# Table of Contents

# 3.5.3 Examples of FWLite macros

## Contents



Detailed Review status

## 3.5.3.1. Introduction

You should view FWLite as a way to access bare ROOT input files of the EDM, with the capability to read and recognise CMSSW DataFormats while the performance advantages of I/O and CPU consumption of the bare ROOT access are sustained. It should be emphasised that FWLite is not an exclusive alternative to the use of the full framework. Rather we propose to use FWLite and full framework in parallel depending on the requirements of your analysis. Many efforts have been made during the developments of FWLite to achieve a user interface maximally facilitating the interchange of code between both paradigms. AT (Analysis Tools) strongly recommends **NOT to rely on *CINT* within ROOT with interactive macros!** We recommend to use compiled code instead. On this page you will find examples of compiled FWLite executables that can be used as starting points for your own analysis. The emphasise is to start from the simplest and most basic skeletons and to give examples how to increase the level of complexity step by step. Finally we will point you to a fully equipped example to do an analysis both with FWLite and within the full EDM framework using the same code. We will assume that you work on *lxplus*, but following the instructions given above you can run these examples on any other computer system that has FWLite installed.

## 3.5.3.2. Setting up of the environment

First of all connect to *lxplus* and go to some working directory. You can choose any directory, provided that you have enough space. You need some minimum of free disc space to do the exercises described here. We recommend you to use your *~/scratch0* space. In case you don't have this (or do not even know what it is) check your quota typing `fs lq` and follow this link. If you don't have enough space, you may instead use the temporary space (*/tmp/your_user_name*), but be aware that this is lost once you log out from *lxplus* (or app. within a day). We will expect in the following that you have such a *~/scratch0* directory.

```
ssh lxplus
[ ... enter password ... ]
cd scratch0/
```

cmsrel CMSSW_5_3_12 ... Hide this help
Create a local release area and enter it, set up the environment (using the *cmsenv* command)

```
cmsrel CMSSW_5_3_12
cd CMSSW_5_3_12/src
cmsenv
```

Created a local release area and enter it, set up the environment (using the_cmsenv_command)

```
setenv SCRAM_ARCH slc6_amd64_gcc530
cmsrel CMSSW_8_1_0_pre16
cd CMSSW_8_1_0_pre16/src
cmsenv
```

**Note:** Please for the time being use the set of tags as described here on top of the recommended release to be able to follow the examples given in the following

Checkout the FWLite package, which contains the examples that will be discussed in the following to be able to inspect them in your favourite editor. You can use the *addpkg* command, which will automatically pick up the proper *cvs* tag from the release.

```
git cms-addpkg PhysicsTools/FWLite
```

**Note:** We are going to go through the exmaples in the FWLite package. The *addpkg* tool picks the right *cvs* version of this package from the release and copies it to your local release area.

**Note:** Don't forget to recompile the package whenever you changed any of the executables. You have to *rehash* the environment after **each compilation** to make sure that the cash of the shell prompt is refreshed. You can do this using the *cmsenv* command or better the command *rehash*.

**Note:** In the current implementation all examples will require a *patTuple.root* file unless stated otherwise. Have a look to WorkBookPATTupleCreationExercise to learn how to create such a PAT tuple.

## 3.5.3.3. Example 1: Access to event information

The first example shows how to access simple event information. We are going to show how to access the run-wise luminosity and peak luminosity from the *edm::Event* content. To run the example type the following command in the *src* directory of your local release area (as long as you don't change any code in the local release area you will not have to compile):

```
FWLiteLumiAccess inputFiles=root://eoscms//eos/cms/store/relval/CMSSW_5_3_11_patch6/RelValTTbar/G
```

**Note:** As you can see we made use of the command line argument *inputFiles* as described on SWGuideCommandLineParsing. Here we make use of a RECO input file from which we want to read the LumiBlock information.

