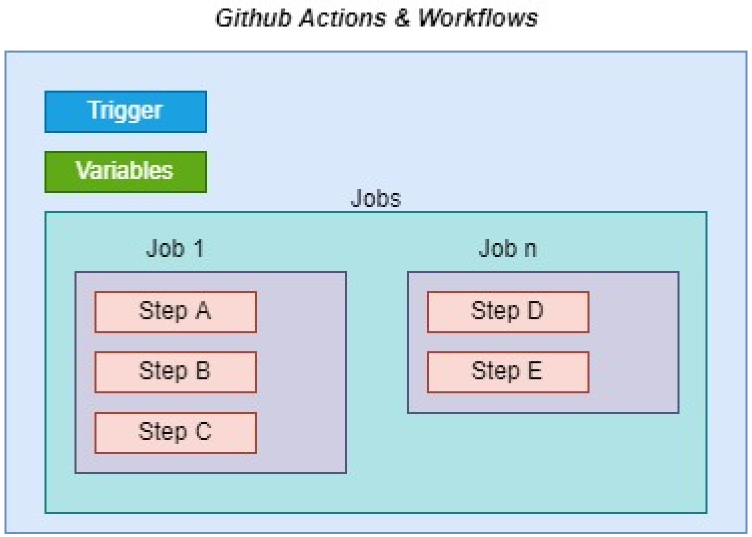
Workflows are YAML files which are always located in the ***.github/workflow*** directory of the respective Github repository. Subsequent changes to a Workflow are therefore possible at any time via changes to the associated file. It should be noted that the change to a Workflow YAML file usually triggers and activates this file itself after saving, since one has made changes to the repository. If you want to prevent this behavior, the Workflow should be deactivated before a change. (***Disable workflow***).



A Workflow consists of :

- a ***Trigger*** information for the Workflow. This Trigger determines when the script is to be activated (e.g. on every code push)
- an optional area, in which global variables can be defined.
- 1...n ***Jobs***, which in turn consist of 1...n ***Steps***. These are the actual tasks that we intend to have executed.

- other sections which are not part of this article



Workflows are executed based on freely definable **Triggers**. Such a Trigger can be, for example, a code push or a new release. Within these main Triggers, further selection criteria can be defined, which, for example, only activate an action for a code push if this push has occurred for the master branch.

Each Workflow normally acts independently of other Workflows. Each Workflow consists of one or more **Jobs**, which act independently of each other by default – even if they are stored in the same Workflow file. However, it is also possible to create dependencies between the respective Jobs, so that, for example, Job B is only executed after Job A has finished. If a Workflow consists of multiple Jobs, an error in one Step will also abort all Steps and Jobs of the subsequent processing tasks.

Workflow Trigger

A Workflow trigger is defined by the keyword ***'on'***, followed by one or more main conditions. Additional sub-conditions can be defined for each main condition. Example:

```
on:
  release:
    types: [published]
    branches: [master]
  push:
```

For the main conditions an 'or' consideration is made, while for the sub-conditions an 'and' linkage is applied. For the previous example, this means that the action is executed on a published master release or on a normal push.

Workflow Environment Variables

Variables are defined in the optional section '***env***' and have a global scope over all possible Jobs of this Workflow. Example:

```
env:
  SOURCE_FILE: ./MyDir/MyFile.py
```

Your Jobs and Steps can reference to these environment variables and use their values. Example:

```
      # Read source file
      - name: Read source file
        id: reader
        uses: juliangruber/read-file-action@v1
        with:
          path: ${{ env.SOURCE_FILE }}
```

Workflow Jobs and Steps

Jobs represent the actual processing steps that we want to automate via our Workflow. A Job has a name and consists of 1...n **Steps**. These steps are processed sequentially, assuming an error-free run. Each Step, as the smallest unit, is in turn assigned a name as well as a unique ID, which can then be used within a Task to reference the results of a preceding processing Step, for example. If the Workflow also supports parameters for Github Actions, these can also be specified for the Step.

If Workflow consists of several Jobs and no dependencies are defined between the Jobs, then all defined Jobs within this YAML Workflow file are executed in parallel. Variables to be passed from one Job to its successor must be passed using the outputs-needs construct. Additionally, an operating system is assigned to each Job by keyword ***runs-on*** (Github-hosted Runners), so that one can start e.g. Workflows on Windows machines (or on own infrastructure) if necessary.

To build a Workflow's Jobs, one can use predefined Actions from the Github Actions Marketplace. These Actions require the developer to use them with a separator '@' and mandatory version info - omitting the version information will result in an error. See the upcoming example for details.

Whenever a Workflow is executed, it uses a checked-out version of your repository. This means that changes made to the repository by your Workflow are **not** persisted unless you instruct your Workflow to do so.

