# Table of Contents

# 9.5 Writing your own framework objects to a file

Complete:

| Newsbox |
| --- |
| Read first WorkBookWriteFrameworkModule |

## Contents



## Introduction

CMSSW includes the nice feature that it is possible to include your own objects and framework in an output file which can then be read in in a private analysis in the framework. The largest advantage of this method is that most of the analysis is done in the framework, which means that it is easy to port analysis tools developed in one analysis to another. Running your analysis mostly in the framework has the additional advantage that you can more easily run using grid and batch tools, thus speeding up the analysis process.

We will only discuss how to use objects that already exist in the framework. It is of course also possible to create new personal object types and book these into your events but this is covered in more advanced tutorials. We will discuss one of many possible analysis techniques.

- We want to save only certain objects in the event and save some additional information. For example: we want to save all standard tracks and their inner and outermost points.

Other possibilities would include the use of the Candidate container classes, or a small analysis that for example would create Z boson or J/Psi candidates from two muons and would save the resonance candidates as LorentzVectors or some kind of Candidate.

## CMSSW releases

This tutorial has been done in `CMSSW_5_3_11`.

# The basic principles and design concepts of EDProducers

A producer is a module in the framework that creates new objects. Besides the usual class constructors and destructor methods a producer has the following methods:

```
  void beginJob( const edm::EventSetup & );
  void produce( edm::Event& , const edm::EventSetup& );
  void endJob();
```

The `beginJob` and `endJob` methods are similar as used in framework analyzers, and can be used to instantiate objects and finish up after running. The actual storing of objects in the event happens in the `produce` method.

The most convenient way to create a producer code skeleton is by using the scripts that are available in the framework. In a set up CMSSW environment you can create a producer by typing:

```
scram p CMSSW CMSSW_5_3_11
cd CMSSW_5_3_11/src
cmsenv
mkdir ProdTutorial
cd ProdTutorial
mkedprod ProducerTest
```

*1. Take a look at the code skeleton created by the mkedprod method. Identify the different methods in the `ProducerTest/src/ProducerTest.cc` file. Things to notice are:*

- In the producer's constructor you define the name and type of object that you will eventually produce;
- In the `produce()` method the objects are created and then saved into the event;
- A framework macro `DEFINE_FWK_MODULE(ProducerTest);` is called to define the producer object as a framework plugin.

Note that you will *always* have to modify the `CMS.BuildFile` of your producer if you want to save objects, to make sure that the appropriate libraries are known to the framework.

# A very simple producer that saves existing tracks and two space points

The following example loops over tracks and saves the inner and outer point of the track, together with the existing track. The point objects will be pointers to type `math::XYZPointD`. This is also the type returned by the track's `Track::outerPosition()` and `Track::vertex()` methods.

## Creating the code

### Create the producer skeleton

The first step would be to create a producer skeleton:

```
cd $CMSSW_BASE/src
cd ProdTutorial
mkedprod TrackAndPointsProducer
```

Note: Doing `cd $CMSSW_BASE/src` above brings you back to the `CMSSW_5_3_11/src` directory.

Open the source file `TrackAndPointsProducer/src/TrackAndPointsProducer.cc` in your favorite editor and start adding the following code:

### Include the appropriate header files

You will use objects of type `Point` and `Track`, and will use the standard vector classes and the standard library, so include these in your list of header files:

```
#include <vector>
#include <iostream>
#include "DataFormats/Math/interface/Point3D.h"
#include "DataFormats/TrackReco/interface/Track.h"
#include "DataFormats/TrackReco/interface/TrackFwd.h"
```