We will now explain how this executable was compiled: To compile an individual executable as part of a CMSSW package the only thing you have to do is to make it known to the *scram* build mechanism via a corresponding line in a given *BuildFile.xml*. We followed the general convention to keep individual executable in a dedicated *bin* directory of the package. There you can find the corresponding BuildFile.xml and within this file the following lines:

```
<use   name="root"/>
<use   name="boost"/>
<use   name="rootcintex"/>
<use   name="FWCore/FWLite"/>
<use   name="DataFormats/FWLite"/>
<use   name="DataFormats/Luminosity"/>
<use   name="FWCore/PythonParameterSet"/>
<use   name="CommonTools/Utils"/>
<use   name="PhysicsTools/FWLite"/>
<use   name="PhysicsTools/Utilities"/>
<use   name="PhysicsTools/SelectorUtils"/>
```

```
<environment>
  <bin   file="FWLiteLumiAccess.cc"></bin>
  <bin   file="FWLiteHistograms.cc"></bin>
  <bin   file="FWLiteWithPythonConfig.cc"></bin>
  <bin   file="FWLiteWithSelectorUtils.cc"></bin>
</environment>
```

**Note:** As you can see the *BuildFile.xml* contains nothing more but compiler directives in *xml* style. The first lines indicate the libraries that are used by the executables located in the directory followed by a list of *.cc* files to be picked up for compilation. Each source will be compiled into an individual executable with the name of the file (omitting the *.cc* ending). You can invoke the executable from the shell prompt after refreshing the cash after compilation (using *cmsenv* or better *rehash*), as shown above. The important line for this example is:

```
<bin   file="FWLiteLumiAccess.cc"></bin>
```

In the *FWLiteLumiAccess.cc* file you can find a basic skeleton to write an FWLite executable and to access some extra information from the *edm::Event* content:

```
int main(int argc, char ** argv){
  // load framework libraries
  gSystem->Load( "libFWCoreFWLite" );
  AutoLibraryLoader::enable();

  // initialize command line parser
  optutl::CommandLineParser parser ("Analyze FWLite Histograms");

  // parse arguments
  parser.parseArguments (argc, argv);
  std::vector<std::string> inputFiles_ = parser.stringVector("inputFiles");

  for(unsigned int iFile=0; iFile<inputFiles_.size(); ++iFile){
    // open input file (can be located on castor)
    TFile* inFile = TFile::Open(inputFiles_[iFile].c_str());
    if( inFile ){
      fwlite::Event ev(inFile);
      fwlite::Handle<LumiSummary> summary;

      std::cout << "---------- Accessing by event ----------------" << std::endl;

      // get run and luminosity blocks from events as well as associated
      // products. (This works for both ChainEvent and MultiChainEvent.)
      for(ev.toBegin(); !ev.atEnd(); ++ev){
// get the Luminosity block ID from the event
std::cout << " Luminosity ID " << ev.getLuminosityBlock().id() << std::endl;
// get the Run ID from the event
std::cout <<" Run ID " << ev.getRun().id()<< std::endl;
// get the Run ID from the luminosity block you got from the event
std::cout << "Run via lumi " << ev.getLuminosityBlock().getRun().id() << std::endl;
// get the integrated luminosity (or any luminosity product) from
// the event
        summary.getByLabel(ev.getLuminosityBlockRange(),"lumiProducer");
      }

      //...

    }
  }
  return 0;
}
```

**Note:** Each C++ executable starts with a *main* function. We allow arguments to be passed on the to executable. The next lines enable the ROOT AutoLibraryLoader, which should be enabled for each FWLite

executable. Next we make use of command line parsing as described on SWGuideCommandLineParsing. The command line option that we read in is the option *inputFiles*. In the following the input files are looped and the event within each input file is accessed and looped. Within the event loop you can find the corresponding lines how to access luminosity and run information from the *edm::Event* content. You can find the implementation of this executable in the FWLiteLumiAccess.cc file in the *bin* directory of the package.

## 3.5.3.4. Example 2: Plotting histograms

The second example shows how to book and fill histograms from objects collections in the event. To run the example type the following command in the *src* directory of your local release area (again this is possible without compiling):

```
FWLiteHistograms inputFiles=root://eoscms//eos/cms/store/relval/CMSSW_5_3_11_patch6/RelValTTbar/G
```

The corresponding line for compilation in the *BuildFile.xml* is the following:

```
<bin   file="FWLiteLumiHistograms.cc"></bin>
```

**Note:** In this example more command line options are used:

- *inputFiles*: to pass on a vector of input file paths.
- *outputFile*: to pass an output file name.
- *maxEvents*: to pass on the maximal number of events to loop.
- *outputEvery*: to indicate after how many events a report should be sent to the prompt.

