CERN OPENDATA WORKSHOP
2022

ESCUELA POLITECNICA NACIONAL

XAVIER A. TINTIN

POSSIBLE TASKS

BY XAVIER A. TINTIN

Urna Semper
1234 Main Street
Anytown, State ZIP
www.example.com

PROLOGUE

On September 17th, 2015, the applications from Escuela Politécnica Nacional to collaborate with the European Organization for Nuclear Research to were brought through. It is my pleasure to give a full on documentation of the many workshops to be exploited.

CERN
CMS OPEN DATA WORKSHOP

CLOUD COMPUTING

A physics analysis usually encompasses running over hundreds of gigabytes of data. At CMS, this is usually performed using high-throughput batch systems such as the HTCondor installation at CERN and at other research institutions as well as the worldwide LHC computing grid (WLCG). Not everyone will have these resources available at their own institution, but nowadays anyone can get access to computing resources via public cloud vendors. This lesson will give you a first taste of running realistic physics analyses workflows “in the cloud” using Kubernetes (as well as giving you a brief introduction to Kubernetes itself and some related tools).

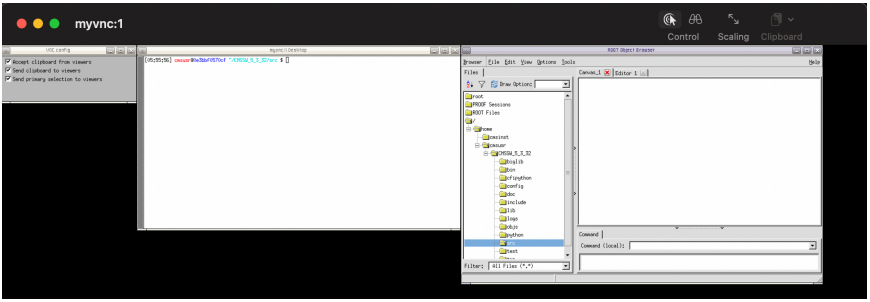
PRE-EXERCISES
DOCKER

A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings.

In short, Docker allows a user to work in a computing environment that has been *frozen* with respect to interdependent libraries and code and related tools. This means that you can use the same software that analysts were using 10 years ago (for example) without downloading all the relevant 10-year-old libraries. :)

```
➜  ~ docker run -it --name my_od -P -p 5901:5901 cmsopendata/cmssw_5_3_32_vnc:latest /bin/bash
WARNING: The requested image's platform (linux/amd64) does not match the detected host platform (linux/arm64/v8) and n
o specific platform was requested
[05:55:24] cmsusr@0e3bbf0570cf ~/CMSSW_5_3_32/src $ start_vnc

You will require a password to access your desktops.

Password:
Verify:
xauth:  file /home/cmsusr/.Xauthority does not exist

New 'myvnc:1' desktop is 0e3bbf0570cf:1

Starting applications specified in /home/cmsusr/.vnc/xstartup
Log file is /home/cmsusr/.vnc/0e3bbf0570cf:1.log

VNC connection points:
       VNC viewer address: 127.0.0.1:5901
       OSX built-in VNC viewer command: open vnc://127.0.0.1:5901
To kill the vncserver enter 'vncserver -kill :1'
```

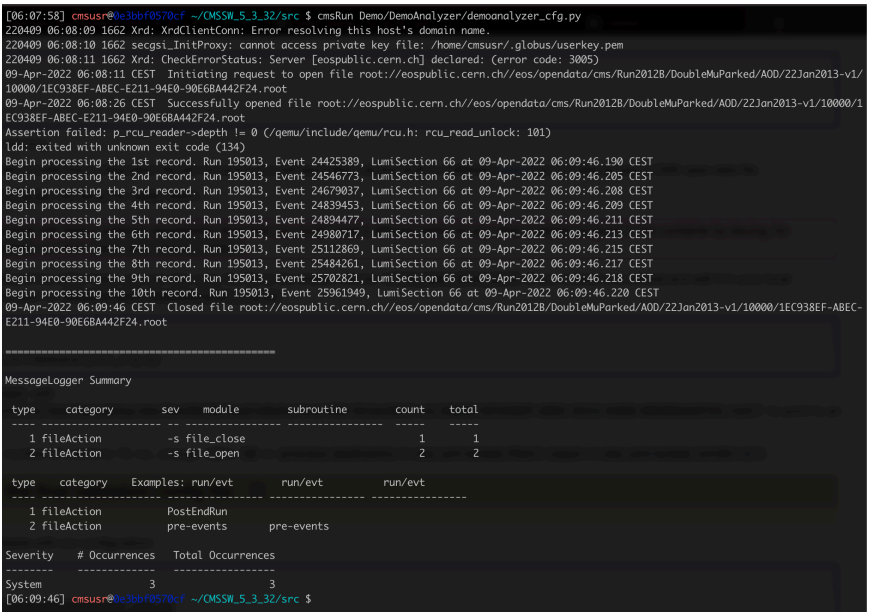
```
[05:55:54] cmsusr@0e3bbf0570cf ~/CMSSW_5_3_32/src $ root
Couldn't find font "-adobe-helvetica-medium-r-*-*-10-*-*-*-*-*-iso8859-1",
trying "fixed". Please fix your system so helvetica can be found,
this font typically is in the rpm (or pkg equivalent) package
XFree86-[75,100]dpi-fonts or fonts-xorg-[75,100]dpi.
  ****************************************
  *                                      *
  *        W E L C O M E  to  R O O T    *
  *                                      *
  *   Version   5.32/00   2 December 2011   *
  *                                      *
  *  You are welcome to visit our Web site  *
  *          http://root.cern.ch         *
  *                                      *
  ****************************************

ROOT 5.32/00 (branches/v5-32-00-patches@42372, Jun 10 2014, 18:26:00 on linuxx8664gcc)

CINT/ROOT C/C++ Interpreter version 5.18.00, July 2, 2010
Type ? for help. Commands must be C++ statements.
Enclose multiple statements between { }.
root [0] TBrowser t
root [1] 1+1
(const int)2
root [2]
```




```
root [2] .q
[05:59:22] cmsusr@0e3bbf0570cf ~/CMSSW_5_3_32/src $ vncserver -kill :1
Killing Xvnc process ID 283
[05:59:35] cmsusr@0e3bbf0570cf ~/CMSSW_5_3_32/src $ exit
exit
➜  ~
```

```
➜  ~ docker start -i my_od
Setting up CMSSW_5_3_32
Waiting for release information to be obtained via https://cmssdt.cern.ch/SDT/releases.map (timeout in 7s)
WARNING: There already exists /home/cmsusr/CMSSW_5_3_32 area for SCRAM_ARCH slc6_amd64_gcc472.
CMSSW should now be available.
[06:01:07] cmsusr@0e3bbf0570cf ~/CMSSW_5_3_32/src $ cmsenv
[06:01:13] cmsusr@0e3bbf0570cf ~/CMSSW_5_3_32/src $ mkdir Demo
[06:01:19] cmsusr@0e3bbf0570cf ~/CMSSW_5_3_32/src $ cd Demo/
[06:01:22] cmsusr@0e3bbf0570cf ~/CMSSW_5_3_32/src/Demo $ mkedanlzr DemoAnalyzer
  I: using skeleton: /opt/cms/slc6_amd64_gcc472/cms/cmssw/CMSSW_5_3_32/bin/slc6_amd64_gcc472/mkTemplates/EDAnalyzer/CfiFile_cfi.py
  I: authors name is: , determined by the gcos entry
  I: creating file: DemoAnalyzer/python/demoanalyzer_cfi.py
  I: using skeleton: /opt/cms/slc6_amd64_gcc472/cms/cmssw/CMSSW_5_3_32/bin/slc6_amd64_gcc472/mkTemplates/EDAnalyzer/edanalyzer.cc
  I: authors name is: , determined by the gcos entry
  I: creating file: DemoAnalyzer/src/DemoAnalyzer.cc
  I: using skeleton: /opt/cms/slc6_amd64_gcc472/cms/cmssw/CMSSW_5_3_32/bin/slc6_amd64_gcc472/mkTemplates/EDAnalyzer/BuildFile.temp
  I: authors name is: , determined by the gcos entry
  I: creating file: DemoAnalyzer/BuildFile.xml
  I: using skeleton: /opt/cms/slc6_amd64_gcc472/cms/cmssw/CMSSW_5_3_32/bin/slc6_amd64_gcc472/mkTemplates/EDAnalyzer/ConfFile_cfg.py
  I: authors name is: , determined by the gcos entry
  I: creating file:  DemoAnalyzer/demoanalyzer_cfg.py
[06:01:29] cmsusr@0e3bbf0570cf ~/CMSSW_5_3_32/src/Demo $ cd ..
[06:01:34] cmsusr@0e3bbf0570cf ~/CMSSW_5_3_32/src $ scram b
****WARNING: No need to export library once you have declared your library as plugin. Please cleanup src/Demo/DemoAnalyzer/BuildFile by removin
g the <export></export> section.
>> Local Products Rules ..... started
>> Local Products Rules ..... done
>> Building CMSSW version CMSSW_5_3_32 ----
>> Entering Package Demo/DemoAnalyzer
>> Creating project symlinks
  src/Demo/DemoAnalyzer/python -> python/Demo/DemoAnalyzer
```

```
docker start -i my_od                                                    ⌥⌘1
 GNU nano 2.0.9                File: Demo/DemoAnalyzer/demoanalyzer_cfg.py                Modified

import FWCore.ParameterSet.Config as cms

process = cms.Process("Demo")

process.load("FWCore.MessageService.MessageLogger_cfi")

process.maxEvents = cms.untracked.PSet( input = cms.untracked.int32(10) )

process.source = cms.Source("PoolSource",
    # replace 'myfile.root' with the source file you want to use
    fileNames = cms.untracked.vstring(
        'root://eospublic.cern.ch//eos/opendata/cms/Run2012B/DoubleMuParked/AOD/22Jan2013-v1/10000/1EC938EF-ABEC-E211-94E0-90E6BA442F24.root'
    )
)

process.demo = cms.EDAnalyzer('DemoAnalyzer'
)


process.p = cms.Path(process.demo)

^G Get Help  ^O WriteOut  ^R Read File ^Y Prev Page ^K Cut Text  ^C Cur Pos
^X Exit      ^J Justify   ^W Where Is  ^V Next Page ^U UnCut Text ^T To Spell
```



- The CMS Docker image contains all the required ingredients to start analyzing CMS open data.
- In order to test and validate the Docker container you can run a simple CMSSW job.

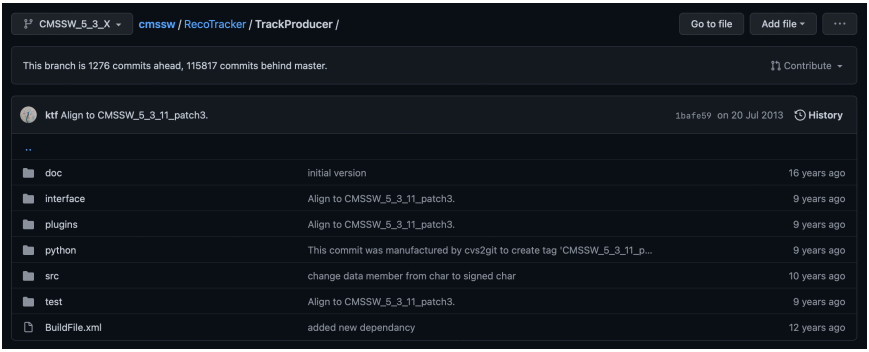
CMSSW

The CMS Software (CMSSW) is a collection of software libraries that the CMS experiment uses in order to acquire, produce, process and even analyze its data. The program is written in C++ but its configuration is manipulated using the Python language.

CMSSW is built around a Framework, an Event Data Model (EDM), and Services needed by the simulation, calibration and alignment, and reconstruction modules that process event data so that physicists can perform analysis. The primary goal of the Framework and EDM is to facilitate the development and deployment of reconstruction and analysis software.

In this workshop, we are only going to look at EDAnalyzers.

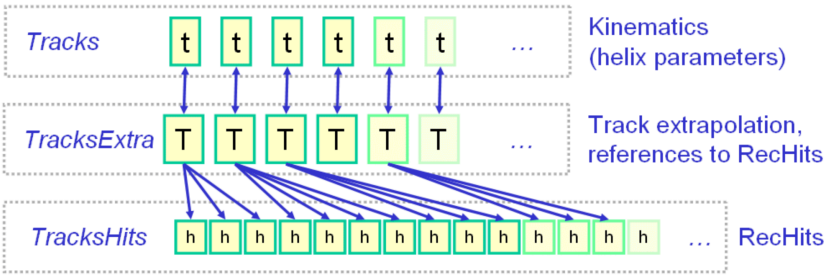
As an example of modularity, take a look at the package used for reconstructing tracks. It has many sub-packages that put in evidence the many bits involved in making a track from detector sensor information. One of those sub-packages is the TrackProducer, which is in charge of putting (recording) the track information in the event. Note the structure of this sub-package:



It has the usual look of a C++ repository. Commonly, you can find a src directory with the bulk of the C++ programming (*.cc files), an interface directory with mostly the *header*files (*.h files) matching the code in the src, and a python directory, where configuration files, written in Python, are stored. Some other *accesories* are in other directories. Of course, this is not a standard rule, and many times the structure follow a different logic. There is also a Buildfile, which controls the package dependencies.

All these packages are, in a sense, plugins to the main Framework, which is also a package by itself.

The event data architecture is modular, just as the framework is. Different data layers (using different data formats) can be configured, and a given application can use any layer or layers. The following diagram illustrates this concept if one thinks about how the information from tracks of charged particles is organized:



All the information regarding the physics of a collision is stored in the **Event**. Computationally, one can think of the Event as an object from which you can pull all the information you need from the collision.

CMS uses different data formats, which are arranged in tiers. Currently CMS open data comes only in the AOD format, therefore that is the format we will be mostly using in this workshop.

- The CMS SoftWare (CMSSW) is the software used by the CMS experiment for acquiring, producing, processing and analyzing its data.

- CMSSW is built in a modular fashion around a main Framework.

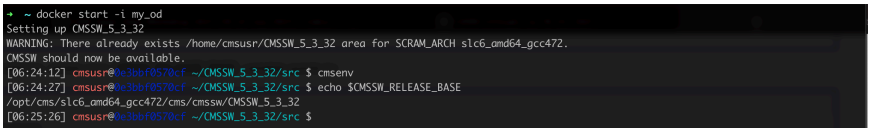
INSTALLATION AND EXECUTION

If you completed Docker you should already have a working CMSSW area.

