Athena tutorial

Johannes Junggeburth, Philipp Gadow

(based on previous slides by K. Köneke, T. Maier, L. Heinrich)

ATLAS-D 2019 meeting, Munich

High Energy Physics is a data-intensive science.
We need software to process and analyse the data.

High Energy Physics is a collaborative science.
We need software that makes it easy to work together.

we therefore use software frameworks to integrate the code we write into a common structure.

The one used most in ATLAS is

Athena

What's a Software Framework?

Software Frameworks provide scaffolding and common infrastructure for developers to work with.

Invert control: "Don't call us, we'll call you.":

The overall program flow is controlled **by the framework** and not by the developer code.

Developers write components:

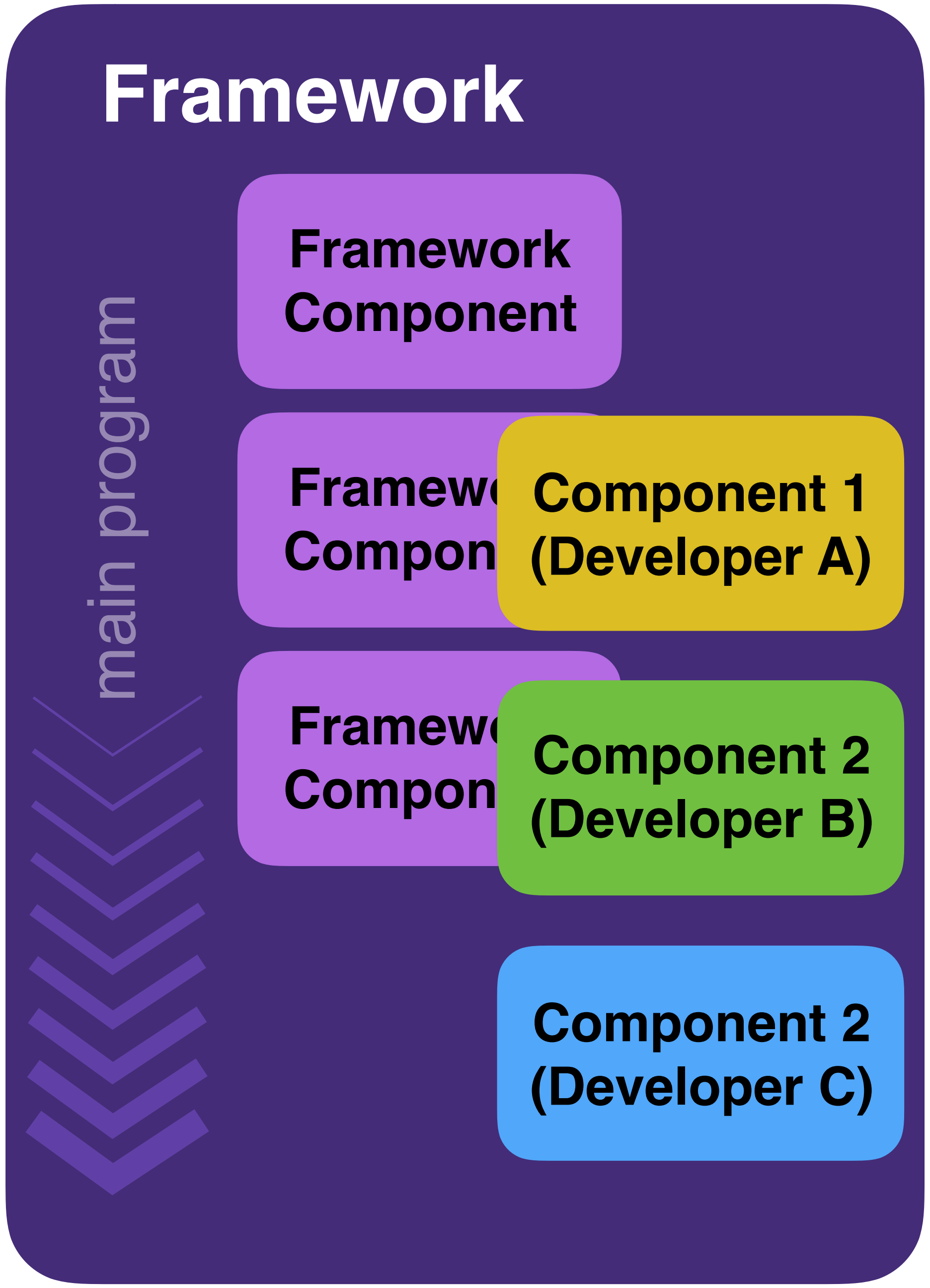
- **use common infrastructure provided by the framework**
- **implement well-defined interfaces**
- **get registered with the framework**