You can find the read out of these options in the *main* function of the *FWLiteHistograms.cc* file:

```
// ...

  // initialize command line parser
  optutl::CommandLineParser parser ("Analyze FWLite Histograms");

  // set defaults
  parser.integerValue ("maxEvents"  ) = 1000;
  parser.integerValue ("outputEvery") =   10;
  parser.stringValue  ("outputFile" ) = "analyzeFWLiteHistograms.root";

  // parse arguments
  parser.parseArguments (argc, argv);
  int maxEvents_ = parser.integerValue("maxEvents");
  unsigned int outputEvery_ = parser.integerValue("outputEvery");
  std::string outputFile_ = parser.stringValue("outputFile");
  std::vector<std::string> inputFiles_ = parser.stringVector("inputFiles");

  // book a set of histograms
  fwlite::TFileService fs = fwlite::TFileService(outputFile_.c_str());
  TFileDirectory dir = fs.mkdir("analyzeBasicPat");
  TH1F* muonPt_  = dir.make<TH1F>("muonPt"  , "pt"  ,   100,   0., 300.);
  TH1F* muonEta_ = dir.make<TH1F>("muonEta" , "eta" ,   100,  -3.,   3.);
  TH1F* muonPhi_ = dir.make<TH1F>("muonPhi" , "phi" ,   100,  -5.,   5.);
  TH1F* mumuMass_= dir.make<TH1F>("mumuMass", "mass",    90,  30.,  120.);

  // ...
```

**Note:** As you see we provide default values for the options *maxEvents*, *outputEvery* and *outputFile*. In the following section a set of histograms are booked making use of the TFileService, which takes automatic care of handling the individual file systems of the different input and output files. To learn more about the TFileService have a look to SWGuideTFileService.

In the event loop we do some gymnastics for the event report and to stop the loop after *maxEvents* have been processed. We access the collection of _reco::Muon_'s, loop the muons and fill the histograms.

```
for(ev.toBegin(); !ev.atEnd(); ++ev, ++ievt){
edm::EventBase const & event = ev;
// break loop if maximal number of events is reached
if(maxEvents_>0 ? ievt+1>maxEvents_ : false) break;
// simple event counter
if(outputEvery_!=0 ? (ievt>0 && ievt%outputEvery_==0) : false)
  std::cout << "  processing event: " << ievt << std::endl;

// Handle to the muon collection
edm::Handle<std::vector<Muon> > muons;
        event.getByLabel(std::string("muons"), muons);

// loop muon collection and fill histograms
for(std::vector<Muon>::const_iterator mu1=muons->begin(); mu1!=muons->end(); ++mu1){
          muonPt_ ->Fill( mu1->pt () );
          muonEta_->Fill( mu1->eta() );
          muonPhi_->Fill( mu1->phi() );
  if( mu1->pt()>20 && fabs(mu1->eta())<2.1 ){
    for(std::vector<Muon>::const_iterator mu2=muons->begin(); mu2!=muons->end(); ++mu2){
      if(mu2>mu1){ // prevent double conting
if( mu1->charge()*mu2->charge()<0 ){ // check only muon pairs of unequal charge
  if( mu2->pt()>20 && fabs(mu2->eta())<2.1 ){
                    mumuMass_->Fill( (mu1->p4()+mu2->p4()).mass() );
                  }
                }
              }
            }
          }
        }

        //...
```

You can find the implementation of this executable in the FWLiteLumiHistograms.cc file in the *bin* directory of the package.

## 3.5.3.5. Example 3: Using *python* configuration

You can get the same results using the *python* configuration you are used to from the full framework. It is much more powerful in passing on an arbitrary number of parameters and is not restricted to a single command line. In case you are not familiar with the *python* configuration language used within CMS have a look to WorkBookConfigFileIntro. To run the example type the following command in the *src* directory of your local release area:

```
FWLiteWithPythonConfig PhysicsTools/FWLite/test/fwliteWithPythonConfig_cfg.py
```

The corresponding line in the *BuildFile.xml* is the following:

```
<bin   file="FWLiteWithPythonConfig.cc"></bin>
```