Note that we are not really “installing” CMSSW but setting up an environment for it. CMSSW was already installed. This is why **every time** you open a new shell you will have to issue the **cmsenv** command, which is just a script that runs to set some environmental variables for your working area:

Now you can check, for instance, where your CMSSW_RELEASE_BASE variable points to:
echo $CMSSW_RELEASE_BASE

The variable may point to a local CMSSW install if you are using a Docker container:



All the packages that comprise the CMSSW release in use have been already compiled and linked to one single **executable**, which is called cmsRun. So, unless you want to create your own plugin (addition) for the software, you won’t even have to re-compile. You can actually try to execute this command by itself, but it will give you a configuration error:

```
[06:25:26] cmsusr@0e3bbf0570cf ~/CMSSW_5_3_32/src $ cmsRun
cmsRun: No configuration file given.
For usage and an options list, please do 'cmsRun --help'.
```

So, inevitably, the cmsRun executable needs a configuration file. This configuration file must be written in Python.

We could simply repeat what we already did while setting up our VM or container: run with the Demo/DemoAnalyzer/demoanalyzer_cfg.py python configuration file. However, this time we could store the output in a dummy.log file and run it in the background. Notice the bash redirector >, the redirection of stderr to stdout (2>&1), and the trailing run-in-the-background control operator &: cmsRun Demo/DemoAnalyzer/demoanalyzer_cfg.py > dummy.log 2>&1 &

You can check the development of your job with: tail -f dummy.log

```
[06:33:06] cmsusr@0e3bbf0570cf ~/CMSSW_5_3_32/src $ cat dummy.log
220409 06:29:22 240 Xrd: XrdClientConn: Error resolving this host's domain name.
220409 06:29:24 240 Xrd: CheckErrorStatus: Server [eospublic.cern.ch] declared: (error code: 3005)
09-Apr-2022 06:29:24 CEST  Initiating request to open file root://eospublic.cern.ch//eos/opendata/cms/Run2012B/DoubleMuParked/AOD/22Jan2013-v1/
10000/1EC938EF-ABEC-E211-94E0-90E6BA442F24.root
09-Apr-2022 06:29:39 CEST  Successfully opened file root://eospublic.cern.ch//eos/opendata/cms/Run2012B/DoubleMuParked/AOD/22Jan2013-v1/10000/1
EC938EF-ABEC-E211-94E0-90E6BA442F24.root
Assertion failed: p_rcu_reader->depth != 0 (/qemu/include/qemu/rcu.h: rcu_read_unlock: 101)
ldd: exited with unknown exit code (134)
Begin processing the 1st record. Run 195013, Event 24425389, LumiSection 66 at 09-Apr-2022 06:31:01.296 CEST
Begin processing the 2nd record. Run 195013, Event 24546773, LumiSection 66 at 09-Apr-2022 06:31:01.313 CEST
Begin processing the 3rd record. Run 195013, Event 24679037, LumiSection 66 at 09-Apr-2022 06:31:01.314 CEST
Begin processing the 4th record. Run 195013, Event 24839453, LumiSection 66 at 09-Apr-2022 06:31:01.315 CEST
Begin processing the 5th record. Run 195013, Event 24894477, LumiSection 66 at 09-Apr-2022 06:31:01.317 CEST
Begin processing the 6th record. Run 195013, Event 24980717, LumiSection 66 at 09-Apr-2022 06:31:01.318 CEST
Begin processing the 7th record. Run 195013, Event 25112869, LumiSection 66 at 09-Apr-2022 06:31:01.320 CEST
Begin processing the 8th record. Run 195013, Event 25484261, LumiSection 66 at 09-Apr-2022 06:31:01.321 CEST
Begin processing the 9th record. Run 195013, Event 25702821, LumiSection 66 at 09-Apr-2022 06:31:01.322 CEST
Begin processing the 10th record. Run 195013, Event 25961949, LumiSection 66 at 09-Apr-2022 06:31:01.324 CEST
09-Apr-2022 06:31:01 CEST  Closed file root://eospublic.cern.ch//eos/opendata/cms/Run2012B/DoubleMuParked/AOD/22Jan2013-v1/10000/1EC938EF-ABEC-
E211-94E0-90E6BA442F24.root

====================================

MessageLogger Summary

 type     category        sev    module        subroutine        count    total
 ---- -------------------- -- ---------------- ----------------  -----    -----
    1 fileAction           -s file_close                           1        1
    2 fileAction           -s file_open                            2        2

 type    category    Examples: run/evt        run/evt          run/evt
 ---- -------------------- --------------- ---------------- ----------------
    1 fileAction           PostEndRun
    2 fileAction           pre-events       pre-events

Severity    # Occurrences   Total Occurrences
--------    -------------   -----------------
System                3                 3
[06:33:16] cmsusr@0e3bbf0570cf ~/CMSSW_5_3_32/src $
```

We use scram, the release management tool used for CMSSW, to compile (**b**uild) the code:

```
[06:33:16] cmsusr@0e3bbf0570cf ~/CMSSW_5_3_32/src $ scram b
Reading cached build data
>> Local Products Rules ..... started
>> Local Products Rules ..... done
>> Building CMSSW version CMSSW_5_3_32 ----
>> Entering Package Demo/DemoAnalyzer
>> Creating project symlinks
  src/Demo/DemoAnalyzer/python -> python/Demo/DemoAnalyzer
>> Leaving Package Demo/DemoAnalyzer
>> Package Demo/DemoAnalyzer built
>> Subsystem Demo built
>> Local Products Rules ..... started
>> Local Products Rules ..... done
gmake[1]: Entering directory `/home/cmsusr/CMSSW_5_3_32'
>> Creating project symlinks
  src/Demo/DemoAnalyzer/python -> python/Demo/DemoAnalyzer
>> Done python_symlink
>> Compiling python modules cfipython/slc6_amd64_gcc472
>> Compiling python modules python
>> Compiling python modules src/Demo/DemoAnalyzer/python
>> All python modules compiled
>> Pluging of all type refreshed.
>> Done generating edm plugin poisoned information
gmake[1]: Leaving directory `/home/cmsusr/CMSSW_5_3_32'
[06:34:08] cmsusr@0e3bbf0570cf ~/CMSSW_5_3_32/src $
```

Note that scram only goes into the Demo/DemoAnalyzer package that we created locally to validate our setup in a previous lesson. All the rest of the packages in the release were already compiled. Since there is nothing new to compile, it finishes very quickly. In a later episode we will modify this DemoAnalyzer and will need to compile again.

Point to be made: if you compile at main src level, all the packages in there will compiled. However, if you go inside a specific package or sub-package, like our Demo/DemoAnalyzer, only the code in that subpackage will be compiled.

Note: if you are using a soft link to an area that is perhaps mounted from the host machine (like in the example from the Docker lesson), you must compile at the main src level, i.e., /home/cmsusr/CMSSW_5_3_32/src, otherwise the compilation will fail. A small price to pay for convenience.

One can find out about other scripts like mkedanlzr by typing mked and hitting the Tab key:

```
[06:38:12] cmsusr@0e3bbf0570cf ~/CMSSW_5_3_32/src/Demo/DemoAnalyzer $ mked
mkedanlzr  mkedfltr   mkedlpr    mkedprod
[06:38:12] cmsusr@0e3bbf0570cf ~/CMSSW_5_3_32/src/Demo/DemoAnalyzer $
```

To find the EventSize of a ROOT EDM file, use one of the EDM utilities mentioned above to find out about the number of events in the ROOT file that is in the Demo/DemoAnalyzer/demoanalyzer_cfg.py config file of your analyzer package.

```
[06:39:44] cmsusr@0e3bbf0570cf ~/CMSSW_5_3_32/src $
DoubleMuParked/AOD/22Jan2013-v1/10000/1EC938EF-ABEC-E211-94E0-90E6BA442F24.root |grep Events//eos/opendata/cms/Run2012B/
Assertion failed: p_rcu_reader->depth != 0 (/qemu/include/qemu/rcu.h: rcu_read_unlock: 101)
ldd: exited with unknown exit code (134)
File root://eospublic.cern.ch//eos/opendata/cms/Run2012B/DoubleMuParked/AOD/22Jan2013-v1/10000/1EC938EF-ABEC-E211-94E0-90E6BA442F24.root Events
 12279
[06:42:11] cmsusr@0e3bbf0570cf ~/CMSSW_5_3_32/src $
```

So the ROOT file has 12279 events.

- A CMSSW area is not really installed but set up.
- cmsRun is the CMSSW executable. There are also utilitarian scripts.
- You can compile CMSSW with scram b

EDANALYZERS

EDAnalyzers are modules that allow read-only access to the Event. They are useful to produce histograms, reports, statistics, etc. Take a look at the DemoAnalyzer package that we created while validating our CMSSW working environment; it is an example of an EDAnalyzer.

Let's explore the DemoAnalyzer package:

```
[06:42:11] cmsusr@0e3bbf0570cf ~/CMSSW_5_3_32/src $ ls Demo/DemoAnalyzer/
BuildFile.xml  demoanalyzer_cfg.py  doc  interface  python  src  test
[06:44:49] cmsusr@0e3bbf0570cf ~/CMSSW_5_3_32/src $ ls Demo/DemoAnalyzer/src/
DemoAnalyzer.cc
[06:45:03] cmsusr@0e3bbf0570cf ~/CMSSW_5_3_32/src $
```

Note that it has a similar structure as any of the CMSSW packages we mentioned before. In this sense, our DemoAnalyzer is just one more CMSSW package. However, the headers and implementation of our simple DemoAnalyzer are coded in one single file under the src directory. The file was automatically named DemoAnalyzer.cc

CMSSW could be very picky about the structure of its packages. Most of the time, scripts or other tools expect to have a Package/SubPackage structure, just like our Demo/DemoAnalyzer example.

We also notice we have a python configuration file called demoanalyzer_cfg.py (unlike its cousins, it is not inside

the python directory). This is the default configurator for the DemoAnalyzer.cc code.

Finally, there is a BuildFile.xml, where we can include any dependencies if needed.

All EDAnalyzers are created equal; of course, if made with the same mkedanlzr, they will look identical. The DemoAnalyzer.cc is a skeleton, written in C++, that contains all the basic ingredients to use CMSSW libraries. So, in order to perform a physics analysis, and extract information from our CMS open data, we just need to understand what to add to this code and how to configure it.

- An EDAnalyzer is a an edm class that generates a template for any analysis code using CMSSW.
- There are essentially three important files in an EDAnalyzer package, the source code in c++, the python config file and a Buildfile for tracking dependencies.

THE SOURCE

Playing with the DemoAnalyzer.cc file

The DemoAnalyzer.cc file is the main file of our EDAnalyzer. As it was mentioned, the default structure is always the same. Let's look at what is inside using an editor like nano:

```
// system include files
#include <memory>

// user include files
#include "FWCore/Framework/interface/Frameworkfwd.h"
#include "FWCore/Framework/interface/EDAnalyzer.h"

#include "FWCore/Framework/interface/Event.h"
#include "FWCore/Framework/interface/MakerMacros.h"

#include "FWCore/ParameterSet/interface/ParameterSet.h"
//
// class declaration
//
```

These are the most basic *Framework* classes that are needed to mobilize the CMSSW machinery. In particular, notice the Event.h class. This class contains essentially all the *accessors* that are needed to extract information from *the Event*, i.e., from the particle collision. Another important class is the ParameterSet.h. This one will allow us to extract

configuration parameters, which can be manipulated using the Demo/DemoAnalyzer/demoanalyzer_cfg.py python file.

Something important to take into account is that you can learn a lot about the kind of information you have access to by exploring the code in the CMSSW repository on Github. For instance, you can look at the Event.h header and check all the available methods. You will notice, for instance, the presence of the getByLabel accessors; we will be using one these to access physics objects.

When exploring CMSSW code on Github, remember to choose the CMSSW_5_3_32 branch.

Including muon headers

Let's pretend that we are interested in extracting the energy of all the muons in the event. We would need to add the appropriate classes for this. After quickly reviewing this chapter of the CMS Open Data Guide (which is still under construction), we conclude that we need to add these two header lines to our analyzer:

//classes to extract Muon information
#include "DataFormats/MuonReco/interface/Muon.h"
#include "DataFormats/MuonReco/interface/MuonFwd.h"

```
// system include files
#include <memory>

// user include files
#include "FWCore/Framework/interface/Frameworkfwd.h"
#include "FWCore/Framework/interface/EDAnalyzer.h"

#include "FWCore/Framework/interface/Event.h"
#include "FWCore/Framework/interface/MakerMacros.h"

#include "FWCore/ParameterSet/interface/ParameterSet.h"


//classes to extract Muon information
#include "DataFormats/MuonReco/interface/Muon.h"
#include "DataFormats/MuonReco/interface/MuonFwd.h"

#include<vector>
```

Next, you will see the class declaration:

The first thing one notices is that our class inherits from the edm::EDAnalyzer class. It follows the same structure as any class in C++. The declaration of the methods reflect the functionality needed for particle physics analysis. Their implementation are further below in the same file.