Frameworks then call components during the execution of the main program.

What’s a Software Framework?

Frameworks provide helpul tools that you will need and that are hard to get right if you wrote them from scratch

- Main Program Flow
- input/output file handling
- Logging
- Job Configuration
- Scheduling
- Data Access between sub-routines
- Monitoring
- etc..



Athena

Athena

Gaudi

Athena is the framework in ATLAS used for

- online data taking
- event generation and simulation
- offline reconstructions and derivation production
- analysis

based on the **Gaudi Framework —** joint project with LHCb, FCC

- main software written in C++
- Python is used as a configuration language

The main program flow is a loop over ATLAS collision events, where **for each event** a set of **algorithms and tools** is used to analyse them and extract new data.

athena

The main entrypoint into Athena is the command line tool `athena[.py]`
Main Usage:

```
athena joboptions.py
```

which reads a python configuration ("joboptions file") and starts the event loop.

```
$> touch joboptions.py
$> athena joboptions.py --evtMax=10 --filesInput data.pool.root.1
...
===========================================================================================================
                                                    Welcome to ApplicationMgr (GaudiCoreSvc v4r1)
                                           running on lxplus048.cern.ch on Mon Sep  3 13:44:19 2018
===========================================================================================================
...
AthenaEventLoopMgr   INFO Initializing AthenaEventLoopMgr - package version AthenaServices-00-00-00
AthMasterSeq         INFO Member list: AthSequencer/AthAlgSeq, AthSequencer/AthOutSeq, AthSequencer/AthRe
...
AthenaEventLoopMgr   INFO   ===>>>  start of run 1    <<<===
AthenaEventLoopMgr   INFO   ===>>>  start processing event #1, run #1 0 events processed so far  <<<===
```

As always you can type `athena --help` to learn about the usage.

Getting Athena

Athena is distributed as part of the official ATLAS software releases. There are various flavours of releases tailored for online, simulation, reconstruction and analysis. Click here for an overview.

On machines with ATLASLocalRootBase installed (needs /cvmfs) you can do:

```
setupATLAS
asetup AthAnalysis,21.2.88 # [or 'latest']
athena --help
```