**Note:** For this examples we completely replaced the command line parsing by the *python* configuration file mechanism. You can find the fwliteWithPythonConfig_cfg.py configuration file in the *test* directory of the package. It contains the following parameters:

```
import FWCore.ParameterSet.Config as cms

process = cms.PSet()
```

```
process.fwliteInput = cms.PSet(
    fileNames   = cms.vstring('rfio:///castor/cern.ch/cms/store/relval/CMSSW_4_1_3/RelValTTbar/GE
    maxEvents   = cms.int32(-1),                              ## optional
    outputEvery = cms.uint32(10),                             ## optional
)

process.fwliteOutput = cms.PSet(
    fileName  = cms.string('analyzeFWLiteHistograms.root'),   ## mandatory
)

process.muonAnalyzer = cms.PSet(
    ## input specific for this analyzer
    muons = cms.InputTag('muons')
)
```

You can find the readout of the configuration file in the *main* function of the *FWLiteWithPythonConfig.cc* file:

```
// ...

  // parse arguments
  if ( argc < 2 ) {
    std::cout << "Usage : " << argv[0] << " [parameters.py]" << std::endl;
    return 0;
  }

  if( !edm::readPSetsFrom(argv[1])->existsAs<edm::ParameterSet>("process") ){
    std::cout << " ERROR: ParametersSet 'process' is missing in your configuration file" << std::
  }
  // get the python configuration
  const edm::ParameterSet& process = edm::readPSetsFrom(argv[1])->getParameter<edm::ParameterSet>
  fwlite::InputSource inputHandler_(process); fwlite::OutputFiles outputHandler_(process);


  // now get each parameter
  const edm::ParameterSet& ana = process.getParameter<edm::ParameterSet>("muonAnalyzer");
  edm::InputTag muons_( ana.getParameter<edm::InputTag>("muons") );

  // book a set of histograms
  fwlite::TFileService fs = fwlite::TFileService(outputHandler_.file().c_str());
  TFileDirectory dir = fs.mkdir("analyzeBasicPat");
  TH1F* muonPt_   = dir.make<TH1F>("muonPt"  , "pt"  ,   100,   0.,  300.);
  TH1F* muonEta_  = dir.make<TH1F>("muonEta" , "eta" ,   100,  -3.,    3.);
  TH1F* muonPhi_  = dir.make<TH1F>("muonPhi" , "phi" ,   100,  -5.,    5.);
  TH1F* mumuMass_= dir.make<TH1F>("mumuMass", "mass",    90,   30., 120.);

  // ...
```

**Note:** The first part just hands over the single argument, the path to the configuration file. This path is passed on to the *PythonProcessDesc* constructor from which the *edm::ParameterSet* is derived. The parameters are then read out equivalent to the full framework case. In addition to the parameters in Example 3 the collection label for the muon collection is also treated as a parameter. You can find the implementation of this executable in the FWLiteLumiWithPythonConfig.cc file in the *bin* directory of the package.

## 3.5.3.6. Example 4: Using event selectors

In this example we will introduce you to some more advanced features of the SelectorUtils package. We will make use of an EventSelector to apply event selections, rather than doing the selection in the main event loop. This will allow to keep track of the statistics of the event selection. More details about Selectors can be found on SWGuidePATSelectors. This particular example will create a simple W selector that will select events with muons with pt>20 GeV, and MET>20 GeV (using PAT the MET will per default be type1 and muon

corrected calorimeter MET). To run the example type the following command in the *src* directory of your local release area:

```
FWLiteWithSelectorUtils PhysicsTools/FWLite/test/fwliteWithSelectorUtils_cfg.py
  processing event: 10
  ...
     0 :              Muon Pt         22
     1 :                  MET         86
```

The last lines correspond to the number of events that have passed the muon pt, and MET cut, respectively. This can be made as complicated as you wish and automatically keeps track of your cut flow.

**Note:** This example is an extension of Example 3. The configuration file has therefore been slightly modified:

```
import FWCore.ParameterSet.Config as cms

process = cms.PSet()

process.fwliteInput = cms.PSet(
    fileNames   = cms.vstring('rfio:///castor/cern.ch/cms/store/relval/CMSSW_4_1_3/RelValTTbar/GE
    maxEvents   = cms.int32(-1),                              ## optional
    outputEvery = cms.uint32(10),                             ## optional
)

process.fwliteOutput = cms.PSet(
    fileName  = cms.string('analyzeFWLiteHistograms.root'),  ## mandatory
)

process.selection = cms.PSet(
        muonSrc      = cms.InputTag('muons'),
        metSrc       = cms.InputTag('metJESCorAK5CaloJetMuons'),
        muonPtMin    = cms.double(20.0),
        metMin       = cms.double(20.0),
       #cutsToIgnore = cms.vstring('MET')
)
```