Declaring info containers

Let's add the declaration of a vector for our energy values: std::vector<float> muon_e;

```
//
// class declaration
//

class DemoAnalyzer : public edm::EDAnalyzer {
   public:
      explicit DemoAnalyzer(const edm::ParameterSet&);
      ~DemoAnalyzer();

      static void fillDescriptions(edm::ConfigurationDescriptions& descriptions);


   private:
      virtual void beginJob() ;
      virtual void analyze(const edm::Event&, const edm::EventSetup&);
      virtual void endJob() ;

      virtual void beginRun(edm::Run const&, edm::EventSetup const&);
      virtual void endRun(edm::Run const&, edm::EventSetup const&);
      virtual void beginLuminosityBlock(edm::LuminosityBlock const&, edm::EventSetup const&);
      virtual void endLuminosityBlock(edm::LuminosityBlock const&, edm::EventSetup const&);

      // ----------member data ---------------------------
      std::vector<float> muon_e; //energy values for muons in the event
};
```

Next, we can see the constructor and destructor of our DemoAnalyzer class:

Note that a ParameterSet object is passed to the constructor. This is then the place where we will read any configuration we might end up implementing through our Demo/DemoAnalyzer/demoanalyzer_cfg.py python configuration file.

```
//
// constructors and destructor
//
DemoAnalyzer::DemoAnalyzer(const edm::ParameterSet& iConfig)

{
   //now do what ever initialization is needed

}


DemoAnalyzer::~DemoAnalyzer()
{

   // do anything here that needs to be done at desctruction time
   // (e.g. close files, deallocate resources etc.)

}
```

The heart of the source file is the analyze method:

Anything that goes inside this routine will loop over all available events. The CMSSW Framework will take care of that, so you do not really have to write a for loop to go over all events. Note that an edm::Event object and a edm::EventSetup object are passed by default. While from the Event we can extract information like physics objects, from the EventSetup we can get information like trigger prescales.

```
// ------------ method called for each event  ------------
void
DemoAnalyzer::analyze(const edm::Event& iEvent, const edm::EventSetup& iSetup)
{
   using namespace edm;



#ifdef THIS_IS_AN_EVENT_EXAMPLE
   Handle<ExampleData> pIn;
   iEvent.getByLabel("example",pIn);
#endif

#ifdef THIS_IS_AN_EVENTSETUP_EXAMPLE
   ESHandle<SetupData> pSetup;
   iSetup.get<SetupRecord>().get(pSetup);
#endif
}
```

Get the muons energy

Now let's add a few lines in the analyzer so we can retrieve the energy of all the muons in each event. We will print out this information as an example. Again, after checking out this guide, the analyze method becomes:

```
// ------------ method called for each event  ------------
void
DemoAnalyzer::analyze(const edm::Event& iEvent, const edm::EventSetup& iSetup)
{
   using namespace edm;
   //clean the container
   muon_e.clear();

   //define the handler and get by label
   Handle<reco::MuonCollection> mymuons;
   iEvent.getByLabel("muons", mymuons);

   //if collection is valid, loop over muons in event
   if(mymuons.isValid()){
      for (reco::MuonCollection::const_iterator itmuon=mymuons->begin(); itmuon!=mymuons->end(); ++itmuon){
          muon_e.push_back(itmuon->energy());
      }
   }

   //print the vector
   for(unsigned int i=0; i < muon_e.size(); i++){
      std::cout <<"Muon # "<<i<<" with E = "<<muon_e.at(i)<<" GeV."<<std::endl;
   }


#ifdef THIS_IS_AN_EVENT_EXAMPLE
   Handle<ExampleData> pIn;
   iEvent.getByLabel("example",pIn);
#endif

#ifdef THIS_IS_AN_EVENTSETUP_EXAMPLE
   ESHandle<SetupData> pSetup;
   iSetup.get<SetupRecord>().get(pSetup);
#endif
}
```

The other methods are designed to execute instructions according to their own name description.

```
// ------------ method called once each job just before starting event loop  ------------
void
DemoAnalyzer::beginJob()
{
}

// ------------ method called once each job just after ending the event loop  ------------
void
DemoAnalyzer::endJob()
{
}

// ------------ method called when starting to processes a run  ------------
void
DemoAnalyzer::beginRun(edm::Run const&, edm::EventSetup const&)
{
}

// ------------ method called when ending the processing of a run  ------------
void
DemoAnalyzer::endRun(edm::Run const&, edm::EventSetup const&)
{
}

// ------------ method called when starting to processes a luminosity block  ------------
void
DemoAnalyzer::beginLuminosityBlock(edm::LuminosityBlock const&, edm::EventSetup const&)
{
}

// ------------ method called when ending the processing of a luminosity block  ------------
void
DemoAnalyzer::endLuminosityBlock(edm::LuminosityBlock const&, edm::EventSetup const&)
{
}
```

For instance, any instructions placed inside the beginRun routine will be executed every time the Framework sees a new Run (a Run is determined by the start and stop of the acquisition of the CMS detector). One may use the beginJob and endJob routines to, for example, book histograms or write output files.

Let's compile (heads-up: it will fail)

The compilation will invariably fail. This is because the Muon classes we added introduced some dependencies that need to be taken care of in the BuildFile.xml. We will deal with this in a moment. Now, however, it is a good time to submit your assignment for this lesson. Please copy the compilation error message you got and paste it to the corresponding section in our assignment form; remember you must sign in and **click on the submit button** in order to save your work. You can go back to edit the form at any time.

So let's modify the Demo/DemoAnalyzer/BuildFile.xml to include DataFormats/MuonReco dependencies. It should look like:

```
<use name="FWCore/Framework"/>
<use name="FWCore/PluginManager"/>
<use name="DataFormats/MuonReco"/>
<use name="FWCore/ParameterSet"/>
<flags EDM_PLUGIN="1"/>
<export>
   <lib name="1"/>
</export>
```

Now, if you compile again, it should work. Then, we can run with the cmsRun executable:

```
[07:34:26] cmsusr@0e3bbf0570cf ~/CMSSW_5_3_32/src $ scram b
Reading cached build data
****WARNING: No need to export library once you have declared your library as plugin. Please cleanup src/Demo/DemoAnalyzer/BuildFile by removin
g the <export></export> section.
>> Local Products Rules ..... started
>> Local Products Rules ..... done
>> Building CMSSW version CMSSW_5_3_32 ----
>> Entering Package Demo/DemoAnalyzer
>> Creating project symlinks
  src/Demo/DemoAnalyzer/python -> python/Demo/DemoAnalyzer
Entering library rule at Demo/DemoAnalyzer
>> Compiling edm plugin /home/cmsusr/CMSSW_5_3_32/src/Demo/DemoAnalyzer/src/DemoAnalyzer.cc
>> Building edm plugin tmp/slc6_amd64_gcc472/src/Demo/DemoAnalyzer/src/DemoDemoAnalyzer/libDemoDemoAnalyzer.so
Leaving library rule at Demo/DemoAnalyzer
@@@@ Running edmWriteConfigs for DemoDemoAnalyzer
--- Registered EDM Plugin: DemoDemoAnalyzer
>> Leaving Package Demo/DemoAnalyzer
>> Package Demo/DemoAnalyzer built
>> Subsystem Demo built
>> Local Products Rules ..... started
>> Local Products Rules ..... done
gmake[1]: Entering directory `/home/cmsusr/CMSSW_5_3_32'
>> Creating project symlinks
  src/Demo/DemoAnalyzer/python -> python/Demo/DemoAnalyzer
>> Done python_symlink
>> Compiling python modules cfipython/slc6_amd64_gcc472
>> Compiling python modules python
>> Compiling python modules src/Demo/DemoAnalyzer/python
>> All python modules compiled
@@@@ Refreshing Plugins:edmPluginRefresh
>> Pluging of all type refreshed.
>> Done generating edm plugin poisoned information
gmake[1]: Leaving directory `/home/cmsusr/CMSSW_5_3_32'
[07:35:11] cmsusr@0e3bbf0570cf ~/CMSSW_5_3_32/src $
```

cmsRun Demo/DemoAnalyzer/demoanalyzer_cfg.py > mylog.log 2>&1 &

Let's check the log file: cat mylog.log

```
Begin processing the 1st record. Run 195013, Event 24425389, LumiSection 66 at 09-Apr-2022 07:38:16.902 CEST
Muon # 0 with E = 8.47069 GeV.
Begin processing the 2nd record. Run 195013, Event 24546773, LumiSection 66 at 09-Apr-2022 07:38:17.049 CEST
Muon # 0 with E = 36.431 GeV.
Muon # 1 with E = 32.3631 GeV.
Begin processing the 3rd record. Run 195013, Event 24679037, LumiSection 66 at 09-Apr-2022 07:38:17.051 CEST
Muon # 0 with E = 16.5536 GeV.
Muon # 1 with E = 12.5258 GeV.
Begin processing the 4th record. Run 195013, Event 24839453, LumiSection 66 at 09-Apr-2022 07:38:17.052 CEST
Muon # 0 with E = 5.19378 GeV.
Muon # 1 with E = 13.6549 GeV.
Begin processing the 5th record. Run 195013, Event 24894477, LumiSection 66 at 09-Apr-2022 07:38:17.053 CEST
Muon # 0 with E = 10.3546 GeV.
Begin processing the 6th record. Run 195013, Event 24980717, LumiSection 66 at 09-Apr-2022 07:38:17.054 CEST
Muon # 0 with E = 25.2733 GeV.
Muon # 1 with E = 10.7545 GeV.
Begin processing the 7th record. Run 195013, Event 25112869, LumiSection 66 at 09-Apr-2022 07:38:17.055 CEST
Muon # 0 with E = 165.026 GeV.
Muon # 1 with E = 145.994 GeV.
Begin processing the 8th record. Run 195013, Event 25484261, LumiSection 66 at 09-Apr-2022 07:38:17.056 CEST
Muon # 0 with E = 130.171 GeV.
Muon # 1 with E = 73.1873 GeV.
Begin processing the 9th record. Run 195013, Event 25702821, LumiSection 66 at 09-Apr-2022 07:38:17.057 CEST
Muon # 0 with E = 24.0889 GeV.
Muon # 1 with E = 14.4771 GeV.
Begin processing the 10th record. Run 195013, Event 25961949, LumiSection 66 at 09-Apr-2022 07:38:17.058 CEST
Muon # 0 with E = 189.134 GeV.
Muon # 1 with E = 38.8268 GeV.
09-Apr-2022 07:38:17 CEST  Closed file root://eospublic.cern.ch//eos/opendata/cms/Run2012B/DoubleMuParked/AOD/22Jan2013-v1/10000/1EC938EF-ABEC-E211-94E
0-90E6BA442F24.root

%MSG-----------------------------------------

MessageLogger Summary

 type     category        sev    module        subroutine        count    total
 ---- -------------------- -- ---------------- ----------------  -----    -----
    1 fileAction           -s file_close                           1        1
    2 fileAction           -s file_open                            2        2

 type     category    Examples: run/evt        run/evt          run/evt
 ---- -------------------- ---------------- ---------------- ----------------
    1 fileAction           PostEndRun
    2 fileAction           pre-events       pre-events

Severity    # Occurrences   Total Occurrences
--------    -------------   -----------------
System                 3                  3
[5]-  Done                     cmsRun Demo/DemoAnalyzer/demoanalyzer_cfg.py > mylog.log 2>&1
[07:39:47] cmsusr@0e3bbf0570cf ~/CMSSW_5_3_32/src $
```

- The C++ source file of an EDAnalyzer is taylored for particle physics analysis under the CMSSW Framework.
- This source file needs to be modified according to the analyzer needs

THE CONFIGURATION

CMSSW configuration Framework

The CMS software framework uses a “software bus” model. A single executable, cmsRun, is used, and the modules are loaded at runtime. A configuration file, fully written in Python, defines which modules are loaded, in which order they are run, and with which configurable parameters they are run. Note that this is not an interactive system. The entire configuration is defined once, at the beginning of the job, and cannot be changed during running. This is the file that you “feed” cmRun when it is executed.

Playing with the demoanalyzer_cfg.py file

In the case of our DemoAnalyzer we have been working with, its configuration file is the demoanalyzer_cfg.py, which resides in the Demo/DemoAnalyzer/ directory of your CMSSW Demo package. Note that it does not reside in the Demo/DemoAnalyzer/python directory (which is usually the case for python configuration files). This does not really matter though. There is a specific situation where this is important, and we will look at it later.

Meanwhile, if we explore what is in the Demo/DemoAnalyzer/ python directory:

```
➜  ~ docker start -i my_od
Setting up CMSSW_5_3_32
[17:10:44] cmsusr@0e3bbf0570cf ~/CMSSW_5_3_32/src $ cmsenv
[17:11:20] cmsusr@0e3bbf0570cf ~/CMSSW_5_3_32/src $ ls Demo/DemoAnalyzer/python
__init__.py  __init__.pyc  demoanalyzer_cfi.py  demoanalyzer_cfi.pyc
```

You will note that there is a demoanalyzer_cfi.py in there. We will not pay attention to this file now, but it instructive to point out that the _cfg and _cfi descriptors are meaningful. While the former one defines a top level configuration, the latter works more like a *module initialization* file. There are also _cff files which bear pieces of configuration and so they are dubbed *config fragments*. You can read a bit more about it in this subsection of the Workbook.

Ok, so the file we will play around with is just the Demo/DemoAnalyzer/demoanalyzer_cfg.py. Let’s take a look with the nano editor:

The first line imports our CMS-specific Python classes and functions, and the second one creates a **process object**. This refers to a CMSSW process (the one we will be configuring, of course). Essentially, the main idea is that we will be *feeding* our process all the tasks that we need done by the CMSSW software or its plugins. The process needs always a name. It could be any short word, but it is usually chosen so it is meaningful. For instance, if the main task would be to process the high level trigger information, an adequate name will be “HLT”; if the process is actually the

full reconstruction of the data, then it is most likely assigned the name “RECO”. For our demo, we will leave our creativity aside and just call it “Demo” (you can, of course, change it to your linking).

```
import FWCore.ParameterSet.Config as cms

process = cms.Process("Demo")

process.load("FWCore.MessageService.MessageLogger_cfi")

process.maxEvents = cms.untracked.PSet( input = cms.untracked.int32(10) )
```

Actually, because of the _cfi tag, we know it is presumably a piece of Python code that initializes some module. Indeed, it is the *MessageLogger* service. As the name describes, it controls how the message logging is handled during the job execution. The string "FWCore.MessageService.MessageLogger_cfi" tells you exactly where to look for it on Githubif you needed it. Note the structure matches the repository’s, except that the python directory name is always omitted when loading modules this way (this is why it is often important to put config files in the python directory).

Changing the logging frequency in our CMSSW job

Suppose you want the Framework to report every 5 events instead of each event. Then one can simply add this line *process.MessageLogger.cerr.FwkReport.reportEvery = 5* right below the *load* line:

it is easy to guess that it controls the number of events that are going to be processed in our CMSSW job. It is worth noting that maxEvents is a *untracked* variable within the Framework. In general, the system keeps track of what parameters are used to create each data item in the Event and saves this information in the output files. This can be used later to help understand how the data was made. However, sometimes a parameter will have no effect on the final objects created. Such parameters are declared *untracked*. More information can be found in the Workbook.

Let’s change the number of events to 100:

Next, there is the first module (also an object by itself) we are attaching to our process object:

Inside the process object there must be exactly one object assigned that has Python type Source and is used for data input. There may be zero or more objects for each of many other Python types. In the official production configurations there can be hundreds or even thousands of objects attached to the process. Your job is configured by your choice of objects to construct and attach to the process, and by the configuration of each object. (This may be done via import statements or calls to the load function, instead of or in addition to object construction.) Some of the Python types that may be used to create these objects are listed in the Workbook.