### Modify the class definition and add members

In the class definition you will need to define the labels of the input objects (`Tracks` in this case). This way you can use your configuration file to easily switch between different track algorithms without recompiling. So add the input information to your class definition. You should also make sure that your class recognizes the containers that you will eventually book into the event, we use `Point` objects in a container class `PointCollection`. After adding these your class definition should look like this:

```
class TrackAndPointsProducer : public edm::EDProducer {
   public:
      explicit TrackAndPointsProducer(const edm::ParameterSet&);
      ~TrackAndPointsProducer();

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

      // ----------member data ---------------------------

    edm::InputTag src_;
    typedef math::XYZPointD Point;
    typedef std::vector<Point> PointCollection;

};
```

### Modify the class constructor

When the class constructor is called, the framework should be instructed that the `TracksAndPointsProducer` class will add something to the file. This is done by calling the `produces < CollectionName > (label)` method. Also this is the place to read in information from the config file. Your constructor should look like this:

```
TrackAndPointsProducer::TrackAndPointsProducer(const edm::ParameterSet& iConfig)
{
  src_  = iConfig.getParameter<edm::InputTag>( "src" );
  produces<PointCollection>( "innerPoint" ).setBranchAlias( "innerPoints");
  produces<PointCollection>( "outerPoint" ).setBranchAlias( "outerPoints");
}
```

### Modify the `produce()` method to book your objects

In the `produce` method you will have to create the appropriate vectors, fill them and put them in the event:

```
void TrackAndPointsProducer::produce(edm::Event& iEvent, const edm::EventSetup& iSetup)
{
   using namespace edm;
   using namespace reco;
   using namespace std;
   // retrieve the tracks
   Handle<TrackCollection> tracks;
   iEvent.getByLabel( src_, tracks );
   // create the vectors. Use auto_ptr, as these pointers will automatically
   // delete when they go out of scope, a very efficient way to reduce memory leaks.
   auto_ptr<PointCollection> innerPoints( new PointCollection );
   auto_ptr<PointCollection> outerPoints( new PointCollection );
   // and already reserve some space for the new data, to control the size
   // of your executible's memory use.

   const int size = tracks->size();
   innerPoints->reserve( size );
   outerPoints->reserve( size );
   // loop over the tracks:
   for( TrackCollection::const_iterator track = tracks->begin();
        track != tracks->end(); ++ track ) {
     // fill the points in the vectors
     innerPoints->push_back( track->innerPosition() );
     outerPoints->push_back( track->outerPosition() );
   }
   // and save the vectors
   iEvent.put( innerPoints, "innerPoint" );
   iEvent.put( outerPoints, "outerPoint" );

}
```

The complete module is here: TrackAndPointsProducer.cc

### Edit the `BuildFile`

Make sure that before you compile you edit the `BuildFile` so it includes the object libraries you will use in your analysis. The following BuildFile includes the Track and Point class libraries:

```
<use name=FWCore/Framework>
<use name=FWCore/PluginManager>
<use name=DataFormats/TrackReco>
<use name=DataFormats/Math>
<flags EDM_PLUGIN=1>
<export>
   <lib name= 1 />
</export>
```

### And... Compile!

You are now ready to compile:

```
cd $CMSSW_BASE/src
cd ProdTutorial
scram b
```

If everything goes to plan you shouldn't have any compilation errors and you should see something similar to the following printout:

Show result... Hide result...

```
[lxplus429] /afs/cern.ch/user/x/xuchen/workbook/CMSSW_5_3_11/src/ProdTutorial > scram b
Reading cached build data
>> Local Products Rules ..... started
>> Local Products Rules ..... done
>> Entering Package ProdTutorial/ProducerTest
>> Creating project symlinks
>> Leaving Package ProdTutorial/ProducerTest
>> Package ProdTutorial/ProducerTest built
>> Entering Package ProdTutorial/TrackAndPointsProducer
>> Leaving Package ProdTutorial/TrackAndPointsProducer
>> Package ProdTutorial/TrackAndPointsProducer built
>> Subsystem ProdTutorial built
>> Local Products Rules ..... started
>> Local Products Rules ..... done
gmake[1]: Entering directory `/afs/cern.ch/user/x/xuchen/workbook/CMSSW_5_3_11'
>> Creating project symlinks
>> Done python_symlink
>> Compiling python modules python
>> Compiling python modules src/ProdTutorial/ProducerTest/python
>> Compiling python modules src/ProdTutorial/TrackAndPointsProducer/python
>> All python modules compiled
>> Pluging of all type refreshed.
gmake[1]: Leaving directory `/afs/cern.ch/user/x/xuchen/workbook/CMSSW_5_3_11'
```

