An Introduction to   
Continuous Integration (CI)

Contents

[1. Introduction 2](#_Toc148780987)

[2. CI Operations 2](#_Toc148780988)

[3. CI in GitHub 3](#_Toc148780989)

[3.1 CI configuration file 3](#_Toc148780990)

[3.2 Event (on) Section 3](#_Toc148780991)

[3.3 Jobs (job) Section 4](#_Toc148780992)

[3.3.1 CI Job Definition 4](#_Toc148780993)

[3.3.2 Sequential Execution 4](#_Toc148780994)

[3.4 Additional Configurations 5](#_Toc148780995)

[3.4.1 Workflow name 5](#_Toc148780996)

[3.4.2 Environmental variables 5](#_Toc148780997)

[3.4.2 Webhooks 5](#_Toc148780998)

[4. Lab 5 Tasks: Continuous Integration 6](#_Toc148780999)

[4.1 Preparation 6](#_Toc148781000)

[4.2 CI Action Specification 6](#_Toc148781001)

[4.2.1 Create a workflow in github 6](#_Toc148781002)

[4.2.2 Modify the workflow locally 7](#_Toc148781003)

[4.3 CI Action execution 7](#_Toc148781004)

[4.3.1 Execute for Success (Java) 7](#_Toc148781005)

[4.3.2 Execute for Failure (Java) 8](#_Toc148781006)

[4.3.3 Execute for Success (Python) 8](#_Toc148781007)

[4.3.4 Execute for Failure (Python) 8](#_Toc148781008)

[4.4 Use a 2nd workflow 8](#_Toc148781009)

[4.4.1 Create a branch 8](#_Toc148781010)

[4.4.2 Link events to branch pushes 8](#_Toc148781011)

[4.4.3 Execute the 2nd workflow 9](#_Toc148781012)

# Introduction

A CI pipeline is a set of actions that run usually when code or data change and should make sure all of your changes work with the rest of the code when it’s integrated. It should also compile your code, run tests, and check that it’s functional.

(A CD pipeline goes one step further and deploys the built code into production.)

The remainder of this document outlines CI/CD operations in github with the objective of producing standard, end-to-end workflows. For complete documentation on github actions please visit

<https://docs.github.com/en/actions>

# CI Operations

**Continuous integration (CI)** is a software practice that requires frequently committing code to a shared repository.

**Frequent commits**:

* detect errors sooner
* reduces the amount of code to debug when finding the source of an error.
* make it easier to merge changes within the software development team.

This results to **reduced time** on

* debugging errors
* resolving merge conflicts.

Continuously **build and test** the code

Tests include:

* code linters (which check style formatting)
* security checks
* code coverage
* functional tests
* other custom checks.

**In everyday tasks, when I push code:**

**have I broken the build?**

**do any tests fail?**

# CI in GitHub

## 3.1 CI configuration file

In github CI is defined using workflows. Each workflow, a CI configuration file follows the .yaml syntax. This is a simple notation but:

**Please pay attention on the indentation of each line.**

More information on .yaml can be found here:

<https://docs.ansible.com/ansible/latest/reference_appendices/YAMLSyntax.html>

We could have multiple workflows defined. For example, one to run on pushing code on master and another on pushing code in a specific branch.

There are several sections in the CI configuration file, apparently some more important than the others. In this document, we concentrate on two main ones namely, on & jobs and we outline a set of useful ones.

**on** specifies the events that could trigger the execute of the tasks in the **jobs**. There could be multiple entries on each one of them.

The successful execution of jobs is obviously not a given. If there are errors, jobs will fail and this is the main point of running CI workflows, such errors to be detected immediately. It is important to note that this is the github notation. Other CI engines may use different formats. However, the logic is everywhere similar; define events (on) that trigger the execution of tasks (jobs).

## 3.2 Event (on) Section

This section specifies when the actions of the job section should execute, in other words which events trigger the execution of this workflow.

The most commonly used event is **git push**. Execute CI operations whenever a file is pushed. Typically, these changes refer to new code but can also include modified configuration files or data.

**on:**

**push:**

**branches: [ "mybranch" ]**

**release:**

**branches: [ "master" ]**

For example, the code snippet above defines two events. The first one is on any push on branch *mybranch*. The second one on performing a release (more on release on CD) on *master* branch. Consequently, the workflow jobs will be executed either when there is a push on *mybranch* or a release on master. Any push on other branches (including master) or a release on other than *master* will be ignored.

For a full list of github workflow events please see

<https://docs.github.com/en/actions/using-workflows/events-that-trigger-workflows>

## 3.3 Jobs (job) Section

### 3.3.1 CI Job Definition

A CI workflow may contain one or multiple jobs. The setup of each job depends on its nature. A job to build the code should specify the runtime environment.

**jobs:**

**build:**

**runs-on: ubuntu-latest**

**steps:**

**- uses: actions/checkout@v3**

**- name: Set up JDK 11**

**uses: actions/setup-java@v3**

**with:**

**java-version: '13'**

**distribution: 'temurin'**

**cache: maven**

**- name: Build with Maven**

**run: mvn -B package --file pom.xml**

In the above code snippet, the first line under jobs: (build:) is the name of the job.

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Description** | **Examples** |
| **runs-on** | Each job runs in a runner environment specified by runs-on. | runs-on: ubuntu-latest  runs-on:windows-latest  runs-on: ubuntu-22.04 |
| **steps:** | The steps to take for the job execution |  |
| **Uses** | Parameters that specify the job execution | uses: actions/checkout@v3  uses: actions/setup-java@v3  with:  java-version: '13'  distribution: 'temurin'  cache: maven |
| **Run** | The execution of a task | run: mvn -B package --file pom.xml |

### 3.3.2 Sequential Execution

By default, jobs run in parallel. To run jobs sequentially, we define dependencies on other jobs using the jobs.