Explore the PoolSource C++

Generally, all modules in our python configuration are associated with its corresponding C++ code. By searching the CMSSW Github repository, will you be able to point exactly to the line of C++ code where the fileNames parameter is read?

Solution:

You will find that the label first appearing in a Python CMSSW module is actually the name of the C++ code. So, we would expect that there be a class PoolSource.h (and perhaps its implementation as PoolSource.C) associated with the "PoolSource" label. Let's then go to the Github CMSSW repository and simply search for PoolSource.Cusing the search field of that page. Immediately, in the search results, we notice there is a IOPool/Input/src/PoolSource.cc file that we can browse. After looking for the variable fileNames, we find that this parameter is read in this line. Note that it is in the constructor of the object where the ParameterSet objects are read.

Note also that the fileNames variable is a vstring, i.e., a vector of strings in the C++ sense. In Python, it is a list, so you can very well input a comma separated list of files. There is a drawback, though. In general, our open datasets will contain more than 255 files, which is the limit for the number of arguments a Python function can take, so very long vstrings cannot be created in one step. There are various alternatives to circumvent this problem. To run over massive amounts of ROOT files, one will usually use the FileUtils module to load index files instead of individual ROOT files.

Configure our DemoAnalyzer

The second to last line in our configuration has to do with our recently created DemoAnalyzer. This module is now just a *declaration of existance* because it is empty. Let's put it to work:

process.demo = cms.EDAnalyzer('DemoAnalyzer')

Making our EDAnalyzer configurable

Let's pretend that for some reason you will need to run your job either for extracting the energy of RECO muons from beam collisions or from cosmic rays. Note that CMS has information from both. If you go back to the guide we used earlier to help us with the muon information extraction, you will notice that there is the possibility to use an InputTagthat is muonsFromCosmics instead of just muons. Your job is to make this configurable in our demoanalyzer_cfg.py so we don't have to re-compile every time we want to make the switch.

Since we are going to make our DemoAnalyzer configurable, the first thing we need to do is to modify the C++ source of our analyzer in order to accommodate configurability. Let's modify then the Demo/DemoAnalyzer/src/DemoAnalyzer.cc file. Again, following the logic in the Physics Ojects guide and using an editor, we should add the

declaration for a muon InputTag. We could include this declaration right below the declaration of our member functions:

```
      virtual void endLuminosityBlock(edm::LuminosityBlock const&, edm::EventSetup const&);
      //declare the input tag for MuonCollection
      edm::InputTag muonInput;

      // ----------member data ---------------------------
      std::vector<float> muon_e; //energy values for muons in the event
};
```

Then we will have to read this InputTag from the configuration. As it was noted above, this is done in the constructor. It will become:

```
//
// constructors and destructor
//
DemoAnalyzer::DemoAnalyzer(const edm::ParameterSet& iConfig)

{
   //now do what ever initialization is needed
   muonInput = iConfig.getParameter<edm::InputTag>("InputCollection");
}
```

Here we will be reading the InputCollection variable from configuration (which is of type edm::InputTag, which is essentially a string) and will store it in the muonInputcontainer.

Next, we will modify the analyze function replacing this line

iEvent.getByLabel("muons", mymuons);,

where the InputTag is hard-coded as “muons”, with

iEvent.getByLabel(muonInput, mymuons);,

where we use the configurable muonInput variable.

The section of interest in the analyze function will then look like:

```
void
DemoAnalyzer::analyze(const edm::Event& iEvent, const edm::EventSetup& iSetup)
{
   using namespace edm;
   //clean the container
   muon_e.clear();

   //define the handler and get by label
   Handle<reco::MuonCollection> mymuons;
   iEvent.getByLabel(muonInput, mymuons);

   //if collection is valid, loop over muons in event
   if(mymuons.isValid()){
      for (reco::MuonCollection::const_iterator itmuon=mymuons->begin(); itmuon!=mymuons->end(); ++itmuon){
          muon_e.push_back(itmuon->energy());
      }
   }

   //print the vector
```

Finally, let’s change the Demo/DemoAnalyzer/demoanalyzer_cfg.py by replacing our empty module statement:

process.demo = cms.EDAnalyzer('DemoAnalyzer')

with

```
import FWCore.ParameterSet.Config as cms

process = cms.Process("Demo")

process.load("FWCore.MessageService.MessageLogger_cfi")
process.MessageLogger.cerr.FwkReport.reportEvery = 5
process.maxEvents = cms.untracked.PSet( input = cms.untracked.int32(100) )

process.source = cms.Source("PoolSource",
    # replace 'myfile.root' with the source file you want to use
    fileNames = cms.untracked.vstring(
        'root://eospublic.cern.ch//eos/opendata/cms/Run2012B/DoubleMuParked/AOD/22Jan2013-v1/10000/1EC938EF-ABEC-E211-94E0-90E6BA442F2$
    )
)

process.demo = cms.EDAnalyzer('DemoAnalyzer',
       InputCollection = cms.InputTag("muons")
)
process.p = cms.Path(process.demo)
```

In this way, we are now able to enter “muons” or “muonsFromCosmics”, depending on our needs.

Now, before re-compiling our code, let’s check that our python configuration is ok. We can validate the syntax of your configuration using python:

python Demo/DemoAnalyzer/demoanalyzer_cfg.py

Now let’s compile the code; again, with scram:

```
[17:53:21] cmsusr@e3bbf0570cf ~/CMSSW_5_3_32/src $ python Demo/DemoAnalyzer/demoanalyzer_cfg.py
[17:53:43] cmsusr@e3bbf0570cf ~/CMSSW_5_3_32/src $ scram b
>> Local Products Rules ..... started
>> Local Products Rules ..... done
>> Building CMSSW version CMSSW_5_3_32 ----
>> Entering Package Demo/DemoAnalyzer
>> Creating project symlinks
  src/Demo/DemoAnalyzer/python -> python/Demo/DemoAnalyzer
>> Compiling edm plugin /home/cmsusr/CMSSW_5_3_32/src/Demo/DemoAnalyzer/src/DemoAnalyzer.cc
>> Building edm plugin tmp/slc6_amd64_gcc472/src/Demo/DemoAnalyzer/src/DemoDemoAnalyzer/libDemoDemoAnalyzer.so
Leaving library rule at Demo/DemoAnalyzer
@@@@ Running edmWriteConfigs for DemoDemoAnalyzer
--- Registered EDM Plugin: DemoDemoAnalyzer
>> Leaving Package Demo/DemoAnalyzer
>> Package Demo/DemoAnalyzer built
>> Subsystem Demo built
>> Local Products Rules ..... started
>> Local Products Rules ..... done
gmake[1]: Entering directory `/home/cmsusr/CMSSW_5_3_32'
>> Creating project symlinks
  src/Demo/DemoAnalyzer/python -> python/Demo/DemoAnalyzer
>> Done python_symlink
>> Compiling python modules cfipython/slc6_amd64_gcc472
>> Compiling python modules python
>> Compiling python modules src/Demo/DemoAnalyzer/python
>> All python modules compiled
@@@@ Refreshing Plugins:edmPluginRefresh
>> Pluging of all type refreshed.
>> Done generating edm plugin poisoned information
gmake[1]: Leaving directory `/home/cmsusr/CMSSW_5_3_32'
[17:54:59] cmsusr@e3bbf0570cf ~/CMSSW_5_3_32/src $
```

Finally, let’s run the CMSSW job:

cmsRun Demo/DemoAnalyzer/demoanalyzer_cfg.py > mylog.log 2>&1 &

If you check the development of the job with

tail -f mylog.log

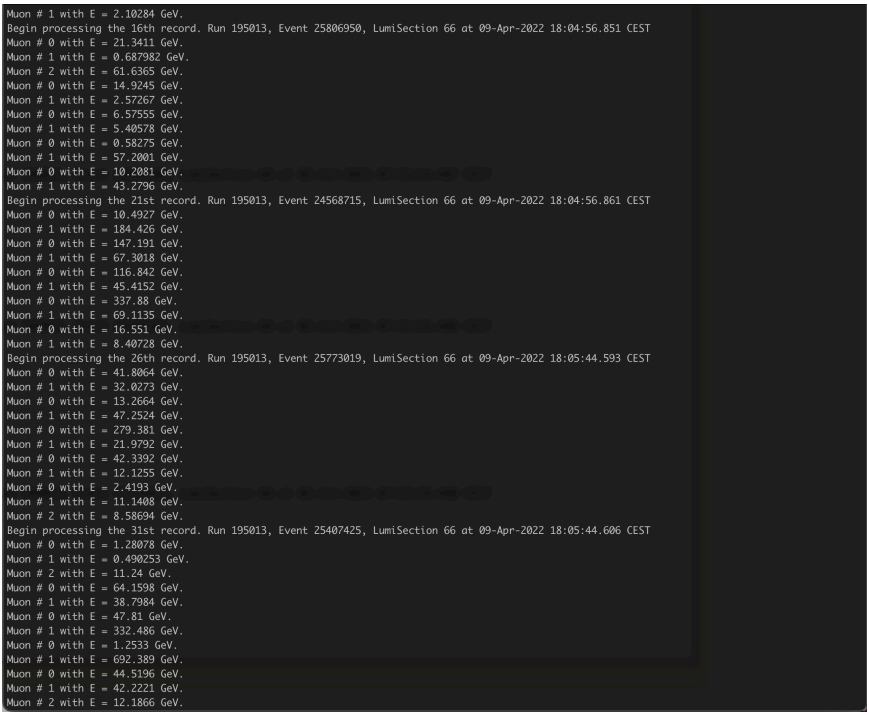
Eventually, as the job progresses, you will see something like:

```
Muon # 0 with E = 22.3278 GeV.
Muon # 1 with E = 36.3877 GeV.
Begin processing the 16th record. Run 195013, Event 25806950, LumiSection 66 at 09-Apr-2022 17:57:06.992 CEST
Muon # 0 with E = 9.28834 GeV.
Muon # 1 with E = 41.184 GeV.
Muon # 2 with E = 60.024 GeV.
Muon # 0 with E = 101.177 GeV.
Muon # 1 with E = 11.2308 GeV.
Muon # 0 with E = 35.0474 GeV.
Muon # 1 with E = 31.4979 GeV.
Muon # 0 with E = 56.0866 GeV.
Muon # 1 with E = 78.7078 GeV.
Muon # 0 with E = 42.6871 GeV.
Muon # 1 with E = 49.626 GeV.
Begin processing the 21st record. Run 195013, Event 24568715, LumiSection 66 at 09-Apr-2022 17:57:07.001 CEST
Muon # 0 with E = 80.9343 GeV.
Muon # 1 with E = 57.3367 GeV.
Muon # 0 with E = 64.6396 GeV.
Muon # 1 with E = 784.396 GeV.
Muon # 0 with E = 174.626 GeV.
Muon # 1 with E = 52.107 GeV.
Muon # 0 with E = 254.615 GeV.
Muon # 1 with E = 115.981 GeV.
Muon # 0 with E = 17.6325 GeV.
Muon # 1 with E = 43.1466 GeV.
Begin processing the 26th record. Run 195013, Event 25773019, LumiSection 66 at 09-Apr-2022 17:57:50.090 CEST
Muon # 0 with E = 896.068 GeV.
Muon # 1 with E = 30.0968 GeV.
Muon # 0 with E = 31.7867 GeV.
Muon # 1 with E = 45.007 GeV.
Muon # 0 with E = 144.005 GeV.
Muon # 1 with E = 32.4624 GeV.
Muon # 0 with E = 109.775 GeV.
Muon # 1 with E = 29.5543 GeV.
Muon # 0 with E = 3.44853 GeV.
Begin processing the 31st record. Run 195013, Event 25407425, LumiSection 66 at 09-Apr-2022 17:57:50.104 CEST
Muon # 0 with E = 38.4875 GeV.
Muon # 1 with E = 37.2395 GeV.
Muon # 0 with E = 58.2166 GeV.
Muon # 1 with E = 64.7743 GeV.
Muon # 0 with E = 73.7936 GeV.
Muon # 1 with E = 48.7898 GeV.
Muon # 0 with E = 9.98582 GeV.
Muon # 1 with E = 20.2956 GeV.
Muon # 2 with E = 34.8767 GeV.
Muon # 0 with E = 79.7684 GeV.
Muon # 1 with E = 49.3363 GeV.
Begin processing the 36th record. Run 195013, Event 24586506, LumiSection 66 at 09-Apr-2022 17:57:50.118 CEST
Muon # 0 with E = 11.7313 GeV.
```

Change the InputTag

Now, change the name of the InputColletion from “muons” to “muonsFromCosmics” in your configuration and run again **without** re-compiling the code. Do you see any difference?

```
 GNU nano 2.0.9                File: Demo/DemoAnalyzer/demoanalyzer_cfg.py                Modified

import FWCore.ParameterSet.Config as cms

process = cms.Process("Demo")

process.load("FWCore.MessageService.MessageLogger_cfi")
process.MessageLogger.cerr.FwkReport.reportEvery = 5
process.maxEvents = cms.untracked.PSet( input = cms.untracked.int32(100) )

process.source = cms.Source("PoolSource",
    # replace 'myfile.root' with the source file you want to use
    fileNames = cms.untracked.vstring(
        'root://eospublic.cern.ch//eos/opendata/cms/Run2012B/DoubleMuParked/AOD/22Jan2013-v1/10000/1EC938EF-ABEC-E211-94E0-90E6BA442F24.root'
    )
)

process.demo = cms.EDAnalyzer('DemoAnalyzer',
      InputCollection = cms.InputTag("muonsFromCosmics")
)
process.p = cms.Path(process.demo)
```



Running some already-available CMSSW code

The last line in our Demo/DemoAnalyzer/demoanalyzer_cfg.py is

process.p = cms.Path(process.demo)

The “software bus” model that was mentioned in the introduction of this episode can be made evident in this line. CMSSW executes its code using *Paths* (which in turn could be arranged in Schedules). Each Path can execute a series of modules (or Sequences of modules). In our example we have just one Path named p that executes the demo process, which corresponds to our DemoAnalyzer.

In general, however, we could add more modules. For instance, the Path line could look like

process.mypath = cms.Path(process.m1+process.m2+process.s1+process.m3)

, where m1, m2, m3 could be CMSSW modules (individual EDAnalyzers, EDFilters, EDProducers, etc.) and s1 could be a modules Sequence.

