Table of Contents

and Configurables

An FAQ and reminder about Configurables, specifically focussing on TupleTools.

Prerequisites:

The gaudi framework handles our job configuration in LHCb. Our software uses and abuses it fully, so mostly every question you can think of can be answered by "This is how it is in Gaudi, how Gaudi works." So, in order to understand how to configure things, you must first understand Gaudi, and have done the basic tutorials assosciated to this.

You will need to be familiar both with C++ and python. After that you will need to do AT LEAST the following tutorials.

- Introduction to LHCb software - part 1 - Software organisation, environment setup and CMT
- Introduction to LHCb software - part 2 - Algorithms, Printing, Job Options.
- Ganga Tutorial - Getting started with Ganga. Ganga Tutorial for LHCb, last updated 2009-03-16, using DaVinci v22r1
- DaVinciTutorial

The configuration of TupleTools is neatly covered in the DaVinciTutorial, as is how to configure many other things, so almost every other question you can think of has an example there.

If you really have really done that, and would like some more details, please read this page, preferably **before** sending an email to the list.

Introduction:

Gaudi basics:

Gaudi has tools and algorithms that mostly run in a big C++ program. Which tools and algorithms are run is determined at run-time by some clever options which are usually written by the python program `gaudirun.py`. Gaudi will do the following to the options file that you give it.

- parse - use python to interpret a load of options
- construct - setup the basic C++ classes
- configure - pass the options to them
- initialize, execute, finalize

Only the first bit of this is really the realm of your configurables, after parsing they don't mean anything or do anything any more, they are only used to define what will be run in the job, they are not doing any of the work themselves. What has been done to make the configurables simpler however can sometimes be confusing, so it is necessary to understand the differences so that you don't go insane.

C++ basics

Gaudi algorithms and tools have three parts:

- A virtual base class
- A concrete implimented class
- An instance in the C++

Each of these parts has a de-facto name. Gaudi objects can own other gaudi objects, and which **type** of object they can own is always hard coded. The type of object is the same as the virtual base class, like IParticleTupleTool. The virtual basclass defines what operations are guaranteed in each derived implimentation. IParticleTupleTool is the C++ class name for the virtual base class, that we call an "interface". It is this interface which is the minimum which must be hard coded in the C++, everything else can be, and sometimes even is, softcoded. The GaudiAlgorithm and GaudiTool interfaces are pretty much the two most basic interfaces which form the framework parts you will interact with the most, and is how the framework talks to itself and runs your code.

Each interface may have many concrete implimentations, which are derived classes, guaranteed to have all the methods from the interface. Which implimentation you pick, can be done at run-time inside gaudi, by giving the implimentation class name, like `"TupleToolKinematic"`.

As the above two steps only define a class, you are at liberty to have multiple instantiated objects of the same class. Each instance has a unique name, like `"SomeInstanceNameIWantToUse"`. This can be hard coded, and can be, and sometimes even is, soft coded.

Then the object itself is fully determined by `"TupleToolKinematic/SomeInstanceNameIWantToUse"`. What comes before the **"/"** is the concrete implementation, what comes after the **"/"** is the instance name. If there is no **"/"** the instance name is the same as the class name.

Gaudi algs/tools are configured/controlled by Properties. `declareProperty()` statements in the constructor determine what properties each algorithm and tool has available. For example the **Verbose** property is defined for all TupleTools, to add extra normaly more verbose entries in your tuple.

Configurable Basics:

A python configurable is an automatically generated piece of python code which can set the properties of the underlying C++.

The key point to remember is that objects you create in python **are not the same** as the object which will be created in the C++.

The python bit tells only how the C++ object properties should be set, that is all.

The python bit knows only about the Properties of the C++, nothing about the internal workings/code, nothing about any hard-coded values can be changed, only properties of Gaudi objects like tools and algorithms.

Automatic python configurables can have three bits:

- The python configurable class which was generated
- A user-defined name which uniquely defines that configurable wherever it is used
- An instance and the local variable where you store this instance

The python class has the same name as the C++ class (the concrete implimentation). The user-defined name tells the configurable which C++ instance you are trying to configure, so i must be the same as the instance name used in the C++. This is all done in python, so you can assign the configurable to whatever local variable you want. Essentially if there was really an existing C++ object which was uniquely defined in gaudi by `"TupleToolKinematic/SomeInstanceNameIWantToUse"` it would be configured by `TupleToolKinematic(name='SomeInstanceNameIWantToUse')`

Algorithms

When you do

```
fish=MyAlg("Fish")
fish.lemon=True
```

for example, you are creating an instance of the **configurable** for MyAlg.

Here MyAlg is the python class, which almost always has the same name as the C++ class, **remember it is not the same class, it just has the same name,** fish is the variable where you hold this instance, and "Fish" is the instance name for the C++ object you want configured. So this would configure some C++ object which was uniquely defined in Gaudi by `"MyAlg/Fish"`.