Minimal sample Workflow structure

```
name: My Package Name
on:
  release:
    types: [published]
    branches: [master]
env:
  PYTHON_VERSION: '3.8'
jobs:
  my-first-job:
    runs-on: ubuntu-latest
    steps:
      - name: step 1
```

```
        id: unique_id
        uses: Github Marketplace Action@version
        with:
          parameter: 'some value'
```

Subsequent Steps of the same Job can later reference and access the results of the previous job via the value of the ***id*** field.

Exchanging variables between different Jobs

To exchange values between individual Jobs, the **outputs-needs construct** must be used. Example:

```
jobs:
  #
  # BEGIN of Job 1
  #
  get-python-version-info:
    runs-on: ubuntu-latest

    # Output which is passed to the PyPi publication job
    outputs:
      my-program-version: ${{ steps.regex.outputs.group1 }}
    steps:
      ....
      ....
      # Run regex on content and try to get the version data
      - name: get version via regex
        id: regex
        uses: actions-ecosystem/action-regex-match@v2
        with:
          regex: ${{ env.REGEX_PATTERN }}
          flags: 'gim'
          text: '${{ steps.reader.outputs.content }}'
  #
  # END of Job 1
  #

  #
  # BEGIN of Job 2
  #

  deploy-to-pypi:
    runs-on: ubuntu-latest
    needs: get-python-version-info

    steps:
      ....
      ....
    # Build the Python package.
    - name: Build package
      run: export GITHUB_PROGRAM_VERSION='${{ needs.get-python-version-info.outputs.my-program-versic
```

Workflow Example: Creating and Deploying Pypi Python Packages

In the following example, we use a workflow to publish a Python package to the Test and Production environments of the Python Package Repository (PyPi). The publishing process will be triggered for pre-releases as well as real releases.

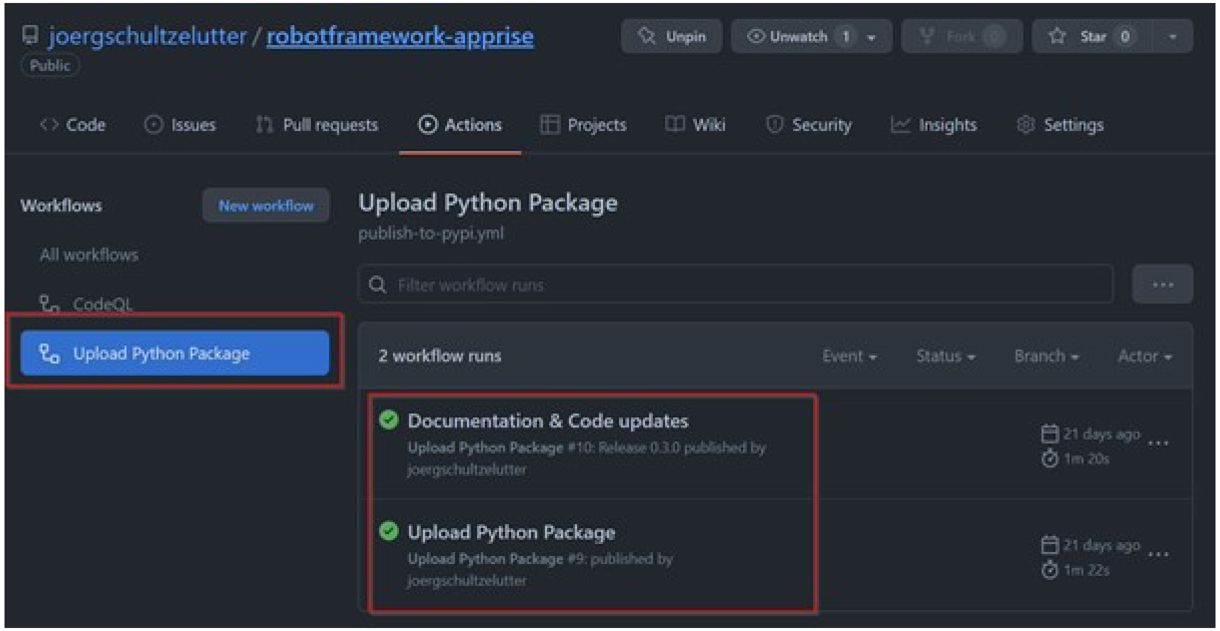
The actual Workflow is divided into two Jobs.

- The first Job (***get-python-version-info***) checks if the version number of the (pre)release in Github is identical to the version info defined in our Python source code and aborts the Workflow if the version data is different.

- If both version infos match, the first job passes that version info to the second job (***publish-to-pypi***) using the **outputs-needs construct**. Our second Job creates the release package and uploads it to PyPi Test. If there is no pre-release but a full release in progress, that created package is also uploaded to PyPi Production.