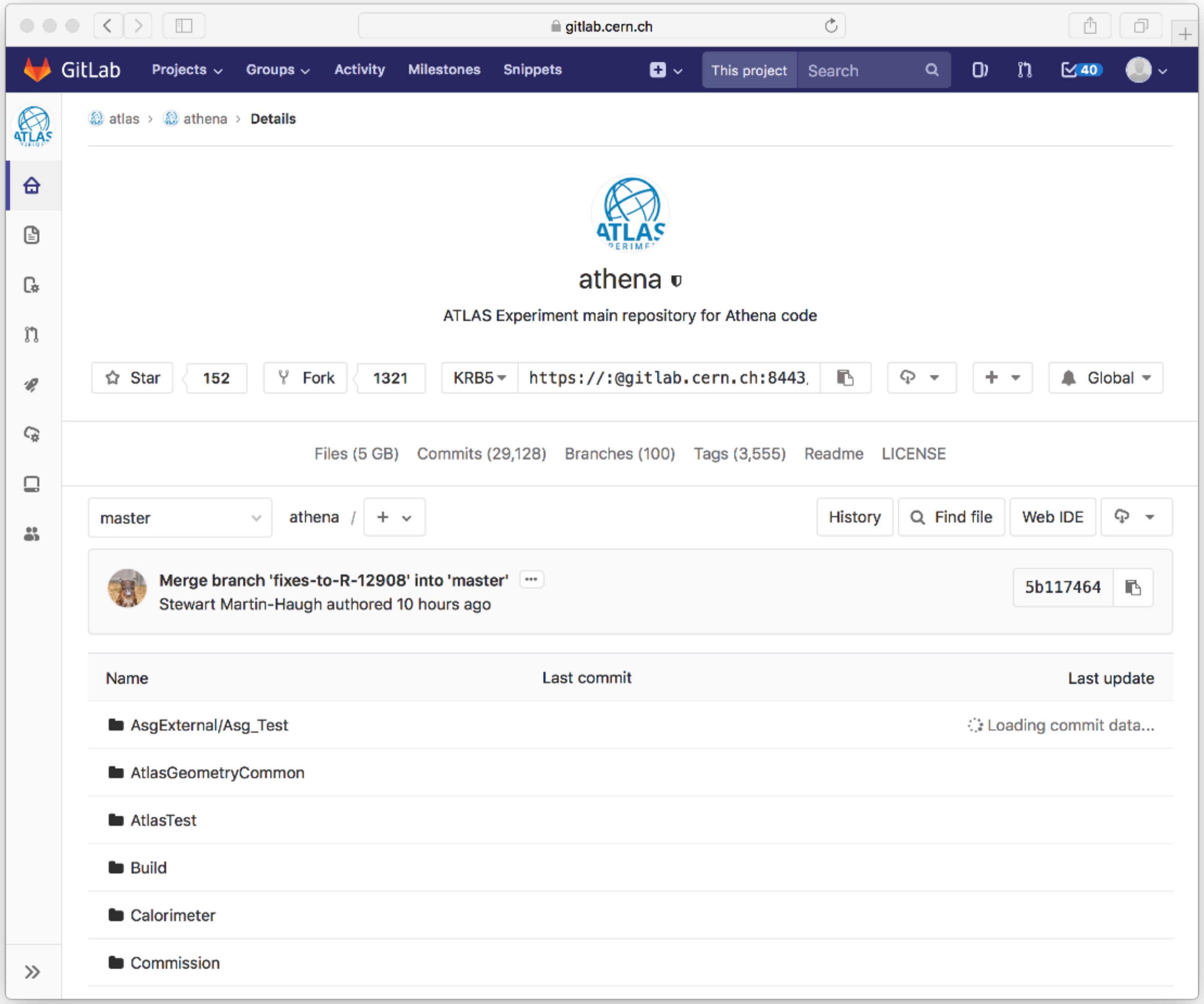
On your laptop (without /cvmfs acces) you can also use official Docker Images (only for analysis-flavored athena)

```
docker pull atlas/athanalysis:21.2.88
docker run --rm -it atlas/athanalysis:21.2.88
source ~/release_setup.sh
```

The Code

Athena is being developed by all of us in the open-source code-base at

https://gitlab.cern.ch/atlas/athena/tree/21.2



Before we proceed…

Athena can seem complex and confusing at first. One does not need to learn all of the pieces — especially not the framework internals.
We aim to give you a birds-eye overview for your orientation.

Many people first encounter it when working e.g. in CP groups in a reconstruction setting, arguably the most complicated usage of Athena.

For analysis you'll mostly be writing algorithms and the framework can handle the rest for you.

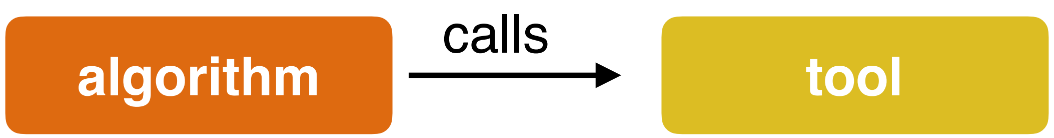
Make sure to follow the hands-on tutorial where we start from scratch and add complexity slowly.