For the C++ object to be created, the ApplicationMgr must know about it by the time the C++ is run. This property of the ApplicationMgr is likewise controlled by the ApplicationMgr() configurable. So, you pass the configurable for MyAlg, to the configurable for ApplicationMgr.

So, eventually in the C++, the ApplicationMgr class property "TopAlgs", which is a list of strings, gains the entry `"MyAlg/Fish"`. This is usually done behind the scenes for you, and then the appliction manager itself will take care of instantiating the class in the C++ once the C++ part of your job starts.

Tools:

For tools it is more complicated. :S Tools do not really exist by themselves, they are used by/owned by algorithms. The name of a tool may be hard-coded in the C++, or it may be a property of the algorithm which uses it.

Because the configurable for the algorithm knows nothing about what's in the C++, you need to at least tell the configurable for the alg that this alg has a tool you want to configure. To do this you add a configurable of the tool, to the configurable of the algorithm:

```
fish.addTool(MyTool())
```

So, most of the time you need to create a configurable to configure the tool with that specific name, play with the configurable. But you also need to make sure you don't mess up other configurations of the same tool.

You can try and configure the tool globally. This should only be done if you really want to share the configuration between multiple instances of the tool.

```
mT=MyTool() #configurable for any/all MyTool("MyTool") C++ objects.
mT.salt=True

mT=MyTool("Chips") #configurable for any/all MyTool("Chips") C++ objects.
mT.salt=True
```

add a global configurable to your Alg:

```
fish.addTool(mT)

fish.addTool(MyTool())
fish.MyTool.salt=True
```

Most of the time you want to be safe, and configure a tool only for your algorithm. In this case you should configure it locally:

```
fish.addTool(MyTool, name="MyTool") #configure the "MyTool" instance in this alg
fish.MyTool.salt=True

fish.addTool(MyTool, name="Chips") #configure the "Chips" instance in this alg
fish.Chips.salt=True
```

:

You decide what TupleTools are going to be in your DecayTreeTuple. DecayTreeTuple is highly configurable, it only demands that the tools you add have a `fill()` method, so they all inherit from a given interface. None of the names are hard coded. This is done through an option of DecayTreeTuple, which when in "initialize" the C++ creates the instances of all the tools you asked for.

Simplest configuration:

From `DecayTreeTuple v3r11p1` and upwards there are wrapper functions (decorators) attached to `DecayTreeTuple`, `EventTuple`, `MCDecayTreeTuple`, `TupleToolDecay` and `TupleToolMCTruth` which simplify greatly the adding and configuring of tools.

At the basic level they just wrap the gaudi addTool methods so that they are less confusing and take less typing, they are not standard Gaudi components, so you need to import DecayTreeTuple a specific way to get at them `from DecayTreeTuple.Configuration import *` NOT `from Configurables import DecayTreeTuple`.

They still use the underlying Gaudi methods, but contain all the code of the most common actions, reducing what you need to do yourself.

```
from DecayTreeTuple.Configuration import *
nt=DecayTreeTuple("MyTuple")
nt.ToolList=[]
ttk=nt.addTupleTool("TupleToolKinematic/TTK")
tistos=nt.addTupleTool("TupleToolTISTOS")
```

The configurable DecayTreeTuple() is for the C++ instance "MyTuple". We start it with an empty tool list, because we don't want the default tools this time. The configurable will pass the ToolList to the C++ property once it has been fully configured.

By using the `addTupleTool` methods the configurable for "MyTuple" automatically imports the correct module and attaches the right private TupleTools to the ToolList for you. Here you have asked for an instance of TupleToolKinematic called "TTK" and an instance of TupleToolTISTOS called "TupleToolTISTOS". We also get a handle to the cofigurables for those tools, `ttk` and `tistos`. You need these configurables to configure the tools, which will eventually be instantiated in the C++.

```
ttk.Verbose=True
tistos.Verbose=True
tistos.TriggerList=['Hlt1 ..... ' ,
                                                    .....
]
```

You can also get access to the TupleTool configurables through the more regular way, in case you lose the references somewhere.

```
ttk=nt.TTK
tistos=nt.TupleToolTISTOS
```

In the most complicated configurations, you may want to configure a tool of another tool of another tool, etc...:

```
truth=nt.addTupleTool("TupleToolMCTruth")
truth.addTool(DaVinciSmartAssociator, name="DaVinciSmartAssociator")
truth.DaVinciSmartAssociator.addTool(BackgroundCategory, name="BackgroundCategory")
truth.DaVinciSmartAssociator.BackgroundCategory.Inclusive=True
```

Standard configuration:

This is much more complicated version of the above standard method, but included here for completeness.

```
from Configurables import DecayTreeTuple
nt=DecayTreeTuple("MyTuple")
nt.ToolList=[
                    "TupleToolKinematic/TTK",
                    "TupleToolTISTOS"
]

nt.TTK.Verbose=True

nt.addTool(TupleToolTISTOS, name="TupleToolTISTOS")
nt.TupleToolTISTOS.Verbose=True
nt.TupleToolTISTOS.TriggerList=['Hlt1 ..... ' ,
                                                .....
]

nt.addTool(TupleToolMCTruth, name="TupleToolMCTruth")
nt.TupleToolMCTruth.addTool(DaVinciSmartAssociator, name="DaVinciSmartAssociator")
nt.TupleToolMCTruth.DaVinciSmartAssociator.addTool(BackgroundCategory, name="BackgroundCategory")
nt.TupleToolMCTruth.DaVinciSmartAssociator.BackgroundCategory.Inclusive=True
```

Wrapper functions (decorators)

There are many uses for decorators. In the table below, the plain Gaudi-style methods and wrapper methods are compared. In all cases the wrappers are easier to understand and require much less typing.

Purpose	.	Standard Method	-	
Import the module	.	`from Configurables import DecayTreeTuple`	>	`from DecayTreeTuple.Confi`
Default tool don't configure	.	`ntuple.ToolList+=['TupleToolSomething']`	=	`ntuple.ToolList+=['TupleT`
Named tool don't configure	.	`ntuple.ToolList+=['TupleToolSomething/instance']`	=	`ntuple.ToolList+=['TupleT`
Default tool do configure	.	`ntuple.ToolList+=['TupleToolSomething']` from Configurables import TupleToolSomething ntuple.addTool(TupleToolSomething) ntuple.TupleToolSomething...	>	`myTool=ntuple.addTupleToo` myTool...
Named tool do configure	.	`ntuple.ToolList+=['TupleToolSomething/instance']` from Configurables import TupleToolSomething ntuple.addTool(TupleToolSomething, name='instance')	>	`myTool=ntuple.addTupleToo` myTool...

		ntuple.instance...		
Local shared tool do configure	.	`ntuple.Branch1.ToolList+=['TupleToolSomething/instance']` ntuple.Branch2.ToolList+=['TupleToolSomething/instance'] from Configurables import TupleToolSomething myTool=TupleToolSomething('instance') myTool... ntuple.Branch1.addTool(myTool) ntuple.Branch2.addTool(myTool)	>	`myTool=ntuple.Branch1.add` myTool... ntuple.Branch2.addTupleTool(
Global shared tool do configure	.	`ntuple.Branch1.ToolList+=['TupleToolSomething/instance']` ntuple.Branch2.ToolList+=['TupleToolSomething/instance'] from Configurables import TupleToolSomething myTool=TupleToolSomething('instance') myTool... ntuple.Branch1.addTool(myTool) ntuple.Branch2.addTool(myTool)	>	`from Configurables import` myTool=TupleToolSomething myTool... ntuple.Branch1.addTupleTool(ntuple.Branch2.addTupleTool(
Add three branches	.	`ntuple.Branches={'1':'decay1', '2' : 'decay2', '3':'decay3'}` from Configurables import TupleToolDecay ntuple.addTool(TupleToolDecay, name='1') ntuple.addTool(TupleToolDecay, name='3') ntuple.addTool(TupleToolDecay, name='3')	>	`ntuple.addBranches({'1':'`

Since shared tools on branches can still be confusing, and are prone to errors in case somebody else tries to make a tool of the same name somewhere else. It is mostly better to define a function in python which adds and configures all instances of the tools so you can share the configuration without sharing the tool configurable.

```
def addMyTool(branch):
  atool=branch.addTupleTool('TupleToolSomething/TTSinstance')
  atool.MyOption1=1
  atool.MyOption2=2

for branch in [ntuple.branch1, ntuple.branch2]:
  addMyTool(branch)
```

Since these decorators just wrap the underlying gaudi functions, you can mix-and-match them if you insist, or simply take advantage of all functionality.

```
ntuple.branch1.TTSinstance.SomeOption=True #i.e. if you lose the reference to the tool, you can g
```

Dear Configurable:

We can understand configurables as a way of communicating with the underlying C++. The key actions can then be summarized as if it were a letter we were writing to the C++ through the "python configurable postal service"

Neatly worded by Phillip Hunt and Matt Charles:

```
Tuple.ToolList += [ "TupleToolTagging/TagTool" ]
```

In the python, not the C++:

- Dear Configurable, please add the line "TupleToolTagging/TagTool" to the property ToolList of the Tuple.

In the C++, not the python:

- Dear C++, please instantiate and use a copy of TupleToolTagging in my ntuple, I want this instance to be called "TagTool".

```
Tuple.addTool(TupleToolTagging, name="TagTool")
```

Dear Configurable: please note that, although you didn't know about it before, the C++ object you will configure will in fact have a TupleToolTagging tool assosciated with it. This tool is called "TagTool" in the C++. Please create a new instance of a configurable TupleToolTagging for "TagTool", and add it as a member of yours, so that I can change its options also.

-- RobLambert - 23-Feb-2010

This topic: LHCb > TupleToolsAndConfigurables
Topic revision: r9 - 2014-02-17 - RobLambert