**Note:** The additional *edm::ParameterSet selection* is passed on to the W boson selector in the implementation of the executable:

```
// get the python configuration
  const edm::ParameterSet& process = edm::readPSetsFrom(argv[1])->getParameter<edm::ParameterSet>
  fwlite::InputSource inputHandler_(process); fwlite::OutputFiles outputHandler_(process);

  // initialize the W selector
  edm::ParameterSet selection = process.getParameter<edm::ParameterSet>("selection");
  WSelector wSelector( selection ); pat::strbitset wSelectorReturns = wSelector.getBitTemplate();

  // book a set of histograms
  fwlite::TFileService fs = fwlite::TFileService(outputHandler_.file().c_str());
  TFileDirectory theDir = fs.mkdir("analyzeBasicPat");
  TH1F* muonPt_  = theDir.make<TH1F>("muonPt", "pt",    100,  0.,300.);
  TH1F* muonEta_ = theDir.make<TH1F>("muonEta","eta",   100, -3.,  3.);
  TH1F* muonPhi_ = theDir.make<TH1F>("muonPhi","phi",   100, -5.,  5.);
```

The definition of the Selector can be found as an own class in the WSelector.h file in the *interface* directory of the package. The class is derived from the EventSelector base class of the SelectorUtils package:

```
class WSelector : public EventSelector {

public:
  /// constructor
  WSelector(edm::ParameterSet const& params) :
```

```
    muonSrc_(params.getParameter<edm::InputTag>("muonSrc")),
    metSrc_ (params.getParameter<edm::InputTag>("metSrc"))
  {
    double muonPtMin = params.getParameter<double>("muonPtMin");
    double metMin    = params.getParameter<double>("metMin");
    push_back("Muon Pt", muonPtMin );
    push_back("MET"    , metMin    );
    set("Muon Pt"); set("MET");
    wMuon_ = 0; met_ = 0;
    if ( params.exists("cutsToIgnore") ){
      setIgnoredCuts( params.getParameter<std::vector<std::string> >("cutsToIgnore") );
    }
    retInternal_ = getBitTemplate();
  }
  /// destructor
  virtual ~WSelector() {}
  /// return muon candidate of W boson
  pat::Muon const& wMuon() const { return *wMuon_;}
  /// return MET of W boson
  pat::MET  const& met()   const { return *met_;  }

  /// here is where the selection occurs
  virtual bool operator()( edm::EventBase const & event, pat::strbitset & ret){
    ret.set(false);
    // Handle to the muon collection
    edm::Handle<std::vector<pat::Muon> > muons;
    // Handle to the MET collection
    edm::Handle<std::vector<pat::MET> > met;
    // get the objects from the event
    bool gotMuons = event.getByLabel(muonSrc_, muons);
    bool gotMET   = event.getByLabel(metSrc_, met   );
    // get the MET, require to be > minimum
    if( gotMET ){
      met_ = &met->at(0);
      if( met_->pt() > cut("MET", double()) || ignoreCut("MET") )
        passCut(ret,"MET");
    }
    // get the highest pt muon, require to have pt > minimum
    if( gotMuons ){
      if( !ignoreCut("Muon Pt") ){
if( muons->size() > 0 ){
          wMuon_ = &muons->at(0);
  if( wMuon_->pt() > cut("Muon Pt", double()) || ignoreCut("Muon Pt") )
            passCut(ret,"Muon Pt");
        }
      }
      else{
        passCut(ret,"Muon Pt");
      }
    }
    setIgnored(ret);
    return (bool)ret;
  }

protected:
  /// muon input
  edm::InputTag muonSrc_;
  /// met input
  edm::InputTag metSrc_;
  /// muon candidate from W boson
  pat::Muon const* wMuon_;
  /// MET from W boson
  pat::MET const* met_;
};
```

The selection is applied in the following lines in the *main* function of the executable code in the *FWLiteWithSelectorUtils.cc* file:

```
if ( wSelector(event, wSelectorReturns ) ) {
  pat::Muon const & wMuon = wSelector.wMuon();
          muonPt_ ->Fill( wMuon.pt()  );
          muonEta_->Fill( wMuon.eta() );
          muonPhi_->Fill( wMuon.phi() );
        }
```