Athena Concepts Overview

Algorithm:

Code that Athena **calls once for each event**. In an algorithm you implement the bulk of your analysis job.

Tool:



To achieve its goal within an algorithms, it can call Tools that can **provide more information** about the event or can **manipulate objects** in the event. As a user you mostly use tools provided by others. If you are part of a CP group you develop Tools to be used by the collaboration.

Services:

Global objects with (almost always) one instance that provide infrastructure like logging (`MessageSvc`) or output (histogram / tree / etc.) handling (`THistSvc`). “Always present” — can be called from anywhere.

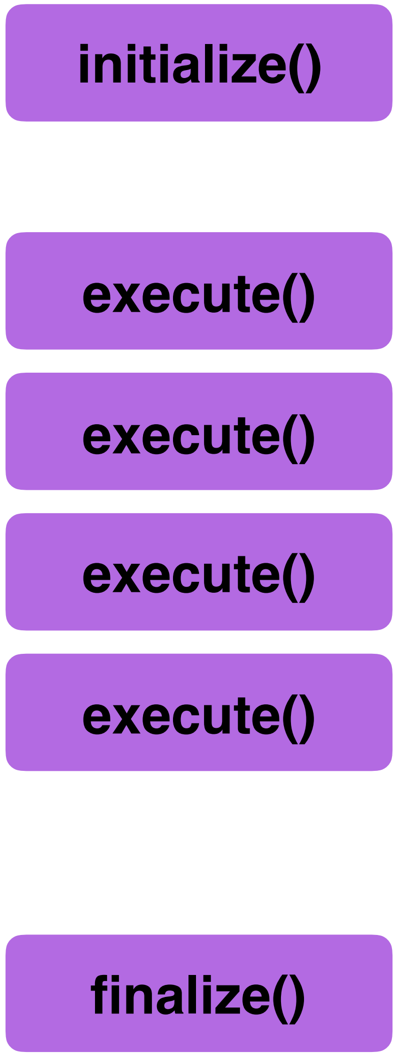
Algorithms

Algorithms are called **once** per event. Their job is to process the data in the event and

- add new event data (e.g. candidate decay pairs)
- modify event data (e.g. calibrate objects)
- summarise data (e.g. fill Histograms, Trees)

Each algorithm must implement three methods

- `initialize()`
  one-time setup. Prepare tools, book histograms, etc...
- `execute()`
  bulk of the algorithm. What to do for each event.
- `finalize()`
  any clean-up operations.



Algorithms

Your algorithm has to inherit from a **Algorithm Base Class**

A number of options available

- `AthAnalysisAlgorithm` — gives access to Metadata Store (see later)
- `AthFilterAlgorithm` — algorithm for event selection
- `AthHistogramAlgorithm` — algorithm for easy histogramming

All methods to be implemented return `StatusCodes`

`StatusCodes` are "smart return codes" that keep track whether the caller of a function has checked the code — more robust code

- **`StatusCode::SUCCESS`**
- **`StatusCode::FAILURE`**
- **`StatusCode::RECOVERABLE`**

You should ALWAYS check the status code using the ATH_CHECK macros!

Tools

Tools are smaller components that manipulate smaller objects present within the event.

Algorithms can use one or more tools to accomplish their task

Example Tools

- TrigDecisionTool — access to Trigger data
  ```
  m_tdt->isPassed("HLT_e.*");
  ```

- JetCleaningTool — select good jets
  ```
  m_jetCleaningTool->keep( *jet );
  ```

- MuonCalibSmearTool – calibrate muon momentum
  ```
  m_muCalibSmearTool->applyCorrection(*muon)
  ```

Tools

Tools have **`initialize()`** `and` **`finalize()`** methods (but not **`execute()`**)

Each tool is defined via an interface defined in a class `ISomeTool`

Tool users **use the only the interface class.**

In an algorithm, add a **ToolHandle** as a data member

```
private:
  ToolHandle<ISomeInterface> m_myTool
```

Tool developers **implement the interface** via some class `SomeTool`

Tools

Tools can be "**public**" or "**private**", which controls the ownership and scope of the tool

A **private tool** is only used by a single algorithm. The instance of the algorithm is not accessible from outside.