Adding a Trigger Filter

In CMSSW, there are other types of code one can execute. Some of these are known as EDFilters. As the name implies, they can be used to filter events. For instance, one could use the HLTHighLevel filter class to only run over events that have passed a certain kind of trigger.

We can get a hint of its usage by scooping around the corresponding python directory of that package. We immediately notice the hltHighLevel_cfi.py module, so let's load it with the line

process.load("HLTrigger.HLTfilters.hltHighLevel_cfi")

Let's configure the HLTPaths parameter in that module so it will only pass events that fired any trigger bit with the pattern HLT_Mu15*:

process.hltHighLevel.HLTPaths = cms.vstring('HLT_Mu15*')

Now, let's add the module to our running Path:

process.p = cms.Path(process.hltHighLevel+process.demo)

```
 GNU nano 2.0.9                    File: Demo/DemoAnalyzer/demoanalyzer_cfg.py

import FWCore.ParameterSet.Config as cms

process = cms.Process("Demo")

process.load("FWCore.MessageService.MessageLogger_cfi")
process.MessageLogger.cerr.FwkReport.reportEvery = 5

process.maxEvents = cms.untracked.PSet( input = cms.untracked.int32(100) )

process.source = cms.Source("PoolSource",
    # replace 'myfile.root' with the source file you want to use
    fileNames = cms.untracked.vstring(
        'root://eospublic.cern.ch//eos/opendata/cms/Run2012B/DoubleMuParked/AOD/22Jan2013-v1/10000/1EC938EF-ABEC-E211-94E0-90E6BA442F24.root'
    )
)

process.demo = cms.EDAnalyzer('DemoAnalyzer',
      InputCollection = cms.InputTag("muons")
)

process.load("HLTrigger.HLTfilters.hltHighLevel_cfi")
process.hltHighLevel.HLTPaths = cms.vstring('HLT_Mu7*')

process.p = cms.Path(process.hltHighLevel+process.demo)
```

Without even having to compile again, the execution of the trigger path will stop if the hltHighLevel filter module throws a False result. The output becomes

```
Begin processing the 1st record. Run 195013, Event 24425389, LumiSection 66 at 09-Apr-2022 18:16:06.598 CEST
Begin processing the 6th record. Run 195013, Event 24980717, LumiSection 66 at 09-Apr-2022 18:16:06.646 CEST
Begin processing the 11th record. Run 195013, Event 24530992, LumiSection 66 at 09-Apr-2022 18:16:06.652 CEST
Begin processing the 16th record. Run 195013, Event 25806950, LumiSection 66 at 09-Apr-2022 18:16:06.656 CEST
Begin processing the 21st record. Run 195013, Event 24568715, LumiSection 66 at 09-Apr-2022 18:16:06.665 CEST
Begin processing the 26th record. Run 195013, Event 25773019, LumiSection 66 at 09-Apr-2022 18:16:06.670 CEST
Begin processing the 31st record. Run 195013, Event 25407425, LumiSection 66 at 09-Apr-2022 18:16:06.674 CEST
Begin processing the 36th record. Run 195013, Event 24586506, LumiSection 66 at 09-Apr-2022 18:16:19.123 CEST
Begin processing the 41st record. Run 195013, Event 24369161, LumiSection 66 at 09-Apr-2022 18:16:19.141 CEST
Begin processing the 46th record. Run 195013, Event 25119433, LumiSection 66 at 09-Apr-2022 18:16:19.152 CEST
Begin processing the 51st record. Run 195013, Event 25759016, LumiSection 66 at 09-Apr-2022 18:16:19.160 CEST
Begin processing the 56th record. Run 195013, Event 25178292, LumiSection 66 at 09-Apr-2022 18:16:19.168 CEST
Begin processing the 61st record. Run 195013, Event 25118983, LumiSection 66 at 09-Apr-2022 18:16:19.175 CEST
Begin processing the 66th record. Run 195013, Event 24534099, LumiSection 66 at 09-Apr-2022 18:16:19.181 CEST
Begin processing the 71st record. Run 195013, Event 25156815, LumiSection 66 at 09-Apr-2022 18:16:19.187 CEST
Begin processing the 76th record. Run 195013, Event 27158713, LumiSection 67 at 09-Apr-2022 18:16:19.195 CEST
Begin processing the 81st record. Run 195013, Event 26563495, LumiSection 67 at 09-Apr-2022 18:16:19.201 CEST
Begin processing the 86th record. Run 195013, Event 27467127, LumiSection 67 at 09-Apr-2022 18:16:19.205 CEST
Begin processing the 91st record. Run 195013, Event 26433779, LumiSection 67 at 09-Apr-2022 18:16:19.210 CEST
Begin processing the 96th record. Run 195013, Event 27692659, LumiSection 67 at 09-Apr-2022 18:16:19.214 CEST
09-Apr-2022 18:16:19 CEST  Closed file root://eospublic.cern.ch//eos/opendata/cms/Run2012B/DoubleMuParked/AOD/22Jan2013-v1/10000/1EC938EF-ABEC-E2
11-94E0-90E6BA442F24.root

====================================

MessageLogger Summary

 type     category        sev    module        subroutine        count    total
 ---- -------------------- -- ---------------- ----------------  -----    -----
    1 fileAction           -s file_close                           1        1
    2 fileAction           -s file_open                            2        2

 type    category    Examples: run/evt        run/evt          run/evt
 ---- -------------------- ---------------- ---------------- ----------------
    1 fileAction           PostEndRun
    2 fileAction           pre-events       pre-events

Severity    # Occurrences   Total Occurrences
--------    -------------   -----------------
System                 3                   3
```

It happens that for these few events we are testing, the filter clearly prevents the execution of our EDAnalyzer.

Congratulations!!, you have made it to the end the lesson.

- The Python implementation of the configuration of CMSSW is fundamentally modular.
- The CMSSW configuration allows for the usage of already available code and/or make changes to yours without having to recompile.

CMS TRIGGER

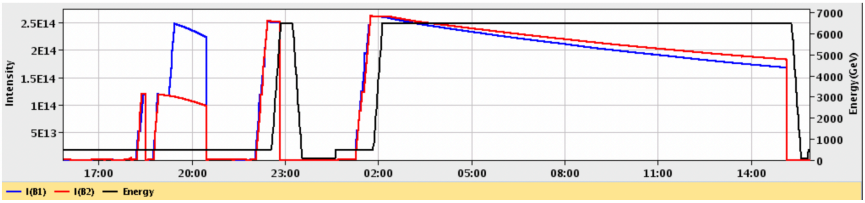
The CMS acquisition and trigger systems

Collisions at the LHC happen at a **rate** close to 40 million per second (40 MHz). Once each collision is *sensed* by the different subdetectors, the amount of information they generate for each one of them corresponds to about what you can fit in a 1 MB file. If we were to record every single collision, it is said (you can do the math) that one can probably fill out all the available disk space in the world in a few days!

Fortunately, as you know, not all collisions that happen at the LHC are interesting, so we do not have to record every single one of them. We want to keep the interesting ones and, most importantly, do not miss the **discovery-quality** ones. In order to achieve that we need a *Trigger*.

Before we jump into the details for the trigger system, let's agree on some terminology:

- **Fill**: Every time the LHC injects beams in the machine it marks the beginning of what is known as a *Fill*. When they dump the beams, it marks the end of that Fill. Each Fill is given a unique number. Some of these fills are declared *stable* and *good for physics* and that is when we, the detectors, collect the data good for doing quality research.
- **Run**: As collisions happen in the LHC, CMS (and the other detectors) decide whether they start *recording* data. Every time the *start* button is pushed, a new *Run* starts and it is given a unique number. The Run can stop for a variety of reasons, like if the LHC dumps the fill, but usually it is because there is some glitch in one the thousands of electronics channels in the detector that causes the acquisition to stop. This is why the *recorded* (by the detectors) and *delivered* data (by the LHC) is not usually the same. When the acquisition is restarted, a new run number is assigned.



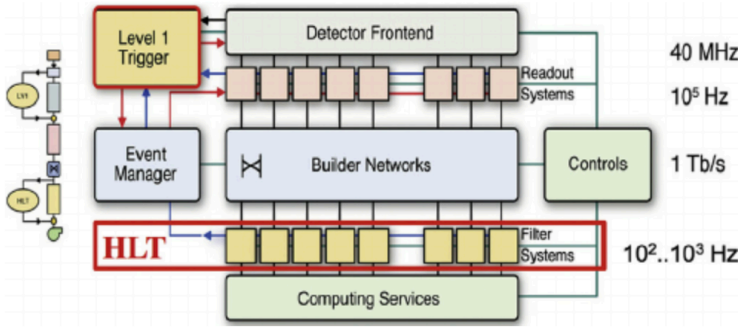
- **Lumi section**: while colliding, the LHC's delivered instantaneous luminosity gets degraded (in general, it decreases) due to different reasons. I.e., it is not constant over time. For practical reasons, CMS groups the events it collects in *luminosity sections*, where the luminosity values can be considered constant.

In the plot you can see the different beam (B) injections. For this particular case, around 2 am that day, collisions were declared stable and lasted until around 15H. The luminosity (which is dependent on the intensity of the proton bunches)starts decreasing as time goes by for this example fill.

The trigger system

Deciding on which events to record is the main purpose of the trigger system. It is like deciding which events to record by taking a **quick picture** of it and, even though a bit blurry, decide whether it is interesting to keep or not for a future, more thorough inspection.

CMS does this in two main steps. The first one, the **Level 1 trigger** (L1), implemented in hardware (fast FPGAs), reduces the input rate of 40 Mhz to around 100 KHz. The other step is the **High Level Trigger** (HLT), run on commercial machines with good-old C++ and Python, where the input rate is leveled around the maximum available budget of around 1-2 KHz.



There are hundreds of different triggers in CMS. Each one of them is designed to pick certain types of events, with different intensities and topologies. For instance the HLT_Mu15trigger, will select events with at least one muon with *15 GeV* of transverse momentum. But how can one know this? We will try to answer that later.

For now, let's just mention that these triggers are **implemented using CMSSW code** (remember, the CMS experiment uses CMSSW to acquire,

produce, process and even analyze its data), using pieces of it (modules) that can be arranged to achieve the desired result: selecting specific kinds of events. Therefore, computationally, triggers are just *Paths* in the CMSSW sense, and one could extract a lot of information by exploring the Python configuration of these paths.

An example of such an arrangement for an HLT trigger looks like:

process.HLT_Mu15_v2 = cms.Path(process.HLTBeginSequenceBPTX + process.hltL1sL1SingleMu10 + process.hltPreMu15 + process.hltL1SingleMu10L1Filtered0 + process.hltL2Mu10L2Filtered10 + process.HLTL2muonrecoSequence + process.hltL3Muon15 + process.HLTL3muonrecoSequence + process.HLTEndSequence)

process.mypath = cms.Path (process.m1+process.m2+process.s1+process.m3)

Where, m1, m2, m3 could be CMSSW modules (individual EDAnalyzers, EDFilters, EDProducers, etc.) and s1 could be a Sequence of modules.

Finally, triggers are code, and those pieces of code are constantly changing. Modifications to a trigger could imply a different version identifier. For instance, our HLT_Mu15 could actually be HLT_Mu15_v2 or HLT_Mu15_v3, etc., depending on the version. Therefore, it is completely normal that the trigger names can **change from run to run**.

Prescales

The need for prescales (and its meaning) is evident if one thinks of different physics processes having different cross sections. Roughly speaking, we are a lot more likely to record one minimum bias event, than an event where a Z boson is produced. Even less likely is to record an event with a Higgs boson. We could have triggers named, say, HLT_ZBosonRecorder for the one in charge of filtering Z-carrying events, or HLT_HiggsBosonRecorder for the one accepting Higgses (the actual names are more sophisticated and complex than that, of course.) The prescales are designed to keep these inputs under control by, for instance, recording just 1 out of 100 collisions that produce a likely Z boson, or 1 out of 1 collisions that produce a potential Higgs boson. In the former case, the prescale would be 100, while for the latter it would be 1; if a trigger has a prescale of 1, i.e., records every single event it identifies, we call it **unprescaled**.

Maybe not so evident is the need for trigger prescale changes for keeping up with luminosity changes. As the luminosity drops, prescales can be relaxed, and therefore **can change from to run** in the same fill.

A trigger can be prescaled at L1 as well as the HLT levels. L1 triggers have their own nomenclature and can be used as HLT trigger *seeds*.

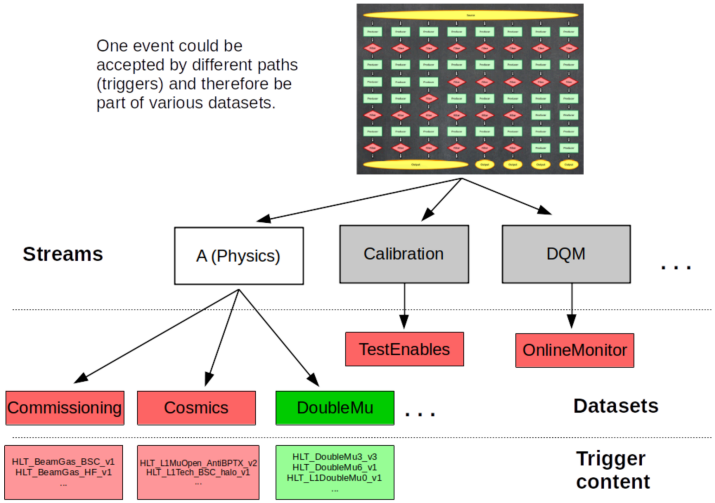
Triggers, streams and datasets

After events are accepted by possibly more than one type of trigger, they are streamed in different categories, called **streams** and then classified and arranged in primary **datasets**. Most, but not all, of the

datasets belonging to the stream A, the physics stream, are or will become available as CMS Open Data. There are many more datasets and streams than depicted in the figure below. Datasets in read are examples of those we do not plan to release as open data, while the one in green is an example of those that we do.

Finally, it is worth mentioning that:

- an event can be triggered by many trigger paths
- trigger paths are unique to each dataset
- **the same event can arrive in two different datasets** (this is speciall important if working with many datasets as event duplication can happen)



- The CMS trigger system filters uninteresting events, keeping the budget high for the flow of interesting data.
- Computationally, a trigger is a CMSSW path, which is composed of several software modules.
- Trigger prescales allow the data acquisition to adjust to changes in instantaneous luminosity while keeping the rate of incomming data under control
- The trigger systems allows for the classification and organization of datasets by physics objects of interest

CHOOSING A TRIGGER

Exploring the triggers in your dataset

After choosing the appropriate dataset for you analysis, the first thing you need to decide is which trigger to use.

