Virtual Pipelines Training “Challenges”

2-4 June 2020

https://indico.cern.ch/event/904759/

Description :
The topics presented here are three “challenge” topics that extend the virtual pipelines training. It assumes that you have thoroughly understood all of the content covered in the training videos created by Kevin and Giordon based in the materials in the following :

- CMS OpenData HTauTau payload
    - https://hsf-training.github.io/hsf-training-cms-analysis-webpage/
- GitLab Pipelines
    - https://hsf-training.github.io/hsf-training-cicd/

If you find yourself struggling on understanding a technical or conceptual aspect of these trainings, we encourage you to focus on these fundamentals. However, if everything is crystal clear, then a wide array of features exist and are beautifully documented in the GitLab docs - https://docs.gitlab.com/ee/ci/ - that can really enhance your CI/CD. These prompts are intended to give you a start and inspire you to try and do something “beyond the lesson”.

Have Questions? :
Start by reaching out on the mattermost workspace to see (Link to Invite Page) if there is a channel with discussion on something related to your interest. And also be sure to check the living google doc (Link to Doc) of frequently asked questions.

Topic 0 : Rinse/Repeat
Write a CI test for each one of the samples in the payload. All of these should be separate CI jobs that run in parallel.

Topic 1 : “only” keyword
Sometimes you have multiple collaborators on a project and you may want a certain test or pipeline to only run on a given branch of a project. One example (below) may be that you only want to build and update documentation pages on the master branch. Or perhaps you know you need to test a suite of backgrounds, but only on the “bkg_dev” branch. In this challenge, use the “only” keyword (go look for it in the documentation linked above) to ensure that one of your tests is only executed when you create a “bkg_dev” branch and push a change to the repo on that branch.

Topic 2 : Pipeline Scheduling
Perhaps your repo is updating software used by others and you want to ensure that it is *always* working and robust. As such, instead of just running your tests when a change is made, go discover how you can schedule automatic runs of your pipeline tests to occur on a

recurring basis, like once per day. (If you aren't sure of where to start, try googling … and then asking on the mattermost)

<u>Topic 3 : dOxygen Documentation Building</u>

Documentation for code is mission critical if you want to move beyond "But how do I get this working?" Would you be able to use any large codebase (e.g. python, ROOT) without it? As such, your code should be documented as well. And this can be done by automatically building dOxygen documentation of your code in a CI test. This can then further be pushed to an EOS (this is CERN specific) space that is linked to a webpage and therefore automatically viewable by your collaborators. This one is a bit advanced because it may require a bit of an appreciation for docker images, but thats what the challenges are for right? So what you should do is put some dOxygen documentation commenting in your code and then write a CI test using this image - *gitlab-registry.cern.ch/cholm/docker-ubuntu-doxygen:latest* - and then pushes it to your personal EOS space. If you already have a CERN webpage setup then pushing it there will make it viewable!

<u>Topic 4 : caches</u>

Caches are powerful tools that allow GitLab to save artifacts and effectively use them between different pipelines. This can be very powerful when trying to do automatic regression tests as you can cache the results (e.g. a cutflow output) from *only* the master branch, and then while working on a dev branch, you can refer back to and retrieve those results that were caches from the master. Then, when your developments are finished, when you push to master, it can automatically cache the new results. Then in the future, the dev branch tests will be referring to the new "gold standard". In this way, we get around having to hardcode cutflow comparisons into our repository and/or could also cache things like histograms for our regression tests. Try doing this for the CMS OpenData payload.

<u>Topic 5 : static code checks</u>

You already learned about setting up a CI to build your code and run jobs. This is great and allows you to check if your code compiles and even allows you to check if your analysis code is doing what you expect it to do. Another way to check e.g. some python code are static code checks. Introducing additional, automatic static code analysis inside your code repository is a very simple way to ensure that your code is clean from known security issues and bad practices. A nice introduction to static code checks is provided by the CERN IT:
https://gitlab.cern.ch/gitlabci-examples/static_code_analysis

<u>Topic 6 : use a docker image to typeset LaTeX</u>

Gitlab CI jobs are extremely flexible, as they can use docker containers to get almost any payload. In the CI/CD tutorial, you noticed the "image" keyword in the CI jobs and were told not to worry about Docker. In fact, Docker containers are extremely useful and allow you to run almost any code in your CI/CD pipeline.
Assuming you are working on a LaTeX document (paper, a thesis, ...) and want GitLab to typeset a PDF as an artifact every time you push a new commit to your project.

All you have to do is to add a .gitlab-ci.yml file to your project. Try to find out using google how to use a LaTeX docker container to automatically typeset your document.

Topic 7 : enforced code style in your project

You may be collaborating with many people on a project, everyone with their own preference on how to format code. Wouldn't it be great if there would be an automated script going over every addition to your common code base nicely formatting it, so that the code base has a consistent layout? You can use tools like "clang" or "black" to automatically format your code. A possible implementation for "clang" is linked here (and the code used to create the docker container is linked here, credits go to Andreas Hoenle).