A **public tool** is shared among a set of different algorithms. It is owned by the **ToolSvc.** The instances are available job-wide to all algorithms.

Tools

To use a tool in an algorithm, create a **ToolHandle.**

1. add a ToolHandle data member with the right interface class

```
private:
  ToolHandle<ISomeInterface> m_myTool;
```

2. initialise the tool in the constructor this selects the instance

```
m_jetCleaningTool("JetCleaningTool/JetCleaningTool",this)
```

private tool

```
m_jetCleaningTool("JetCleaningTool/JetCleaningTool")
```

public tool

Tools

To use a tool in an algorithm, create a **ToolHandle.**

3. in your algorithms **`initialize()`** "retrieve the tool"

```
StatusCode MyAlg::initialize(){
  ATH_CHECK(m_jetCleaningTool.retrieve());
}
```

grabs the right instance and after this `m_myTool->method()` will work as expected.

4. use the tool in **`execute()`**

```
StatusCode MyAlg::execute(){
...
bool keep = m_jetCleaningTool->keep(*jet);
...

}
```

StoreGate — runtime data access

With algorithms and tools you can manipulate data..
…but how do you get data?

reading data: retrieve data from the file or create by other algorithms
writing data: write new objects for use by other algorithms (e.g. Calibrated muons objects)

Athena provides `StoreGate` data stores in which algorithms can place data objects (these are **Services)**

- **`evtStore()`** event-by-event data (the main store)
- **`detStore()`** detector information
- **`inputMetaStore()`** metadata information (e.g. Cutflow data)

Data is stored under **Keys** that are unique for a given **Type**

i.e. "`Type#Key`" uniquely identifies a datum, e.g.
`xAOD::MuonContainer#Muons`

retrieve data by passing a pointer of the desired type and the key

```
const xAOD::MuonContainer* muons = 0;
ATH_CHECK( evtStore()->retrieve( muons, "Muons" ) );
```