If you click on the 2012 TauPlusX dataset record, you will find a list triggers that were streamed to that dataset. Now, the question is, which one would you use for this analysis? Here's where the physics starts

playing a role. Let's imagine we concentrate only on tau lepton pairs of which one tau lepton decays into a muon and two neutrinos and the other tau lepton hadronically, then we shall pick our trigger(s) accordingly. Note that the published CMS analysis considers additional decay channels.

Let's try to click on one of those triggers, for instance HLT_IsoMu8_eta2p1_LooseIsoPFTau20; there, you can find additional information about this trigger. One thing to notice, if you check the 2012 run range, is that this trigger was only available towards the end of the data taking period. As a matter of fact, since we have only released half of the 2012 data, you will not find that trigger yet in the current releases.

So we have learned that triggers could be not persistent and be only available for certain runs. Besides, as it was mentioned, the trigger code evolves. There may be different versions of this trigger, for instance, HLT_IsoMu8_eta2p1_LooseIsoPFTau20_v1, or v2, v3, etc.

In order to check the trigger information we need to use the trigger tools provided in CMSSW. This record points you to some examples of the usage of such code. Let's use, for instance, some of the snippets presented in the source file of the GeneralInfoAnalyzer package to dump all the triggers in our dataset.

First, make sure you go to your CMSSW_5_3_32/src area and issue a **cmsenv** command, as you will always do.

To simplify the exercise, we will use the same source file we used for validation, the Demo/DemoAnalyzer/src/DemoAnalyzer.cc. Let's modify it so we can use it to dump all the triggers in our dataset. In the process, **let's comment out some of what we did for extracting the muon energy** so it does not clutter our output.

Insert the header for the HLTConfigProvider class:

```
//classes to extract Muon information
#include "DataFormats/MuonReco/interface/Muon.h"
#include "DataFormats/MuonReco/interface/MuonFwd.h"
//for trigger configuration
#include "HLTrigger/HLTcore/interface/HLTConfigProvider.h"
//the standard c++ vector class
#include<vector>
```

Next, insert the declaration of the variables we will need in our configuration file (process name, dataset name) and an object of class HLTConfigProvider, which we can use to extract the information about what the trigger configuration was for some run:

```
    virtual void beginLuminosityBlock(edm::LuminosityBlock const&, edm::EventSetup const&);
    virtual void endLuminosityBlock(edm::LuminosityBlock const&, edm::EventSetup const&);
    //declare the input tag for MuonCollection
    //edm::InputTag muonInput;

    // ----------member data ---------------------------
    //std::vector<float> muon_e; //energy values for muons in the event
    //for trigger config
    std::string   processName_;
    std::string   datasetName_;
    //HLT config provider object
    HLTConfigProvider hltConfig_;


};
```

Add the lines to read the configuration in the constructor and print it out (note the way it is done differs a bit from what we did earlier for the muons. They are, of course, equivalent):

```
{
  //now do what ever initialization is needed
  //muonInput = iConfig.getParameter<edm::InputTag>("InputCollection");
  using namespace std;
  using namespace edm;

  //Print the configuration just to check
  cout << "Here is the information passed to the constructor:" <<endl;
  cout << "Configuration: " << endl
       << "   ProcessName = " << processName_ << endl
       << "   DataSetName = " << datasetName_ << endl;
}
```

Don't forget to comment out all the muon stuff in the analyze method so it does not bother us:

```
{
  using namespace edm;
  //clean the container
  //muon_e.clear();

  //define the handler and get by label
  //Handle<reco::MuonCollection> mymuons;
  //iEvent.getByLabel(muonInput, mymuons);

  //if collection is valid, loop over muons in event
  //if(mymuons.isValid()){
  //   for (reco::MuonCollection::const_iterator itmuon=mymuons->begin(); itmuon!=mymuons->end(); +$
  //       muon_e.push_back(itmuon->energy());
  //   }
  //}

  //print the vector
  //for(unsigned int i=0; i < muon_e.size(); i++){
  //   std::cout <<"Muon # "<<i<<" with E = "<<muon_e.at(i)<<" GeV."<<std::endl;
  //}
```

And, finally, modify the beginRun fuction giving a name to iRun and iSetup arguments and adding the trigger dump. Remember we have to check the triggers available at each change in runs:

```
// ------------ method called when starting to processes a run  ------------
void
DemoAnalyzer::beginRun(edm::Run const&, edm::EventSetup const&)
{
  using namespace std;
  using namespace edm;

  //If the hltConfig can be initialized, then the below is an example of
  //how to extract the config information for the trigger from the
  //so-called provenance.

  // The trigger configuration can change from
  // run to run (during the run is the same),
  // so it needs to be called here.

  ///   "init" return value indicates whether intitialisation has succeeded
  ///   "changed" parameter indicates whether the config has actually changed

  bool changed(true);
  if (hltConfig_.init(iRun,iSetup,processName_,changed)) {
      if (changed) {
        const vector<string> triggerNamesInDS = hltConfig_.datasetContent(datasetName_);
         for (unsigned i = 0; i < triggerNamesInDS.size(); i++) {
            cout<<triggerNamesInDS[i]<<endl;
          }
            }
  }
}
```

Before compiling, change your Demo/DemoAnalyzer/BuildFile.xml to include the HLTrigger/HLTcore package, where the HLTConfigProvider resides:

```
 GNU nano 2.0.9              File: Demo/DemoAnalyzer/BuildFile.xml                    Modified

<use name="FWCore/Framework"/>
<use name="FWCore/PluginManager"/>
<use name="DataFormats/MuonReco"/>
<use name="FWCore/ParameterSet"/>
<use name="HLTrigger/HLTcore"/>
<flags EDM_PLUGIN="1"/>
<export>
   <lib name="1"/>
</export>
```

Compile with scram b.

Now, let's modify the configuration file Demo/DemoAnalyzer/demoanalyzer_cfg.py to adapt it to our exercise. First, let's go back to logging for each event (and not for every 5) and change the number of events to -1, so we can run over all of them. Also, change the PoolSource file; replace it with a couple of files from our dataset selection. In addition, comment out what we had done for extracting the muon information and adding the HLTHighLevel filter, and replace it with parameters we need at configuration. Do not forget to notice that we are naming our process mytrigger now, and not demo.

The connect variable in one of those lines just modifies they way in which the framework is going to access these snapshots. For the VM we access them through the shared files system area at CERN (cvmfs). Read

in this way, the conditions will be cached locally in your virtual machine the first time you run and so the CMSSW job will be slow. Fortunately, we already did this while setting up our VM, so our jobs will run much faster. In addition, those soft links he had to make are simply pointers to these areas.

On the other hand, in the Docker container, these database snapshots live locally in your /opt/cms-opendata-conddb directory. Running over them is much quicker.

Feel free to just replace the whole config file with the final version below (if using the VM, *uncomment and comment out the section in question appropriately*).

The final config file should look something like:

```
 GNU nano 2.0.9                File: Demo/DemoAnalyzer/demoanalyzer_cfg.py                Modified

import FWCore.ParameterSet.Config as cms

process = cms.Process("Demo")

process.load("FWCore.MessageService.MessageLogger_cfi")
process.MessageLogger.cerr.FwkReport.reportEvery = 1

process.maxEvents = cms.untracked.PSet( input = cms.untracked.int32(-1) )

process.source = cms.Source("PoolSource",
    # replace 'myfile.root' with the source file you want to use
    fileNames = cms.untracked.vstring(
    #    'file:myfile.root'
         #'root://eospublic.cern.ch//eos/opendata/cms/Run2012B/DoubleMuParked/AOD/22Jan2013-v1/10000/1EC938EF-ABEC-E211-94E0-90E6BA442F24.root'
    'root://eospublic.cern.ch//eos/opendata/cms/Run2012B/TauPlusX/AOD/22Jan2013-v1/20000/0040CF04-8E74-E211-AD0C-00266CFFA344.root',
    'root://eospublic.cern.ch//eos/opendata/cms/Run2012C/TauPlusX/AOD/22Jan2013-v1/310001/0EF85C5C-A787-E211-AFC9-003048C6942A.root'
    )
)

#uncomment to access the conditions data from the Virtual Machine (and comment out the Docker container set below)
#process.load('Configuration.StandardSequences.FrontierConditions_GlobalTag_cff')
#process.GlobalTag.connect = cms.string('sqlite_file:/cvmfs/cms-opendata-conddb.cern.ch/FT53_V21A_AN6_FULL.db')
#process.GlobalTag.globaltag = 'FT53_V21A_AN6::All'

#needed to access the conditions data from the Docker container
process.load('Configuration.StandardSequences.FrontierConditions_GlobalTag_cff')
process.GlobalTag.connect = cms.string('sqlite_file:/opt/cms-opendata-conddb/FT53_V21A_AN6_FULL_data_stripped.db')
process.GlobalTag.globaltag = 'FT53_V21A_AN6_FULL::All'

process.mytrigger = cms.EDAnalyzer('DemoAnalyzer',
        #InputCollection = cms.InputTag("muons")
    processName = cms.string("HLT"),
    datasetName = cms.string("TauPlusX"), #specific dataset example (for dumping info)
)

#process.load("HLTrigger.HLTfilters.hltHighLevel_cfi")
#process.hltHighLevel.HLTPaths = cms.vstring('HLT_Mu15*')


#process.p = cms.Path(process.hltHighLevel+process.demo)
process.p = cms.Path(process.mytrigger)
```

KUBERNETES CLUSTER

INTRODUCTION

In this demonstration we will show you the very basic way in which you can create a computer cluster (a Kubernetes cluster to be exact) in the cloud so you can do some data processing and analysis using those resources. In the process we will make sure you learn about the jargon. During the hands-on session tomorrow, a cluster similar to this one will be provided to you for the exercises.

BASIC CONCEPTS

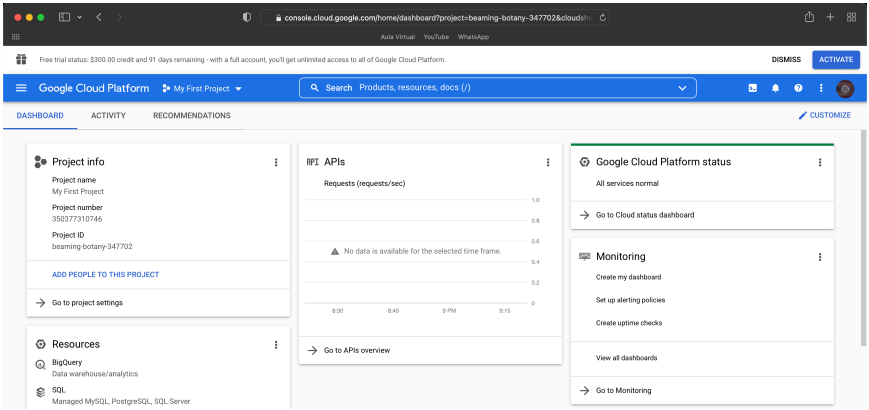
The Google Cloud Platform (GCP)

A place on the web that interfaces the user with all the different services that google provide on the cloud.

GCP Console

The exact name of the GCP interface where you can explore all the different services that GCP provides. They include, but are not limited to individual virtual machines, disk storage, kubernetes clusters, etc.

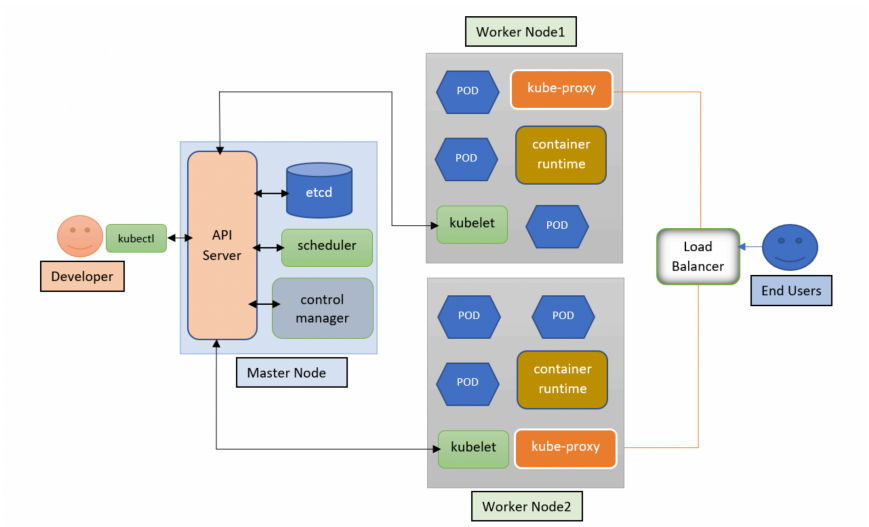
Google Kubernetes Engine



A Google service to create Kubernetes clusters and run conteinerized application and/or jobs/workflows.

Kubernetes (K8s)

Software which orchestrates containers in a computer cluster. You already had a chance to learn about its architecture.



Workflow

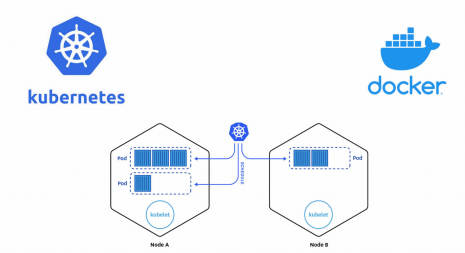
A series of sequential operations in order to achieve a final result. In our case it could be, for instance

skimming -> merging output files -> EventLoop analysis of resulting files -> Plotting histograms

In the context of the cloud, they are written in *yaml* files.

Pod

The smallest abstraction layer in a K8s cluster. For any practical purposes, a pod is an abstraction of a container running in the K8s cluster.



Deployment

This is an abstraction layer which is above pods. In practice, you always create deployments in K8s, not pods.