As a feature of the base class you can switch off all parts of the selection in our example by uncommenting the following line in the configuration file (indicating the corresponding selection step):

```
cutsToIgnore = cms.vstring('MET')     # <<<----- Uncomment this line
```

This configuration parameter is picked up in these lines in the executable code:

```
if ( params.exists("cutsToIgnore") )
    setIgnoredCuts( params.getParameter<std::vector<std::string> >("cutsToIgnore") );
```

**Note:** In general the Selector works with the old PAT feature of `strbitset`. You can operate it completely without this feature though, by changing the argument of the if statement in the *main* function of the code from `if ( wSelector(event, wSelectorReturns ) )` to `if ( wSelector(event) )` and adding the following hook to the constructor of your selector:

```
retInternal_ = getBitTemplate();
```

**Note:** Recently the Selectors have been speeded up in CPU performance by introducing selection index cashing. By un/commenting the *.h* files according to the following lines in the *FWLiteWithSelectorUtils.cc* file:

```
//#include "PhysicsTools/FWLite/interface/WSelector.h"
#include "PhysicsTools/FWLite/interface/WSelectorFast.h"
```

you can see the effect (don't forget to recompile and to refresh the cash of your shell). You can have a look into the WSelectorFast.h file in the *interface* directory of the package to check the differences to the common *WSelector.h* file in the same directory.

You can find the implementation of this executable in the FWLiteLumiWithSelectorUtils.cc file in the *bin* directory of the package. You can find a more elaborate example of a V+Jets selector on CMS.VplusJetsSelectors.

# 3.5.3.7 Example 5:Using FWLite and full framework in parallel

In this example we will show you how to write your analysis code that can be used both in FWLite and full framework following only a few very simple rules. It makes use of the BasicAnalyzer class, a powerful tool of the UtilAlgos package. You can concentrate on the actual analysis code and with less than 10 lines will be able to transform it either into a FWLite executable as described above or into a full framework EDAnalyzer. For the 10 extra lines you gain that you don't have to care about the FWLite event loop or any *python* configuration parameter parsing any more in FWLite case, which is taken care for you. You can combine this kind of programming style with any examples and FWLite features shown on this page. **This is the way of doing an analysis as recommended by the AT (Analysis Tools) group**. The example given here is equivalent to Example 3. To run it type the following command in the *src* directory of your local release area:

```
FWLiteWithBasicAnalyzer PhysicsTools/FWLite/test/fwliteWithPythonConfig_cfg.py
```

**Note:** As you see it also makes use of the same configuration file as Example 3. The example though originates from the *PhysicsTools/UtilAlgos* package. Having a look into the FWLiteWithBasicAnalyzer.cc file there, for the implementation of the executable you will recognise that it is significantly shorter:

```
int main(int argc, char* argv[])
{
  // load framework libraries
  gSystem->Load( "libFWCoreFWLite" );
  AutoLibraryLoader::enable();

  // only allow one argument for this simple example which should be the
  // the python cfg file
  if ( argc < 2 ) {
    std::cout << "Usage : " << argv[0] << " [parameters.py]" << std::endl;
    return 0;
  }
  if( !edm::readPSetsFrom(argv[1])->existsAs<edm::ParameterSet>("process") ){
    std::cout << " ERROR: ParametersSet 'plot' is missing in your configuration file" << std::end
  }

  WrappedFWLiteMuonAnalyzer ana(edm::readPSetsFrom(argv[1])->getParameter<edm::ParameterSet>("pro
  ana.beginJob();
  ana.analyze();
  ana.endJob();
  return 0;
}
```

**Note:** What remains is the parameter check to read in the *python* configuration file. The *edm::ParameterSet* is handed over to the *WrappedFWLiteAnalyzer* object, which is treated in maximal analogy to an EDAnalyzer in the following. This object has been declared by a simmple template expansion in the first lines:

```
#include "PhysicsTools/FWLite/interface/BasicMuonAnalyzer.h"
#include "PhysicsTools/UtilAlgos/interface/FWLiteAnalyzerWrapper.h"

typedef fwlite::AnalyzerWrapper<BasicMuonAnalyzer> WrappedFWLiteMuonAnalyzer;
```

The *FWLiteAnalyzerWrapper.h* is just a simple tool. You can use it as a black box if you like or have a short glimpse into the implementation in the *UtilAlgos* package if you like. E.g. you will find back the event loop there that you know from Example 3, which is already implemented there for you so that you don't have to programm it over and over again, for each new executable that you plan to write.