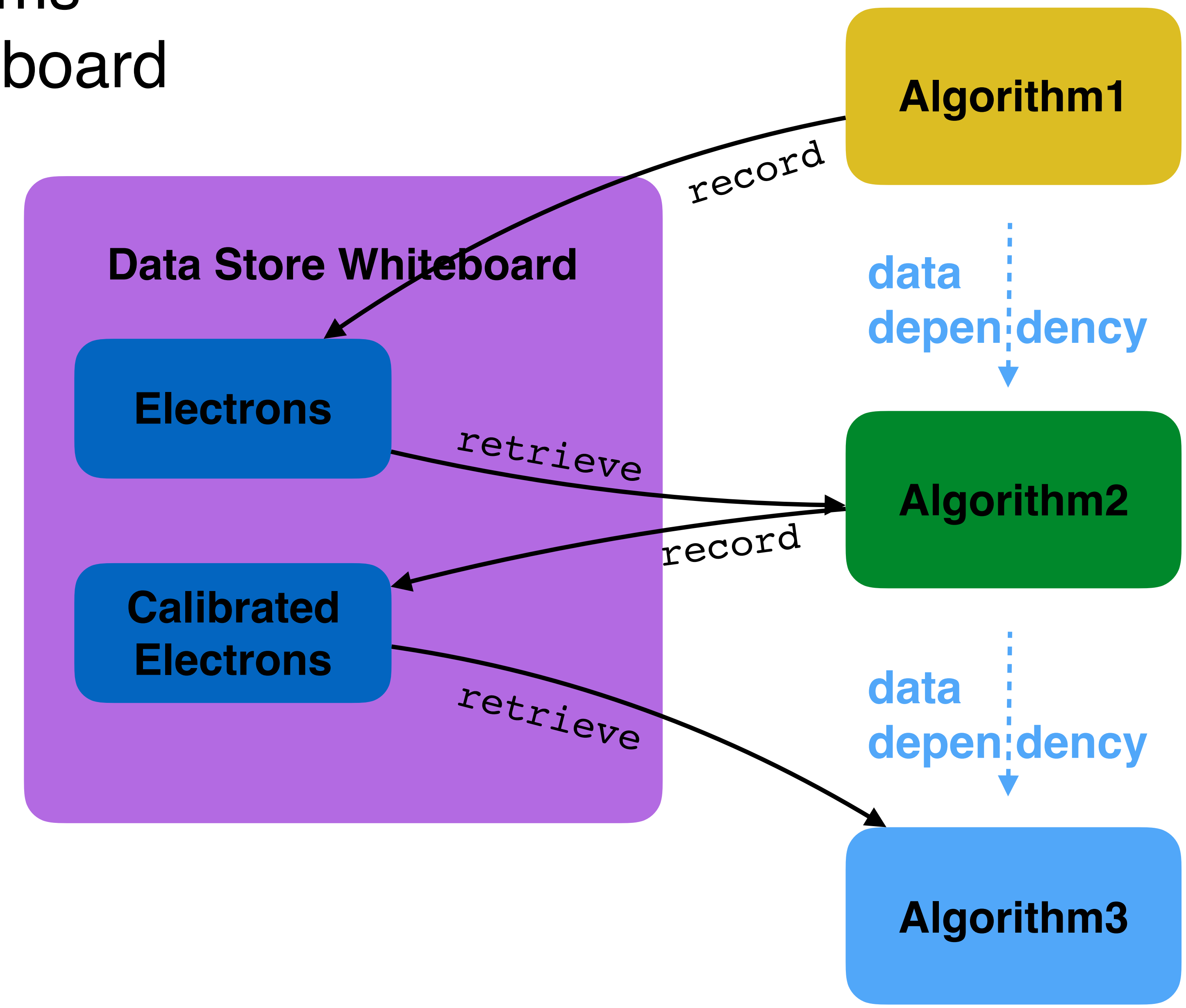
record data by passing the object and the key

```
xAOD::JetContainer* goodJets = new xAOD::JetContainer;
...
ATH_CHECK( evtStore()->record(goodJets,"GoodJets") );
```

StoreGate — runtime data access

In data stores, data exchange is done like **a whiteboard**

- post new data to the whiteboard to make it accessible to other algorithms
- read existing data from whiteboard



Data dependencies between algorithms are implicit by what they read/write.

Need to make sure algorithms run in correct order!

Sequences

The set of algorithms must be executed in some order, esp. to ensure that implicit data dependencies are satisfied

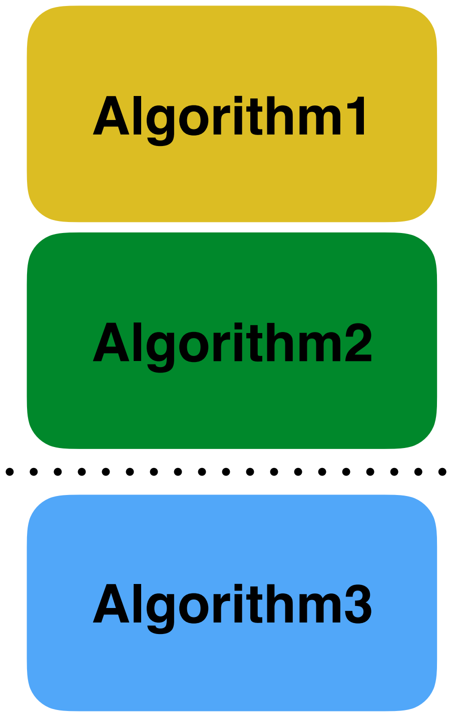
Athena defines a number of **sequences**: **ordered lists of algorithms**

In every processing stage (`initialize`, `execute`, `finalize`), the algorithms are called in that order.

