Test Automation on Continuous Integration Server

In the previous reading unit, we demonstrated how to create tests which we can run using pytest. Typically, we run these while we are creating or refactoring production code, to help us understand when our task is complete.

In this reading unit, we'll put the test and production code into a gitlab repository, and add a configuration file to allow the Continuous Integration Server (in this case, gitlab.com) to run these tests automatically.

Here are the steps required:

Create Git repository

We'll create a git repository for our project on gitlab.com.

- We'll include a README.md file, so that your project is not completely empty. This allows it to be cloned locally.
- We'll add a .gitignore file to the root folder of the repository. This prevents working files from being accidentally added to the repository. It contains a list of filea and folder names, with wildcards. We'll need a .gitignore file that is suitable for Python, e.g. this one here.
- We then clone the repository onto our local machine. This folder will contain the two files in the repository (README.md and .gitignore), and then the system folder .git.

Add Source Files

We will now move our test and production folders (tests and my_utils) into the repository, as well and the pytest.ini file.

- We use an anaconda prompt in the root folder of your repository, use 'git add *' to add all these files (or file changes) to the repository.
- We use 'git status' to check that all the correct files are added to the next proposed version. This should include the '__init__.py' file in the package folder.
- We use 'git commit -m "your message here"' to create a new version of the repository that includes these extra files.
- We use 'git push' to update the origin (in this case, gitlab) with this new version.
- We an check that the project page on gitlab.com includes these updated files.

Create Test Runner Script

To schedule the gitlab.com Continuous Integration server to run these tests (using a 'test runner'), we include a .gitlab-ci.yml file in the root folder of the repository.

In this file, we schedule the jobs to be completed by the test runner, and mark the dependencies between jobs. Here is the reference page.

```
# This file is a template, and might need editing before it works on your project.
# To contribute improvements to CI/CD templates, please follow the Development guide at:
# https://docs.gitlab.com/ee/development/cicd/templates.html
# This specific template is located at:
# https://gitlab.com/gitlab-org/gitlab/-
/blob/master/lib/gitlab/ci/templates/Python.gitlab-ci.yml

# Official language image. Look for the different tagged releases at:
# https://hub.docker.com/r/library/python/tags/
image: python:latest

# Change pip's cache directory to be inside the project directory since we can
# only cache local items.
variables:
  PIP_CACHE_DIR: "$CI_PROJECT_DIR/.cache/pip"

# Pip's cache doesn't store the python packages
# https://pip.pypa.io/en/stable/reference/pip_install/#caching
#
# If you want to also cache the installed packages, you have to install
# them in a virtualenv and cache it as well.
cache:
  paths:
    - .cache/pip
    - venv/

before_script:
  - python -V  # Print out python version for debugging
  - pip install virtualenv
  - virtualenv venv
  - source venv/bin/activate

test:
  script:
    - pip install pytest
    - python -m pytest
```

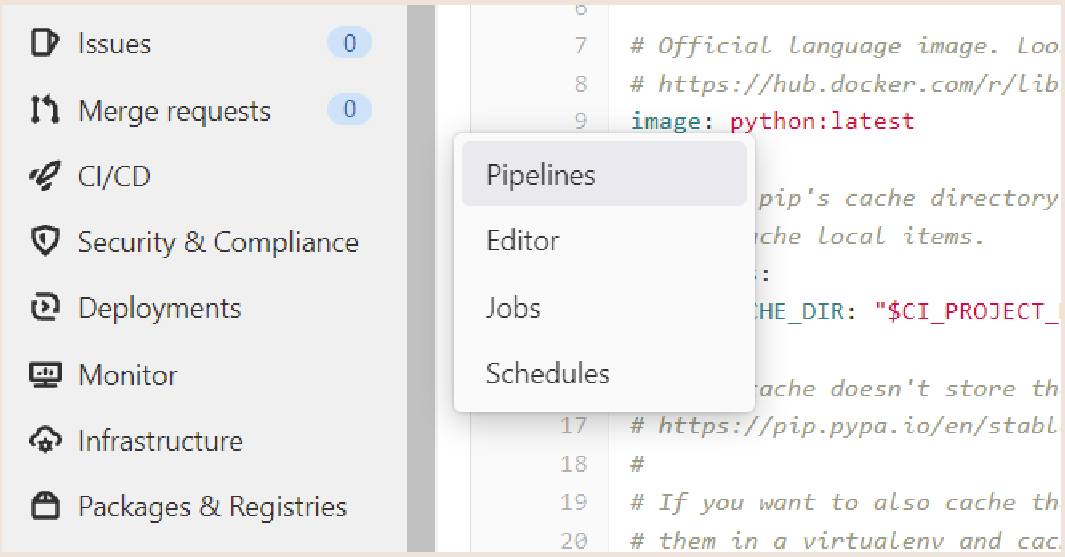
Some notes on this script:

- The 'image' statement is specifying the type of host that we'll be using for running the tests. Here, we've specified the latest stable version of Python.
- The target named 'cache' is to save the earlier installation for subsequent test runs, to save time and bandwidth. It's specifying the folders to be retained ('cached') after running the tests.
- The target named 'before_script' is an in-built target that is built before attempting any of the other targets. Here, we are installing
- The target named 'test' is stating that we should run the test script, which installs and then runs pytest.
- This particular invocation of pytest ('python -m pytest') ensures that the current working directory is added to the Python Path, so that pytest can find the production package.

Once this file is added to the repository (and this change is committed as a new version), then the gitlab.com Continuous Integration Server will attempt to run this script. (In addition, gitlab.com requires authentication from new users)

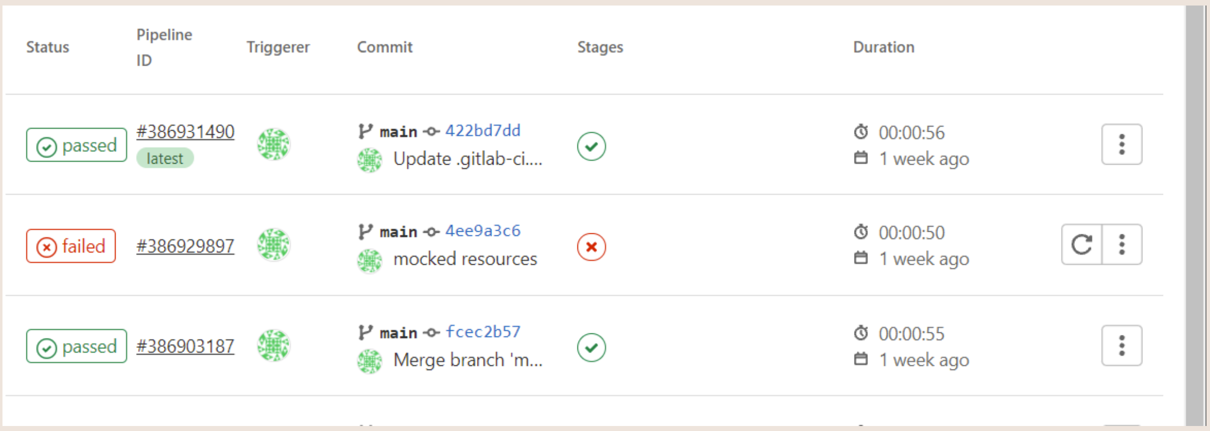
Check Script Execution on CI Server

To check that this file has run, we can access the 'CI/CD Pipelines' page from this project on gitlab.com:



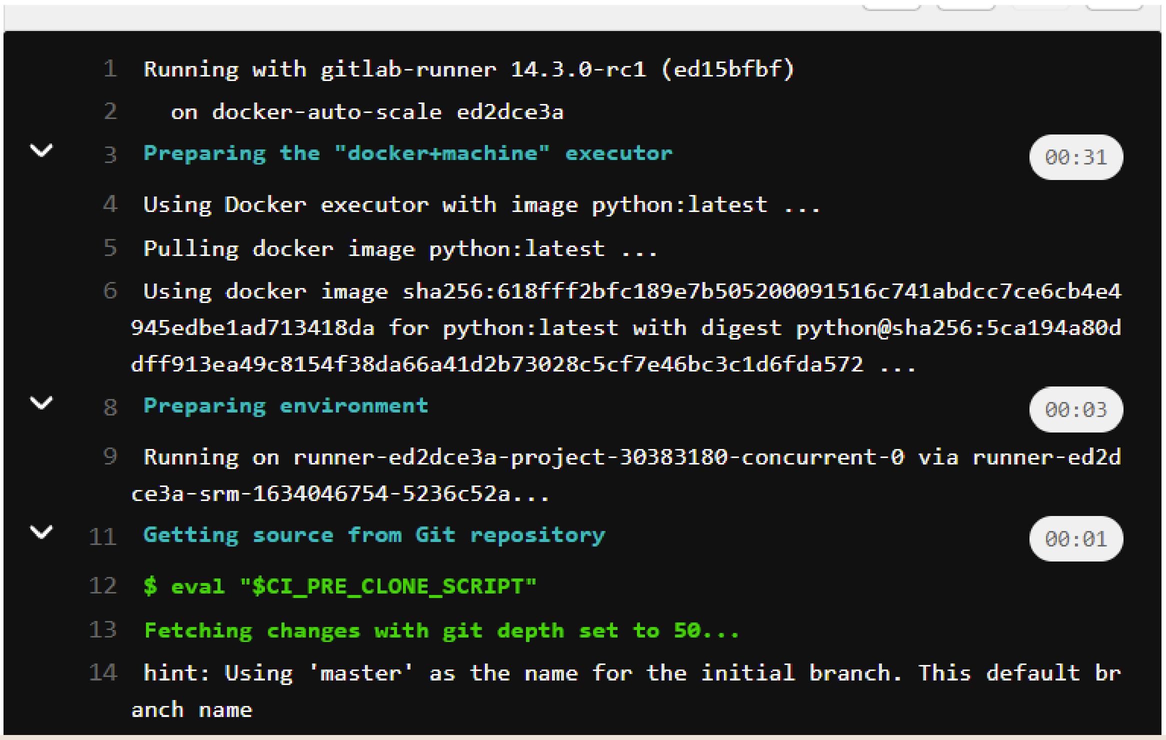
```
# Official language image. Loo
# https://hub.docker.com/r/lib
image: python:latest

      pip's cache directory
     che local items.
        :
     HE_DIR: "$CI_PROJECT_

      ache doesn't store th
# https://pip.pypa.io/en/stabl
#
# If you want to also cache th
# them in a virtualenv and cac
```

This shows that the status of the pipeline, i.e. whether the job has 'succeeded' (in this case, tests passed):



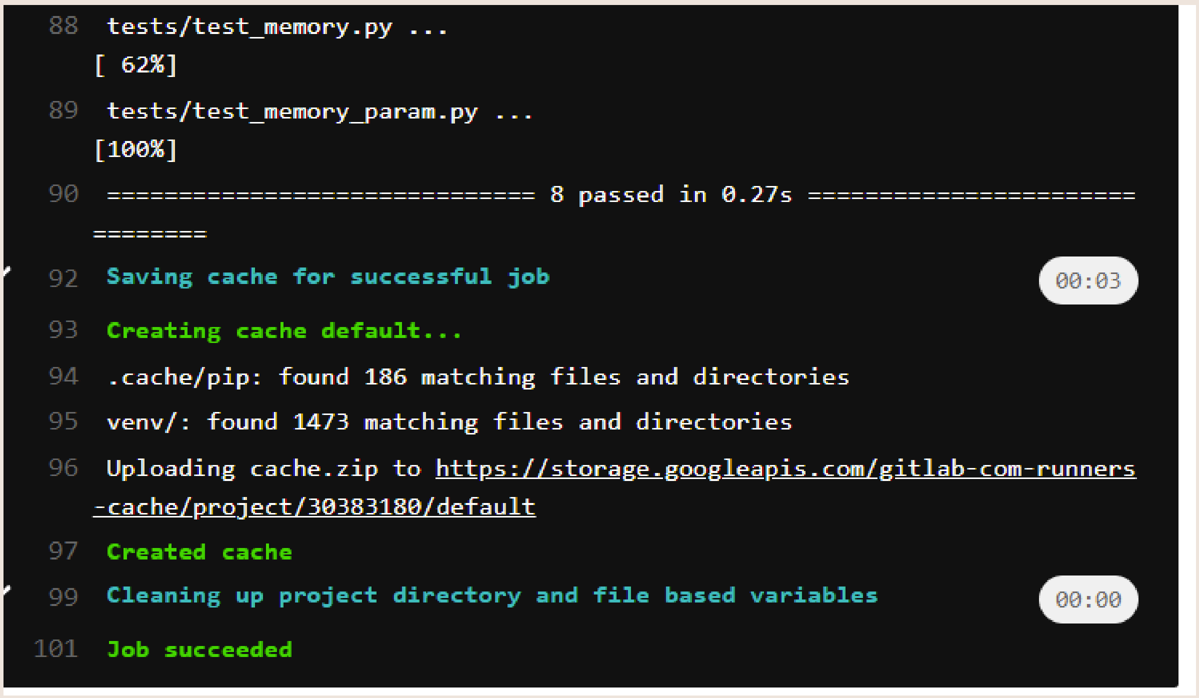
Status	Pipeline ID	Triggerer	Commit	Stages	Duration	
✓ passed	#386931490 latest		main 422bd7dd Update .gitlab-ci....	✓	00:00:56 1 week ago	⋮
✗ failed	#386929897		main 4ee9a3c6 mocked resources	✗	00:00:50 1 week ago	↻ ⋮
✓ passed	#386903187		main fcec2b57 Merge branch 'm...	✓	00:00:55 1 week ago	⋮