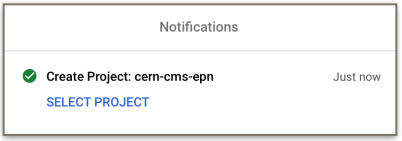
Argo

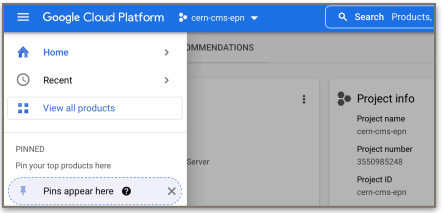
Argo Workflows is an open source container-native workflow engine for orchestrating parallel jobs on Kubernetes. In this case we will be using Docker.

CREATING YOUR OWN CLUSTER ON GKE

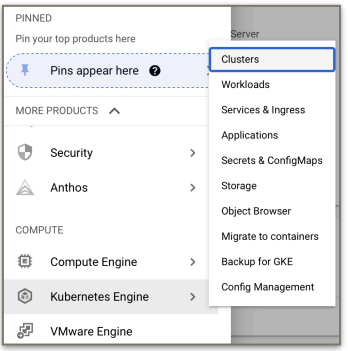
For the hands-on part of this lesson you will not have to create the cluster for yourself, it will be already done for you. For pedagogical reasons, however, we will show an example of how to do it by hand. The settings below should be good and cheap enough for CMSSW-related workflows.

- Get to the Console
- Create a new project or select one of your interest (if you already have one)

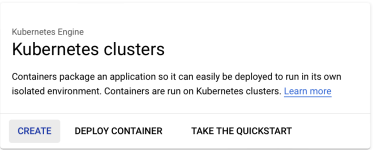




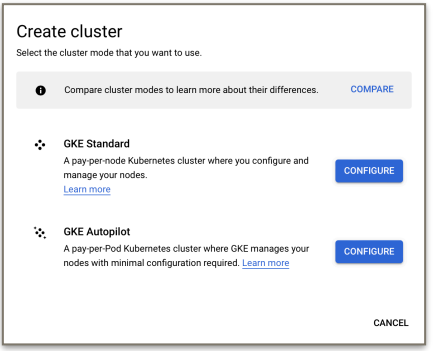
Click on the Kubernetes engine/clusters section on the left side menu



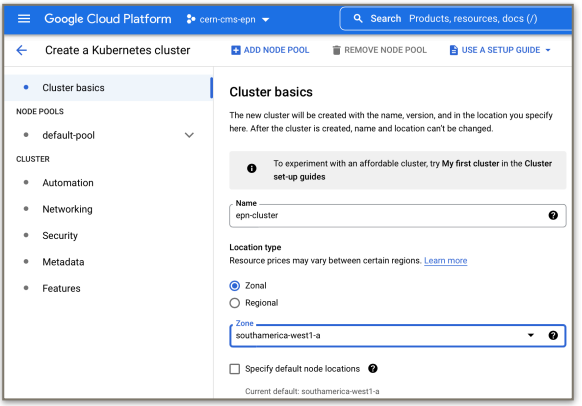
Select create cluster (standard)



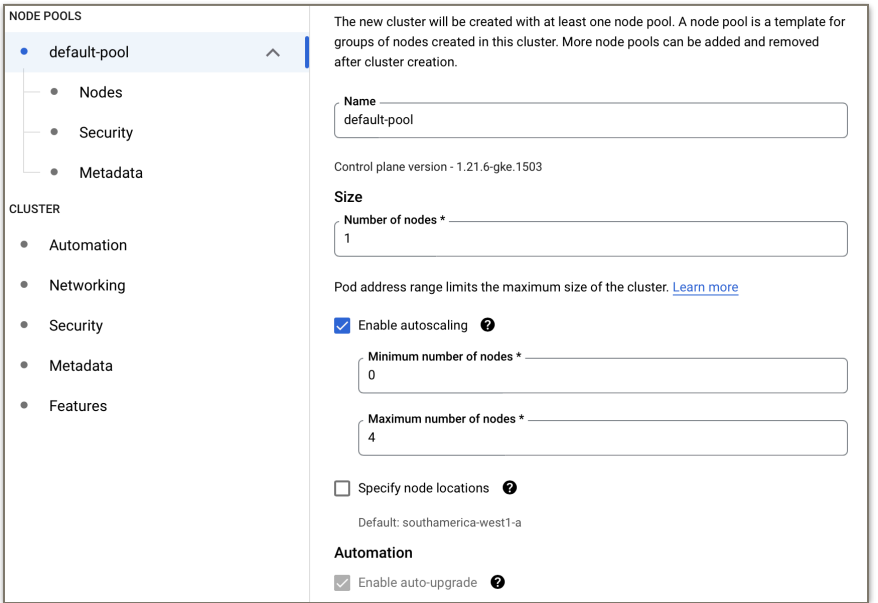
Click GKE Standard

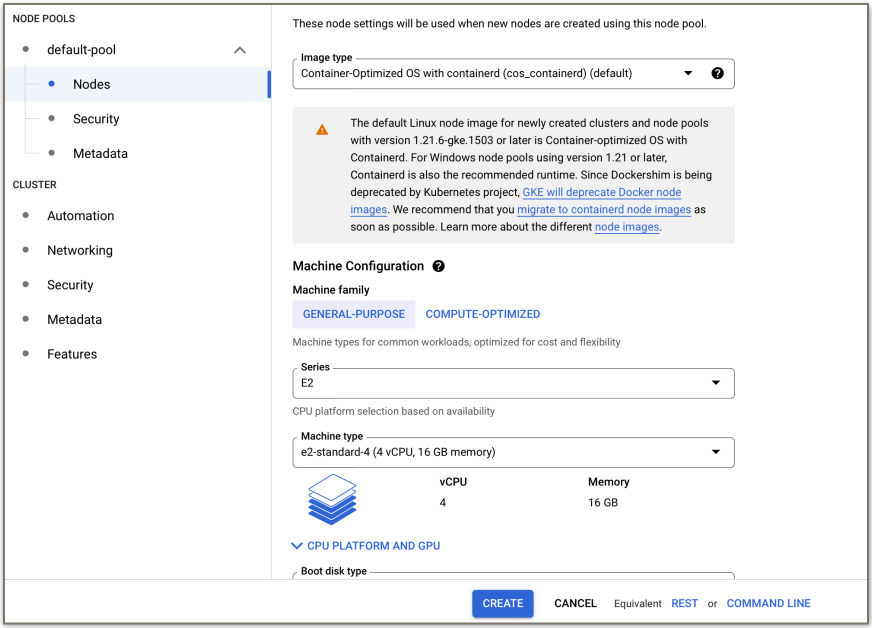


Give it a name



- Many ways to configure the cluster, but let's try an efficient one with autoscaling
- Go to default pool
- Choose size: 1 node

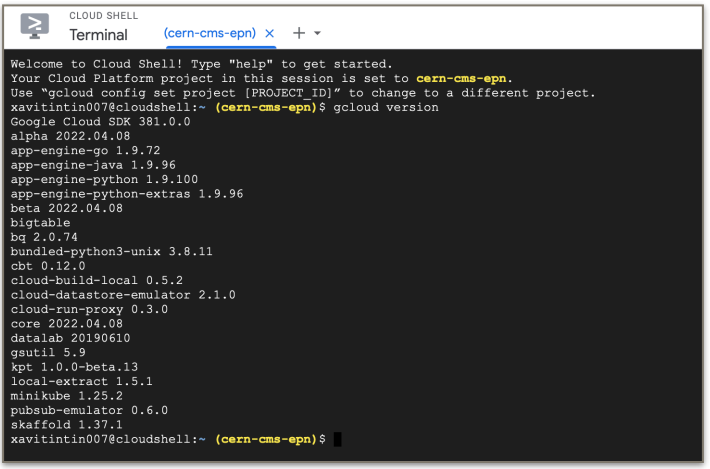




- Autoscaling 0 to 4
- Go to Nodes
- Choose a machine e2-standar-4
- Leave the rest as it is
- Hit create

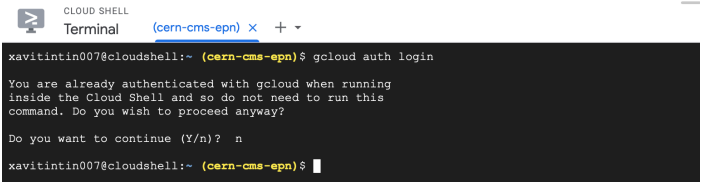
Creation will take while
While we wait, lets inspect the Cloud shell...

CLOUD SHELL



GCP provides an access machine so you can interact with their different services, including our newly created K8s cluster. This machine (and the terminal) is not really part of the cluster. As was said, it is an entry point. From here you could connect to your cluster.

Get verified for login, type: *gcloud auth login*

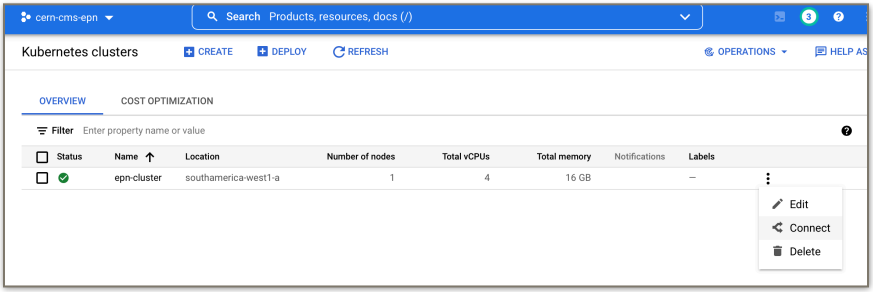


The *gcloud* command

The gcloud command-line interface is the primary CLI tool to create and manage Google Cloud resources. You can use this tool to perform many common platform tasks either from the command line or in scripts and other automations.

CONNECT TO YOUR CLUSTER

Once the cluster is ready (green check-mark should appear)
Click on the connect button of your cluster:



Execute that command in the cloud shell:

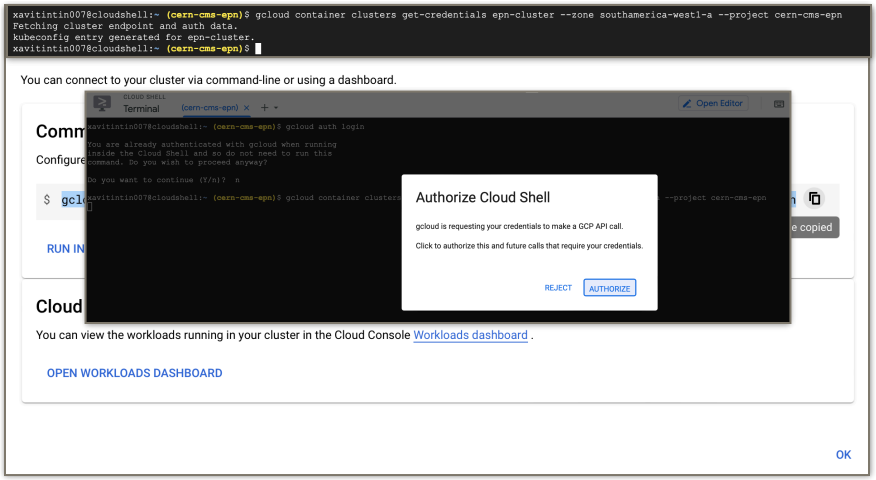
Click Authorize

Kubernetes is up and running

Key Points:

- *It takes just a few clicks to create you own K8s cluster*

KUBECTL AND ADDITIONAL TOOLS AND SERVICES



The kubectl command

Just as gcloud is the one command to rule them all for the GCP, the kubectl command is the main tool for interacting with your K8s cluster. You will use it to do essentially anything in the cluster. Here is the official cheatsheet, which is very useful but already very long.

Let's run a few examples.
Get the status of the nodes in your cluster:
kubectl get nodes

Get the cluster info:
kubectl cluster-info # Display addresses of the master and services

Let's list some kubernetes components:
Check pods
kubectl get pod

Check the services
kubectl get services

We don't have much going on. Let's create some components.
Inspect the create operation
kubectl create -h

CLOUD SHELL

Terminal (cern-cms-epn)

```
apiVersion: apps/v1
kind: Deployment
metadata:
    name: nginx-deployment
    labels:
      app: nginx
spec:
    replicas: 2
    selector:
      matchLabels:
        app: nginx
    template:
      metadata:
        labels:
          app: nginx
      spec:
        containers:
        - name: nginx
          image: nginx:1.16
          ports:
        - containerPort: 8080
```

Terminal (cern-cms-epn)

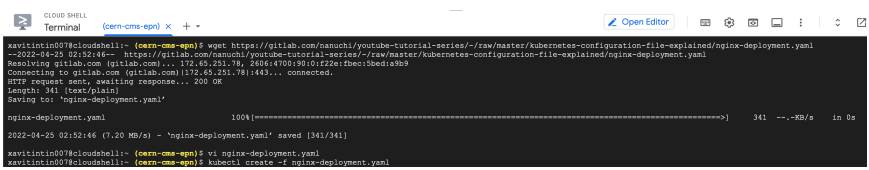
```
xavitintin007@cloudshell:~ (cern-cms-epn)$ kubectl create deployment mynginx-depl --image=nginx
deployment.apps/mynginx-depl created
xavitintin007@cloudshell:~ (cern-cms-epn)$ kubectl get deployments
NAME           READY   UP-TO-DATE   AVAILABLE   AGE
mynginx-depl   1/1     1            1           15s
xavitintin007@cloudshell:~ (cern-cms-epn)$ kubectl delete  deployment mynginx-depl
deployment.apps "mynginx-depl" deleted
```

Note there is no pod on the list, so in K8s you don't create pods but deployments. These will create pods, which will run under the hood.

Let's create an application, it does not matter which. Let's go for nginx:

kubectl create deployment mynginx-depl --image=nginx

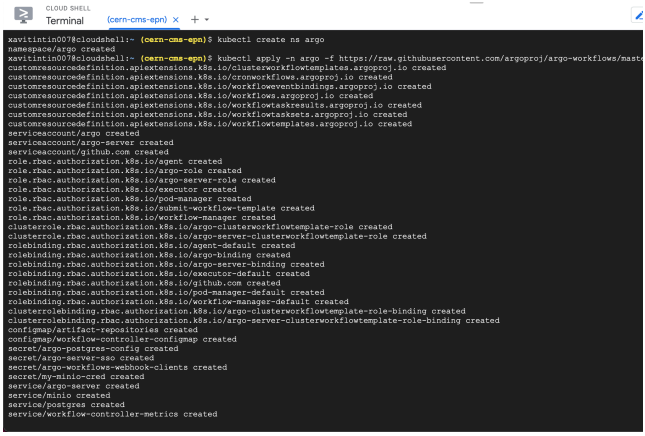
The nginx image will be pulled down from the Docker Hub. This is the



most minimalist way of creating a deployment.