Also advanced use-cases like sub-sequences, filters, etc…

```
algseq  = CfgMgr.AthSequencer("AthAlgSeq")
algseq += CfgMgr.MyAlgorithm()
algseq += CfgMgr.AnotherAlgorithm()
algseq += CfgMgr.AThirdAlgorithm()
```

Sequences — Filters




```
algseq  = CfgMgr.AthSequencer("AthAlgSeq")
algseq += CfgMgr.MyAlgorithm()
algseq += CfgMgr.SomeFilterAlg()
algseq += CfgMgr.AThirdAlgorithm()
```

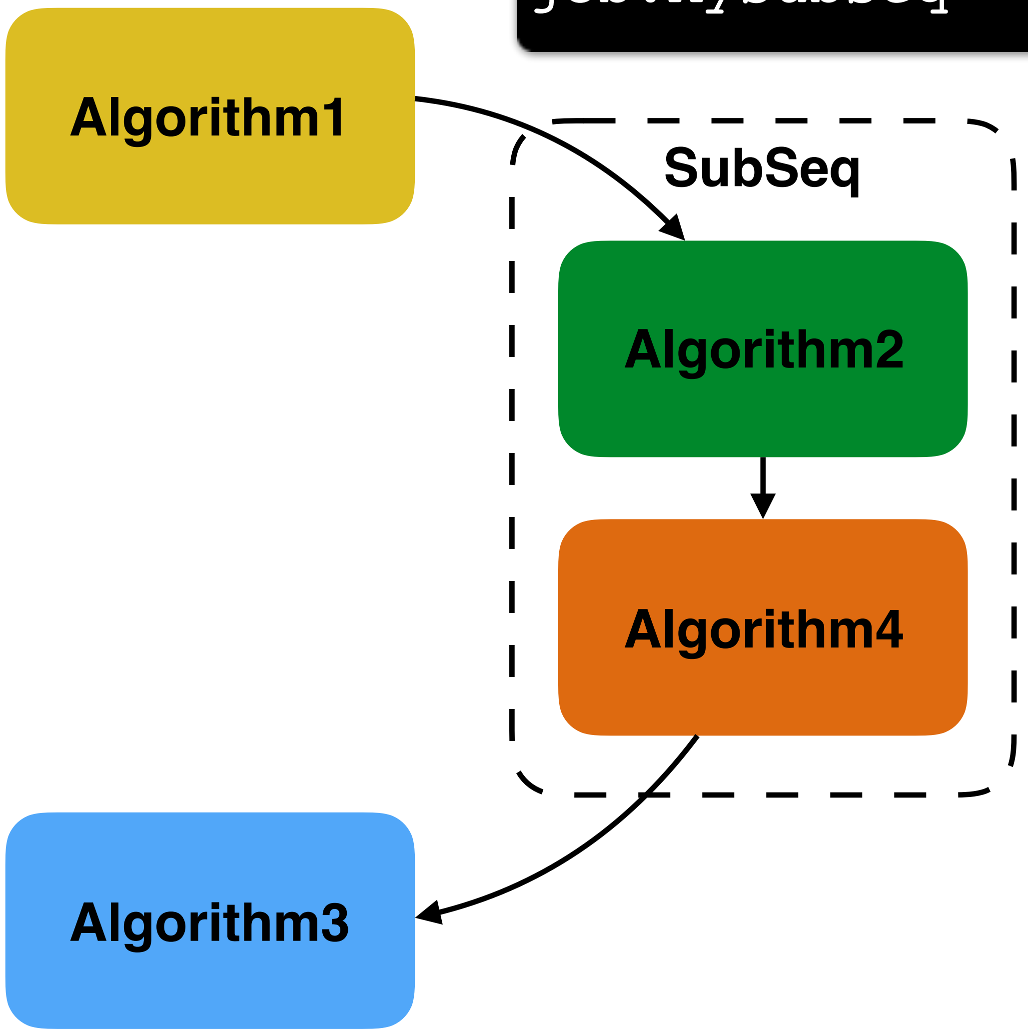
If filter fails, the following algorithms are not run for this event:

```
StatusCodeSomeFilterAlg::execute(){
  this->setFilterPassed(false);
  bool somedecision = decide();
  if(!somedecision) return StatusCode::SUCCESS;
  this->setFilterPassed(true);
  return StatusCode::SUCCESS;
}
```