## Running the producer

If you do not have any compilation errors you should now have a working producer module. The next step would be running the module. This means creating a config file, which should contain the following:

- a definition of your producer module. Note that you can easily define more than one if you want to compare objects/algorithms. Note that only at this point you will have to choose *which* track objects to use;
- an input file of some sorts. You can explore use the DBS/DLS database discovery tool to find data files you want to look at, or create your own;
- an output file that contains the information you need and removes everything you are not interested in.

Actually this file ( which you will modify to have the contents below) already exits in the `$CMSSW_BASE/src/ProdTutorial/TrackAndPointsProducer/` directory. It is called `trackandpointsproducer_cfg.py`. Either replace the contents this file with the lines below, otherwise this file in its entirety is here called trackandpointsproducer_cfg.py.txt

```
import FWCore.ParameterSet.Config as cms

process = cms.Process("OWNPARTICLES")

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

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(
       'file:/afs/cern.ch/cms/Tutorials/TWIKI_DATA/CMSDataAnaSch_RelValZMM536.root'
```

```
    )
)

#from ProdTutorial.TrackAndPointsProducer.trackandpointsproducer_cfi import *
process.MuonTrackPoints = cms.EDProducer('TrackAndPointsProducer'
        ,src    =cms.InputTag('globalMuons')

)

process.TrackTrackPoints = cms.EDProducer('TrackAndPointsProducer'
        ,src    =cms.InputTag('generalTracks')
)

process.out = cms.OutputModule("PoolOutputModule",
    fileName = cms.untracked.string('myOutputFile.root')
    ,outputCommands = cms.untracked.vstring('drop *',
      "keep *_generalTracks_*_*",
      "keep *_globalMuons_*_*",
       "keep *_MuonTrackPoints_*_*",
      "keep *_TrackTrackPoints_*_*")

)


process.p = cms.Path(process.MuonTrackPoints*process.TrackTrackPoints)

process.e = cms.EndPath(process.out)
```

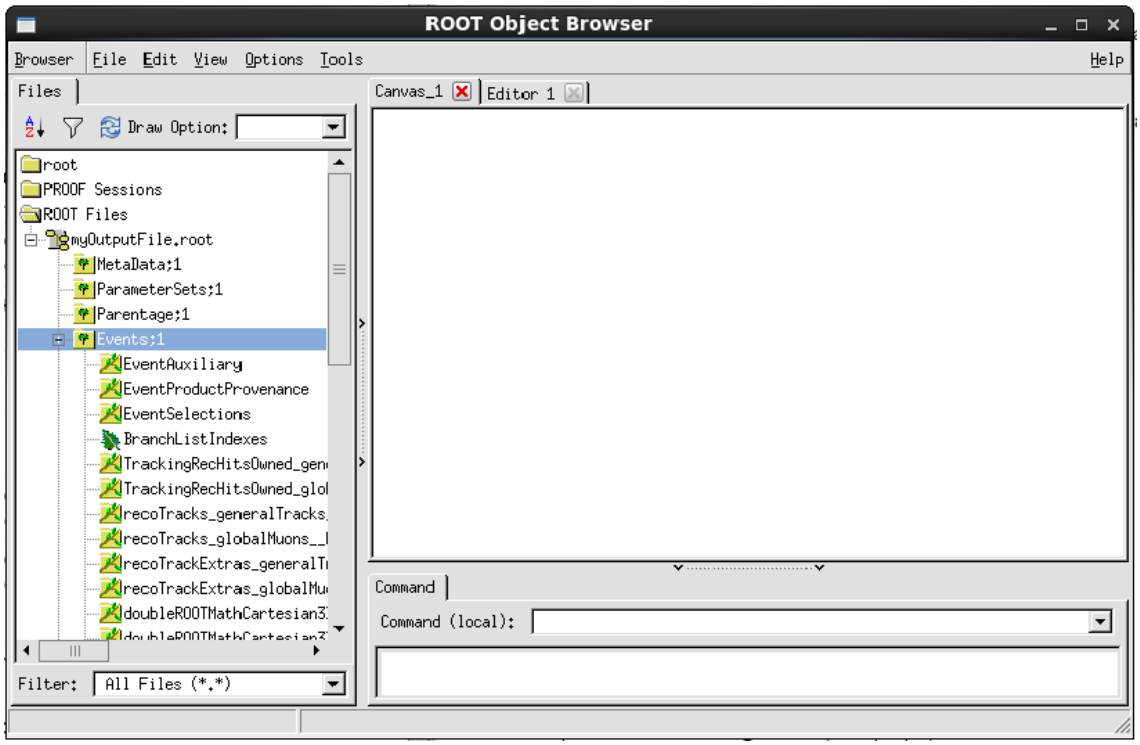
To run the module do the following:

```
cmsRun trackandpointsproducer_cfg.py
```