The key element that drives the analysis is the BasicMuonAnalyzer class, which is defined in the *interface* directory of the package:

```
class BasicMuonAnalyzer : public edm::BasicAnalyzer {

 public:
  /// default constructor
  BasicMuonAnalyzer(const edm::ParameterSet& cfg, TFileDirectory& fs);
  /// default destructor
  virtual ~BasicMuonAnalyzer(){};
  /// everything that needs to be done before the event loop
  void beginJob(){};
  /// everything that needs to be done after the event loop
  void endJob(){};
  /// everything that needs to be done during the event loop
  void analyze(const edm::EventBase& event);

 private:
```

```
  /// input tag for mouns
  edm::InputTag muons_;
  /// histograms
  std::map<std::string, TH1*> hists_;
};
```

**Note:** Those already familiar with the structure of EDAnalyzer's will immediately recognise the similarities. The class derives from the *BasicAnalyzer* class, which is an abstract base class. It has a *beginJob*, *endJob* and an *analyze* function in analogy to the EDAnalyzer, which is a requirement of the base class. In addition we added a *std__map* for histogramming and an input tag for a muon collection.

You can find the implementation in the BasicMuonAnalyzer.cc file in the *src* directory of the package:

```
/// default constructor
BasicMuonAnalyzer::BasicMuonAnalyzer(const edm::ParameterSet& cfg, TFileDirectory& fs):
  edm::BasicAnalyzer::BasicAnalyzer(cfg, fs),
  muons_(cfg.getParameter<edm::InputTag>("muons"))
{
  hists_["muonPt"  ] = fs.make<TH1F>("muonPt"  , "pt"  ,  100,  0., 300.);
  hists_["muonEta" ] = fs.make<TH1F>("muonEta" , "eta" ,  100, -3.,   3.);
  hists_["muonPhi" ] = fs.make<TH1F>("muonPhi" , "phi" ,  100, -5.,   5.);
  hists_["mumuMass"] = fs.make<TH1F>("mumuMass", "mass",   90, 30., 120.);
}

/// everything that needs to be done during the event loop
void
BasicMuonAnalyzer::analyze(const edm::EventBase& event)
{
  // define what muon you are using; this is necessary as FWLite is not
  // capable of reading edm::Views
  using reco::Muon;

  // Handle to the muon collection
  edm::Handle<std::vector<Muon> > muons;
  event.getByLabel(muons_, muons);

  // loop muon collection and fill histograms
  for(std::vector<Muon>::const_iterator mu1=muons->begin(); mu1!=muons->end(); ++mu1){
    hists_["muonPt" ]->Fill( mu1->pt () );
    hists_["muonEta"]->Fill( mu1->eta() );
    hists_["muonPhi"]->Fill( mu1->phi() );
    if( mu1->pt()>20 && fabs(mu1->eta())<2.1 ){
      for(std::vector<Muon>::const_iterator mu2=muons->begin(); mu2!=muons->end(); ++mu2){
if(mu2>mu1){ // prevent double conting
  if( mu1->charge()*mu2->charge()<0 ){ // check only muon pairs of unequal charge
    if( mu2->pt()>20 && fabs(mu2->eta())<2.1 ){
              "mumuMass"]->Fill( (mu1->p4()+mu2->p4()).mass() );
            }
          }
        }
      }
    }
  }
}
```

**Note:** You find the histogram booking in the constructor and the analysis implementation in the *analyze* function of the class. It is not the topic of this chapter but you can transform this very code into EDAnalyzer by a few lines like this:

```
#include "PhysicsTools/PatExamples/interface/BasicMuonAnalyzer.h"
#include "PhysicsTools/UtilAlgos/interface/EDAnalyzerWrapper.h"

typedef edm::AnalyzerWrapper<BasicMuonAnalyzer> WrappedEDMuonAnalyzer;
```

```
#include "FWCore/Framework/interface/MakerMacros.h"
DEFINE_FWK_MODULE(WrappedEDMuonAnalyzer);
```

You can indeed find such an plugin definition at the end of the modules.cc file in the *plugins* directory of the *PhysicsTool/UtilAlgos* package. Try to run the plugin:

```
addpkg PhysicsTools/UtilAlgos V08-02-09-01
scram b -j 4
cmsRun PhysicsTools/UtilAlgos/test/cmsswWithPythonConfig_cfg.py
```

You will get the exact same result as with the FWLite executable having used the exact same class *BasicMuonAnalyzer*.