Clicking on the 'passed' or 'failed' buttons shows the running of the test. In the example below, the beginning of the pipeline is shown, where it is accessing the latest python image to run the tests.



```
Running with gitlab-runner 14.3.0-rc1 (ed15bfbf)
  on docker-auto-scale ed2dce3a
Preparing the "docker+machine" executor                                00:31
Using Docker executor with image python:latest ...
Pulling docker image python:latest ...
Using docker image sha256:618fff2bfc189e7b505200091516c741abdcc7ce6cb4e4
945edbe1ad713418da for python:latest with digest python@sha256:5ca194a80d
dff913ea49c8154f38da66a41d2b73028c5cf7e46bc3c1d6fda572 ...
Preparing environment                                                  00:03
Running on runner-ed2dce3a-project-30383180-concurrent-0 via runner-ed2d
ce3a-srm-1634046754-5236c52a...
Getting source from Git repository                                     00:01
$ eval "$CI_PRE_CLONE_SCRIPT"
Fetching changes with git depth set to 50...
hint: Using 'master' as the name for the initial branch. This default br
anch name
```

Then, at the end of the test script, it is announcing that the job has succeeded (because pytest reported that all tests had passed.)



```
tests/test_memory.py ...
[ 62%]
tests/test_memory_param.py ...
[100%]
============================== 8 passed in 0.27s =======================
========
Saving cache for successful job                                        00:03
Creating cache default...
.cache/pip: found 186 matching files and directories
venv/: found 1473 matching files and directories
Uploading cache.zip to https://storage.googleapis.com/gitlab-com-runners
-cache/project/30383180/default
Created cache
Cleaning up project directory and file based variables                 00:00
Job succeeded
```

Conclusion and Next Steps

In this reading unit, we've shown how a test script can be included in a repository, so that the unit tests can be run remotely and automatically.

In the following reading units, we'll work on making our automated tests more sophisticated, to work with multiple test cases and with external resources like files and databases.
Before we leave this topic of automatically running tests on a continuous integration server, it's worth pointing out three ways in which this script can be extended, so that it provides value to the development team:

- Our git repository has a single branch called 'master'. Instead, it's good practice to commit changes to a different, 'development' branch, called something like 'test'. We would then associate a 'merge request' with that commit -- literally a request to merge our development back into the master branch. The continuous integration server can be set up to deliver this 'merge request' to the owner of the master branch, but only if all the automated tests are passed by the test runner.
- Our git repository has used a single testing tool, 'pytest', in the 'venv' virtual environment. We can also use additional testing tools such as 'flake8' to check how much our source code complies with the PEP008 guidelines. We can also use more sophisticated virtual test environments such as 'tox'. These topics are discussed further in unit 9.
- Our git repository has a single main target called 'test'. Instead, it's typical to have some further targets that would then go on to be built, if these tests were successfully passed.
    - For example, one target could automatically build the documentation for the project (from the source code).
    - Another example is a target that specifies what needs to happen to deploy the application, for example on a production server. We have then moved from 'Continuous Integration' into 'Continuous Deployment'.