This creates an output root file called `myOutputFile.root`

## Look at the output

The output file `myOutputFile.root` can now be viewed in bare root or analyzed further in the framework. Here you see the newly added objects:




# Example of a Simple Producer ()

In many workflows, you might run a filter that drops some portion of events, but you want to keep track of how many events were present before the filter was run. This information can be provided by a simple tool called EventCountProducer.

In your python configuration, you simply create an instance and then include it in your path at the point where you want to count events. For example, if you include a filter to select events with muons, you could create two producers, one to count events before the filter and one to count the number of events that pass the filter:

```
process.nEventsTotal = cms.EDProducer("EventCountProducer")
process.nEventsFiltered = cms.EDProducer("EventCountProducer")

process.p = cms.Path(
    process.nEventsTotal *
    process.muonFilter *
    process.nEventsFiltered
)
```

The EventCountProducer stores its product in the luminosity block and is able to merge event counts from multiple files. So, if you were creating patTuples in the previous step, and were now running an analyzer on several patTuple files, you could access the event counts in the `endLuminosityBlock` method of your analyzer:

```
void MyAnalyzer::endLuminosityBlock(const edm::LuminosityBlock & lumi, const EventSetup & setup)
// Total number of events is the sum of the events in each of these luminosity blocks
Handle
```

# Summary and Conclusion

This tutorial shows several ways to use producers for analysis purposes. Using producers for analysis is a very efficient way to use the framework. It reduces the amount of code duplication that usually happens in private

`ntuple` based analyses and has the additional advantage that all debugging facilities in the framework are available, which makes debugging easier than in root macros.

## Review Status

| Editor/Review and date | Comments |
|---|---|
| FreyaBlekman - 23 Feb 2007 | Original Author |
| JennyWilliams - 05 Mar 2007 | Moved tutorial into WorkBook, moved contents of this WB page to SWGuideEDProducer |
| ChristopherJones - 28 Jan 2008 | Updated to work with releases equal to or greater than 1_5_X |
| AndriusJuodagalvis - 2009-09-05 | Added info how to access the produced branches |
| XuanChen - 10 Jun 2014 | Changed CMSSW release to CMSSW_5_3_11, replaced files and outputs |

Responsible: Sudhir Malik
Last reviewed by: Reviewer. Sudhir Malik- 26 November 2009.

- Set ALLOWTOPICCHANGE = SudhirMalik, JohnStupak , XuanChen

---

This topic: CMSPublic > WorkBookEDMTutorialProducer
Topic revision: r43 - 2014-06-10 - XuanChen