**Note:** these lines imply that you have chaecked out the *PhysicsTools/UtilAlgos* package with a tag larger or equa then *V08-02-09-01*

# Appendix: Using DB Information in FWLite

In this appendix we explain how to access data base information in FWLite: we will first need to create an FWLite-readable ROOT file that will "cache" the payloads from the Conditions Database (`CondDB`). As an example we choose B-Tag information. The first part will access the database, and dump the payloads to a ROOT file:

```
cd RecoBTag/PerformanceDB/test
cmsRun testFWLite_Writer_cfg.py
```

It's obvious that for this operation you to be logged in to *lxplus* or somewhere else with apropriate DB connection. You can find the *python* configuration file that we used here. We will go through it step by step. The following lines connect to the database and accesses the appropriate payloads:

```
process.load("CondCore.DBCommon.CondDBCommon_cfi")
process.load ("RecoBTag.PerformanceDB.PoolBTagPerformanceDB100426")
process.load ("RecoBTag.PerformanceDB.BTagPerformanceDB100426")
process.PoolDBESSource.connect = 'frontier://FrontierProd/CMS_COND_31X_PHYSICSTOOLS'
```

The specific records you want to cache will be specified with the "toGet" method:

```
process.PoolDBESSource.toGet = cms.VPSet(
    cms.PSet(
    record = cms.string('PerformanceWPRecord'),
    tag = cms.string('BTagPTRELSSVMwp_v1_offline'),
    label = cms.untracked.string('BTagPTRELSSVMwp_v1_offline')
    ),
                                                    cms.PSet(
    record = cms.string('PerformancePayloadRecord'),
    tag = cms.string('BTagPTRELSSVMtable_v1_offline'),
    label = cms.untracked.string('BTagPTRELSSVMtable_v1_offline')
    ))
```

The actual module that dumps the payloads is specified here:

```
process.myrootwriter = cms.EDAnalyzer("BTagPerformaceRootProducerFromSQLITE",
                                      name = cms.string('PTRELSSVM'),
                                      index = cms.uint32(1001)
                                      )
```

The root file containing the cached payloads will be accessed within FWLite as shown below:

```
TestPerformanceFWLite_ES
```

The source code of this file is located here.

To access the dumped DB information, the syntax is:

```
fwlite::ESHandle< PerformancePayload > plHandle;
  es.get(testRecID).get(plHandle,"PTRELSSVM");
  fwlite::ESHandle< PerformanceWorkingPoint > wpHandle;
  es.get(testRecID).get(wpHandle,"PTRELSSVM");

  if ( plHandle.isValid() && wpHandle.isValid() ) {
    BtagPerformance perf(*plHandle, *wpHandle);
```

From there you can access the payloads as you would in the full framework:

```
// check beff, berr for eta=.6, et=55;
    BinningPointByMap p;

    std::cout <<" test eta=0.6, et=55"<<std::endl;

    p.insert(BinningVariables::JetEta,0.6);
    p.insert(BinningVariables::JetEt,55);
    std::cout <<" nbeff/nberr ?"<<perf.isResultOk(PerformanceResult::BTAGNBEFF,p)<<"/"<<perf.isRe
    std::cout <<" beff/berr ?"<<perf.isResultOk(PerformanceResult::BTAGBEFF,p)<<"/"<<perf.isResul
    std::cout <<" beff/berr ="<<perf.getResult(PerformanceResult::BTAGBEFF,p)<<"/"<<perf.getResul
```

# Review status

Show Hide

| Reviewer/Editor and Date (copy from screen) | Comments |
|---|---|
| RogerWolf - 09 Nov 2010 | Revision for Nov Tutorial |

Responsible: RogerWolf

---

This topic: CMSPublic > WorkBookFWLiteExamples
Topic revision: r44 - 2018-01-23 - ElizaMelo