**jobs:**

**job1:**

**job2:**

**needs: job1**

**job3:**

**needs: [job1, job2]**

## 3.4 Additional Configurations

### 3.4.1 Workflow name

The name of the workflow is specified usually on the top of the workflow. This is the name that appears on the Actions section of github

**name:CI Workflow**

### 3.4.2 Environmental variables

Environment variables (e.g., URL of a database)are defined in the workflow file. The scope of a custom environment variable is limited to the element in which it is defined. You can define environment variables that are scoped for:

* The entire workflow, by using env at the top level of the workflow file.
* The contents of a job within a workflow, by using jobs.<job\_id>.env.
* A specific step within a job, by using jobs.<job\_id>.steps[\*].env.

**env:**

**DAY\_OF\_WEEK: Monday**

**jobs:**

**buildAndTest:**

**steps:**

**- name: Postgress configuration**

**run: node client.js**

**env:**

**POSTGRES\_HOST: postgres**

**POSTGRES\_PORT: 5432**

<https://gitlab.com/gitlab-examples/spring-gitlab-cf-deploy-demo.git>

For full documentation on the syntax of workflows in github please see

<https://docs.github.com/en/actions/using-workflows/workflow-syntax-for-github-actions>

### 3.4.2 Webhooks

Webhooks are mechanisms to link a remote repository (in this case github) to third-party systems. With Webhooks we can trigger POST requests to be send to a URL when certain events happen.

# Lab 5 Tasks: Continuous Integration

## 4.1 Preparation

1. Download the sample project code from canvas.

There are two sample projects. These have the minimum code as our focus is not on coding but on understanding the CI pipelines.

The first one is a skeleton Java Spring boot project with a simple class and a tests. We will use to have a workflow to build on push.

It has two Classes

* DemoApplication the main Java Spring boot application, required to run
* DemoApplicationTests the class that contains tests. We can modify these to experiment with pass/fail situations

The second one is a python project with two files

* app.py which has a method
* tests.py that has a number of tests on the method of app.py

*Alternatively, please feel free to use tour own codebase if it conforms with the execution of the tasks below*

1. Create **a new repository** on github and unzip the project there
2. Use an IDE (e.g., Eclipse) or an editor to view the files. The important ones
3. Java source code file DemoApplication.java
4. Java Junit Test file DemoApplicationTests.java This file is executed when we call maven with test, package, install, deploy. It checks all methods annotated as @Test which will assert a simple test and succeed or fail.
5. pom.xml configuration file. This specifies dependencies to other java libraries (e.g., spring boot and test)

(We will use this in Continuous Integration to enable packaging and deployment into github)

1. Or do the same with the python code.

The project can be build using maven either in the IDE (e.g. Eclipse, IntelliJ, Pycharm) or in the command line.

## 4.2 CI Action Specification

## 4.2.1 Create a workflow in github

1. On the github repository, choose actions from the menu and **New WorkFlow**

Graphical user interface, application, Word, website

Description automatically generated

You will see that github provides many templates and please browse to see what is involved in them.

For this lab purposes, we will create one manually, please choose “Setup a workflow yourself”. Github creates a skeleton code for you in Java or an empty one in Python.

Click on Start Commit to create the skeleton workflow. This now creates a file in git under **.github/workflows**

## 4.2.2 Modify the workflow locally

1. (Java) Pull the code and locate the Workflow created in the folder. Or use the github interface to edit and commit the workflow file

## 4.3 CI Action execution

### 4.3.1 Execute for Success (Java)

The skeleton workflow created, in the jobs section has a few echo statements, therefore it does not perform anything useful.

1. Add build command in the jobs section. For the sample Spring Boot code

**build:**

**runs-on: ubuntu-latest**

**steps:**

**- uses: actions/checkout@v3**

**- name: Set up JDK 11**

**uses: actions/setup-java@v3**

**with:**

**java-version: '13'**

**distribution: 'temurin'**

**cache: maven**

**- name: Build with Maven**

**run: mvn -B package --file pom.xml**

1. Push the changes on the workflow document.
2. Make a small change on any of the source code files and push changes
3. Inside the actions menu on github you should see the workflow being executed and eventually executed successfully

Check the logs of the workflow and identify the different steps.

### 4.3.2 Execute for Failure (Java)

1. Modify any of the source code and introduce a syntax error
2. Push the change
3. Observe that the workflow is executed but fails to build
4. Did you receive an email with the failure? Check that you get an email on the address you used to create the github account

### Execute for Success (Python)

1. For an empty workflow created, use the following code (take care of indentation)

**name: Python package**

**on: [push]**

**jobs:**

**build:**

**runs-on: ubuntu-latest**

**steps:**

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

**- name: Pyhton Test**

**run:**

**python -m unittest tests.test**

push, this workflow. This should trigger the execution and all tests are passed

### 4.3.4 Execute for Failure (Python)

Modify one of the tests in tests.py so it fails. Push again and observe the failure in execution.

## 4.4 Use a 2nd workflow

### 4.4.1 Create a branch

1. create and start using a git branch, including pushing code to the remote repository

### 4.4.2 Link events to branch pushes

1. Create a new workflow and make it executed when code is pushed on the branch. How? Find out!

### 4.4.3 Execute the 2nd workflow

1. Repeat tasks 6.3.1, 6.3.2 for the 2nd workflow
2. What are the expected results?