For both tasks, we use a Workflow I developed. This action requires only a few configuration steps for deployment and activation and is therefore universally usable. All configuration steps are described in detail in the associated repository: https://github.com/joergschultzelutter/pypi-publish-workflow

Here's one of my repositories where I use this Workflow: https://github.com/joergschultzelutter/robotframework-apprise/actions/workflows/publish-to-pypi.yml

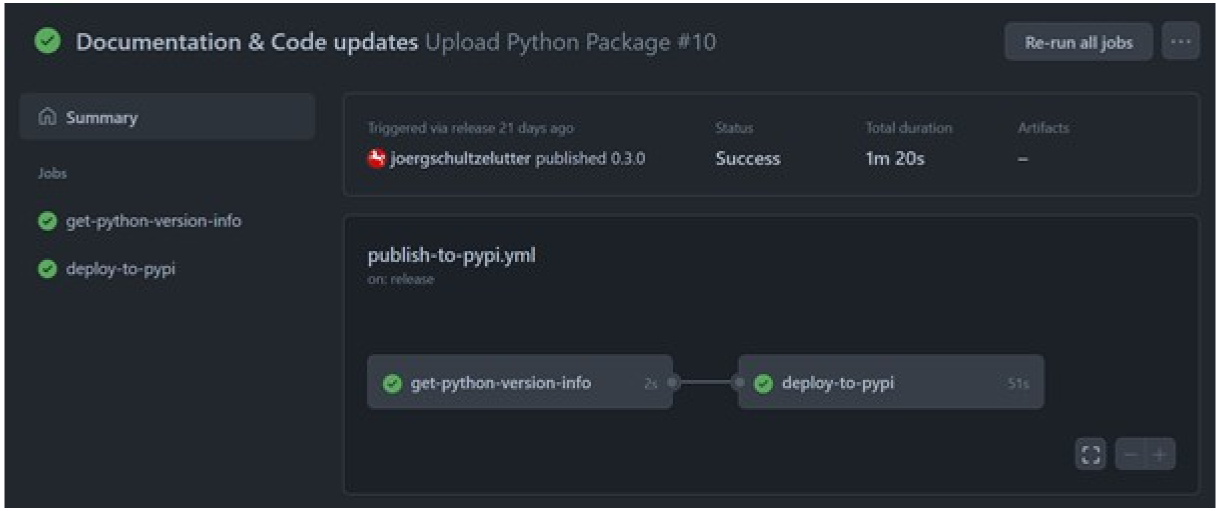


Upload Python Package
publish-to-pypi.yml

Documentation & Code updates
Upload Python Package #10: Release 0.3.0 published by joergschultzelutter
21 days ago
1m 20s

Upload Python Package
Upload Python Package #9: published by joergschultzelutter
21 days ago
1m 22s

On the left side we first see our name of the Workflow (***Upload Python Package***) and on the right side the corresponding actions that have been executed by this Workflow. If we click on one of these Workflow runs, we get access to the two Jobs of our task (**get-python-version-info** and ***deploy-to-pypi***):



✓ Documentation & Code updates Upload Python Package #10

Triggered via release 21 days ago	Status	Total duration	Artifacts
joergschultzelutter published 0.3.0	Success	1m 20s	–