Check the deployments
kubectl get deployment

Check pods



kubectl get pod

Yaml Files

Another way of creating components in a K8s cluster is trough yaml files. These are intuitive, logical and configurable, although very picky about identation.

Let's take a look at one of these files, nginx-deployment.yaml:
https://gitlab.com/nanuchi/youtube-tutorial-series/-/raw/master/kubernetes-configuration-file-explained/nginx-deployment.yaml

Let's delete our previous deployment and deploy using the yaml file:
Delete deployments
kubectl delete deployment mynginx-depl
Deploy using yaml files
kubectl apply -f nginx-deployment.yaml

ARGO

While jobs can also be run manually, a workflow engine makes defining and submitting jobs easier. In this tutorial, we use argo quick start page to install it:

Quick Start¶

To see how Argo Workflows work, you can install it and run examples of simple workflows and workflows that use artifacts.

Firstly, you'll need a Kubernetes cluster and kubectl set-up

Install Argo Workflows¶

To get started quickly, you can use the quick start manifest which will install Argo Workflow as well as some commonly used components:

kubectl create ns argo

kubectl apply -n argo -f https://raw.githubusercontent.com/argoproj/argo-workflows/master/manifests/quick-start-postgres.yaml

On GKE, you may need to grant your account the ability to create new clusterroles

kubectl create clusterrolebinding YOURNAME-cluster-admin-binding --clusterrole=cluster-admin --user=YOUREMAIL@gmail.com

kubectl create ns argo

kubectl apply -n argo -f https://raw.githubusercontent.com/argoproj/argo-workflows/stable/manifests/quick-start-postgres.yaml

Download argo CLI:

 # Download the binary

curl -sLO https://github.com/argoproj/argo-workflows/releases/download/v3.1.2/argo-linux-amd64.gz

 # Unzip

gunzip argo-linux-amd64.gz

 # Make binary executable

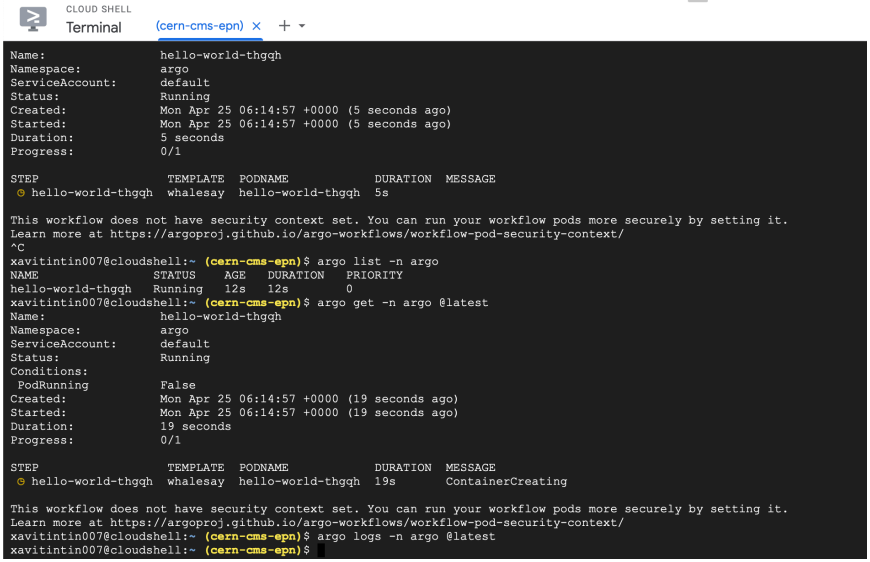
chmod +x argo-linux-amd64

 # Move binary to path

sudo mv ./argo-linux-amd64 /usr/local/bin/argo

 # Test installation

argo version



Run a simple test flow:

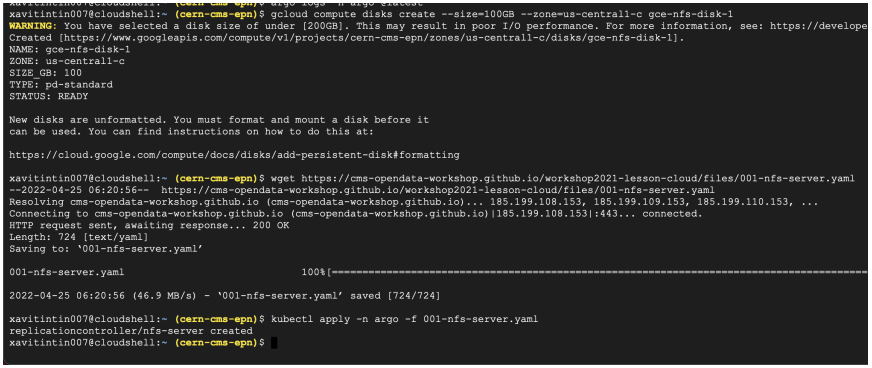
argo submit -n argo --watch https://raw.githubusercontent.com/argoproj/argo-workflows/master/examples/hello-world.yaml

argo list -n argo

argo get -n argo @latest

argo logs -n argo @latest

Storage volumes



If we run some application or workflow, we usually require a disk space where to dump our results. There is no persistent disk by default, we have to create it.

You could create a disk clicking on the web interface above, but lets do it faster in the command line.

Create the volume (disk) we are going to use

gcloud compute disks create --size=100GB --zone=us-central1-c gce-nfs-disk-1

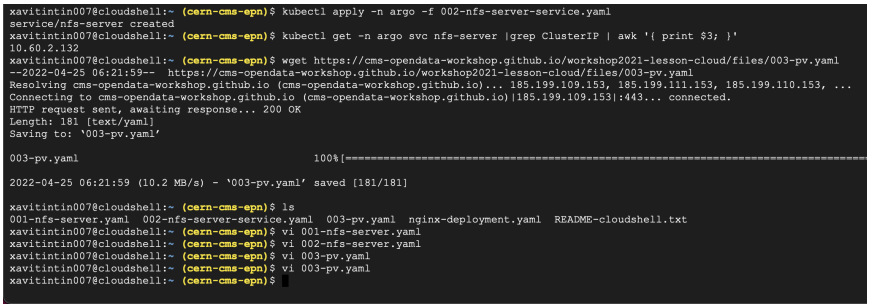
Set up an nfs server for this disk:

wget https://cms-opendata-workshop.github.io/workshop2021-lesson-cloud/files/001-nfs-server.yaml

kubectl apply -n argo -f 001-nfs-server.yaml

Set up a nfs service, so we can access the server:

wget https://cms-opendata-workshop.github.io/workshop2021-lesson-cloud/files/002-nfs-server-service.yaml



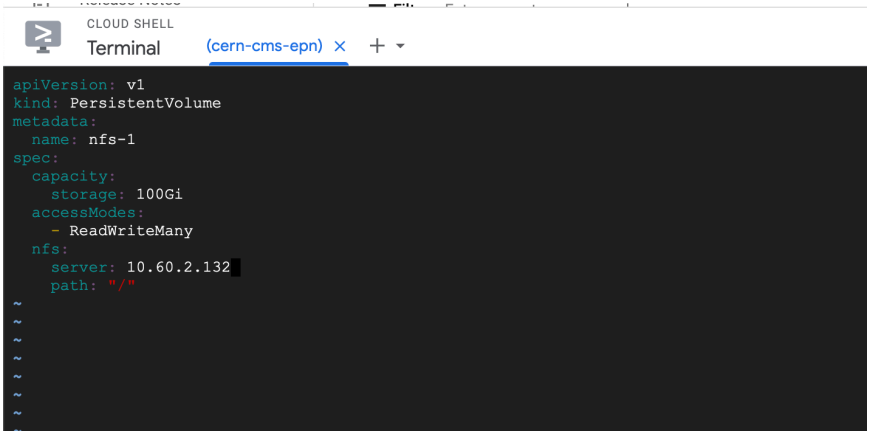
kubectl apply -n argo -f 002-nfs-server-service.yaml

Let's find out the IP of the nfs server:

kubectl get -n argo svc nfs-server |grep ClusterIP | awk '{ print $3; }'

Let's create a persisten volume out of this nfs disk. Note that persisten volumes are not namespaced they are available to the whole cluster.

We need to write that IP number above into the appropriate place in this file:



wget https://cms-opendata-workshop.github.io/workshop2021-lesson-cloud/files/003-pv.yaml

Deploy:
kubectl apply -f 003-pv.yaml
Check:

```
xavitintin007@cloudshell:~ (cern-cms-epn)$ vi 002-nfs-server.yaml
xavitintin007@cloudshell:~ (cern-cms-epn)$ vi 003-pv.yaml
xavitintin007@cloudshell:~ (cern-cms-epn)$ vi 003-pv.yaml
xavitintin007@cloudshell:~ (cern-cms-epn)$ kubectl apply -f 003-pv.yaml
persistentvolume/nfs-1 created
xavitintin007@cloudshell:~ (cern-cms-epn)$ kubectl get pv
NAME    CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS      CLAIM   STORAGECLASS   REASON   AGE
nfs-1   100Gi      RWX            Retain           Available                                   7s
```

kubectl get pv
Apps can claim persistent volumes through persistent volume claims (pvc). Let's create a pvc:
wget https://cms-opendata-workshop.github.io/workshop2021-lesson-cloud/files/003-pvc.yaml
kubectl apply -n argo -f 003-pvc.yaml
Check:
kubectl get pvc -n argo

```
xavitintin007@cloudshell:~ (cern-cms-epn)$ wget https://cms-opendata-workshop.github.io/workshop2021-lesson-clo
--2022-04-25 06:26:29--  https://cms-opendata-workshop.github.io/workshop2021-lesson-cloud/files/003-pvc.yaml
Resolving cms-opendata-workshop.github.io (cms-opendata-workshop.github.io)... 185.199.109.153, 185.199.111.153
Connecting to cms-opendata-workshop.github.io (cms-opendata-workshop.github.io)|185.199.109.153|:443... connect
HTTP request sent, awaiting response... 200 OK
Length: 194 [text/yaml]
Saving to: '003-pvc.yaml'

003-pvc.yaml                                  100%[=================================================================

2022-04-25 06:26:29 (17.9 MB/s) - '003-pvc.yaml' saved [194/194]

xavitintin007@cloudshell:~ (cern-cms-epn)$ kubectl apply -n argo -f 003-pvc.yaml
persistentvolumeclaim/nfs-1 created
xavitintin007@cloudshell:~ (cern-cms-epn)$ kubectl get pvc -n argo
NAME    STATUS   VOLUME   CAPACITY   ACCESS MODES   STORAGECLASS   AGE
nfs-1   Bound    nfs-1    100Gi      RWX                           8s
xavitintin007@cloudshell:~ (cern-cms-epn)$
```

Deleting Workspace

Commands:

kubectl delete ns argo
*rm**
gcloud compute disks list

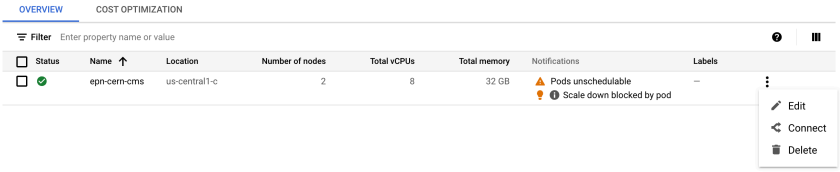
Only delete gce-nfs-disk-..... with

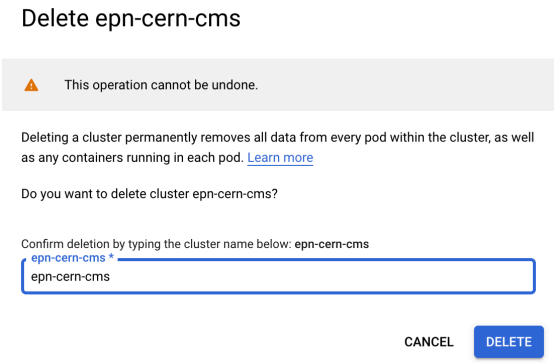
```
xavitintin007@cloudshell:~ (cern-cms-epn)$ gcloud compute disks list
NAME: gce-nfs-disk-1
LOCATION: us-central1-c
LOCATION_SCOPE: zone
SIZE_GB: 100
TYPE: pd-standard
STATUS: READY

NAME: gke-epn-cern-cms-default-pool-d23d8bc4-svf4
LOCATION: us-central1-c
LOCATION_SCOPE: zone
SIZE_GB: 100
TYPE: pd-standard
STATUS: READY

NAME: gke-epn-cern-cms-default-pool-d23d8bc4-wtzj
LOCATION: us-central1-c
LOCATION_SCOPE: zone
SIZE_GB: 100
TYPE: pd-standard
STATUS: READY
xavitintin007@cloudshell:~ (cern-cms-epn)$
```

gcloud compute disks delete gce-nfs-disk-001 --zone=us-central1-c

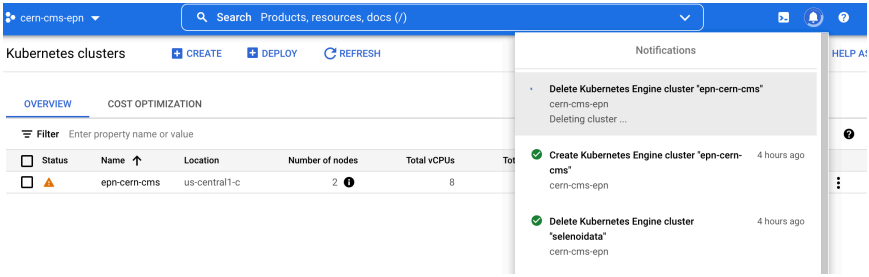
```
xavitintin007@cloudshell:~ (cern-cms-epn)$ gcloud compute disks delete gce-nfs-disk-1 --zone=us-central1-c
The following disks will be deleted:
 - [gce-nfs-disk-1] in [us-central1-c]

Do you want to continue (Y/n)?  y

Deleted [https://www.googleapis.com/compute/v1/projects/cern-cms-epn/zones/us-central1-c/disks/gce-nfs-disk-1].
xavitintin007@cloudshell:~ (cern-cms-epn)$
```

Delete kubernetes Cluster:





Standby to see the complete deletion of the cluster



Perfect you're ready to start over