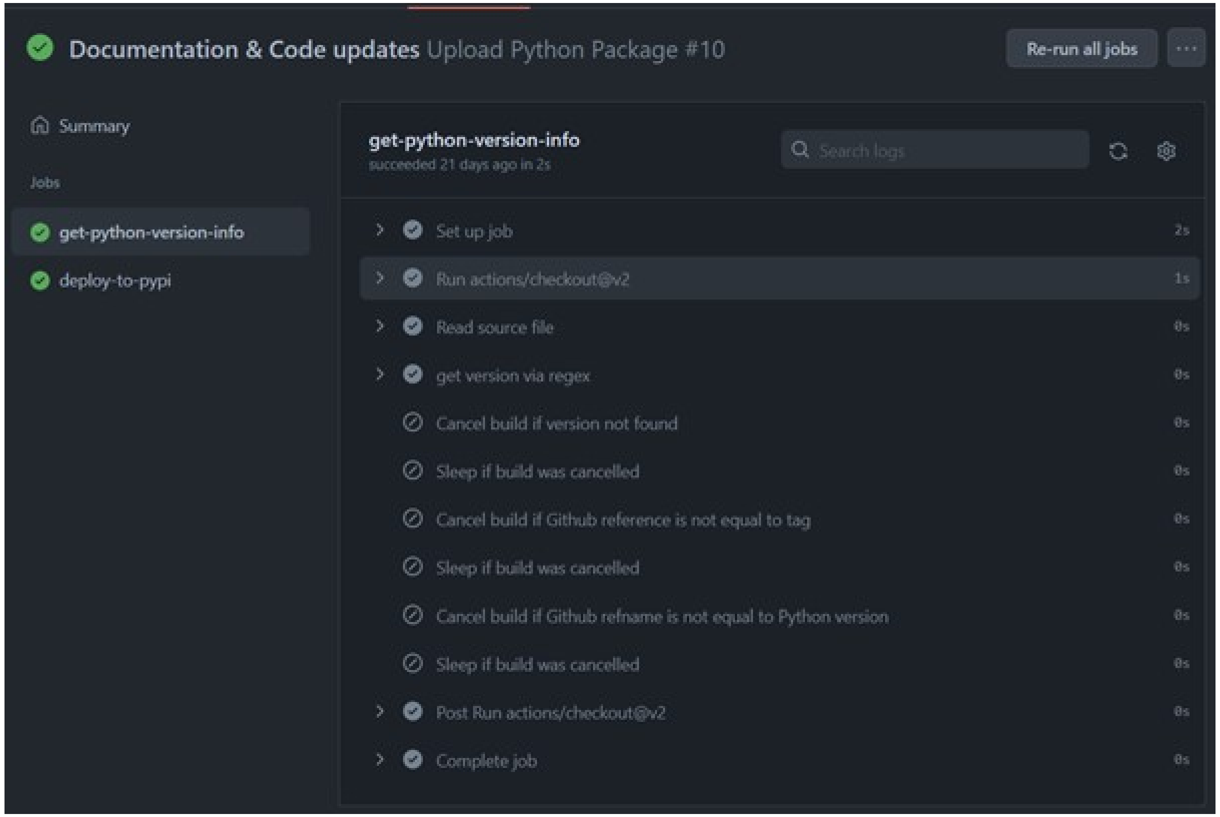
✓ get-python-version-info

✓ deploy-to-pypi

publish-to-pypi.yml
on: release

✓ get-python-version-info 2s — ✓ deploy-to-pypi 51s

These tasks can then be broken down into the individual Steps by clicking on a Jobs:



Documentation & Code updates Upload Python Package #10

get-python-version-info
succeeded 21 days ago in 2s

```
Set up job                                                  2s
Run actions/checkout@v2                                     1s
Read source file                                            0s
get version via regex                                       0s
Cancel build if version not found                           0s
Sleep if build was cancelled                                0s
Cancel build if Github reference is not equal to tag        0s
Sleep if build was cancelled                                0s
Cancel build if Github refname is not equal to Python version 0s
Sleep if build was cancelled                                0s
Post Run actions/checkout@v2                                0s
Complete job                                                0s
```

Steps with a crossed-out circle symbol were not executed.

Check out the documentation of this workflow if you want to learn more about it.

Caveat emptor

Finally, I would like to discuss the most important do's and don'ts in connection with Github Actions:

- If a Workflow is not started despite correct trigger, you should first check not only the actual trigger of that Workflow (*push*, *release*) but also whether **the Github Actions service has been affected by technical problems**.
- Github Actions does not natively provide an option or keyword for Workflows to be canceled / aborted. There is a separate **third-party Action Keyword for this** on the Github Marketplace. It is important to know that the execution of this Github Action

keyword does *not* lead to an immediate termination of the Workflow - you may be forced to use a combination of sleep commands and *If...Then* queries in order to avoid the remaining Steps and Jobs from being executed.

- When using a Github '***on release***' Trigger, the Workflow's Trigger does not distinguish between ***Pre-Releases*** and ***Releases***. If your Workflow is only supposed to be run for Releases, you need to programmatically e.g. abort the Workflow in case of Pre-Releases. Details: have a look at my PyPi Workflow example.
- If your Workflow is going to execute external programs such as Python and these external programs depend on 'exported' environment variables to be set, it should be noted that the lifespan of a variable's export is limited to a single Step within its Job/Workflow. This means that if you cannot export a variable in Step A and use it in Step B. Fortunately, a step can consist of several concatenated commands, meaning that a Step like in the following example is valid:

```
- name: Build package
      run: export GITHUB_PROGRAM_VERSION='${{ needs.get-python-version-info.outputs.my-program-versio
```

The opinions and information stated in this article are personal to the individual author and do not necessarily represent Bertelsmann.

About the Author

JS

Jörg Schultze-Lutter

Arvato

Tags

#automation #github

Share Article

Newest job offers

Senior Backend Engineer (C#/.Net) (m/f/d)

Berlin, BE, DE, 10623

IT Security Engineer (m/f/d)

Berlin, Baden-Baden, Verl, Osl, BE, DE, 10623