Sequences — SubSequences

```
from AthenaCommon.AlgSequence import AlgSequence
from AthenaCommon import CfgMgr
job = AlgSequence()
job += CfgMgr.MyAlg("Algorithm1")


job += CfgMgr.AthSequencer("MySubSeq")
job.MySubSeq += CfgMgr.MyAlg("Algorithm2")
job.MySubSeq += CfgMgr.YourAlg("Algorithm4")
```



You can outsource chains of algorithms into a **subsequence**.

Use case:
put filter algorithms in subsequence without aborting main sequence to schedule algorithms.

Configurables

Athena is C++ but uses Python for configuration. C++ ↔ Python bridge defined via **Configurables**!

Tools and Algorithms can be declared as configurables (if you use helper scripts this is done for you).

Once declared there are accessible from Python. Can make data-members of the class accessible via **`declareProperty`** calls

C++

```
… {
  declareProperty("myprop", m_prop = 0)
}
```

set default value

Py

```
myAlg = CfgMgr.MyAlgorithm()
myAlg.myprop = 10
```

set custom value

JobOptions

With defined configurables, athena jobs can be made very flexible through joboptions files without having to re-compile.

```
theApp.EvtMax = 500
import AthenaPoolCnvSvc.ReadAthenaPool

svcMgr += CfgMgr.AthenaEventLoopMgr (EventPrintoutInterval = 100)

svcMgr.EventSelector.InputCollections = [ "/nfs/dust/atlas/group/atlas-d/tutorial-2019/
mc16_13TeV.
361107.PowhegPythia8EvtGen_AZNLOCTEQ6L1_Zmumu.merge.AOD.e3601_s3126_r9364_r9315/AOD.
11182597._003298.pool.root.1
" ]

# Fetch the AthAlgSeq, i.e., one of the existing master sequences where one should attach
all algorithms
algseq = CfgMgr.AthSequencer ("AthAlgSeq")

# Select muons above a pt threshold and
# create an output muon container only with the selected muons
algseq += CfgMgr.ParticleSelectionAlg ( "MyMuonSelectionAlg",
                                       InputContainer      = "Muons",
                                       OutputContainer     = "SelectedMuons",
                                       Selection           = "Muons.pt > 15.0*GeV"
                                       )
```

JobOptions — Input Data

Give list of files to athena for processing. Data will be available in the `evtStore()`. Streaming also works of course.

```
import AthenaPoolCnvSvc.ReadAthenaPool
svcMgr.EventSelector.InputCollections=[
  "/path/to/AODFile.root" ,
  "root:///also/streaming/data/works.root"
]
```

Specifying the input collection in the job option will always override the `--filesInput` argument of the `athena[.py]` command.

JobOptions — Writing Out Data

Can write both standard ROOT objects: `TTree`, `TH1` etc

```
from OutputStreamAthenaPool.MultipleStreamManager import MSMgr
outputStream = MSMgr.NewRootStream(“MyStream”,“myFile.root”)
```

…or write out xAODs (i.e. “POOL” format). Useful for preprocessing / skimming / calibrating / etc…

```
from OutputStreamAthenaPool.MultipleStreamManager import MSMgr
outputStream = MSMGr.NewPoolRootStream(“MyStream”,“myFile.pool.root”)
```

You can use the THistSvc to write standard ROOT objects: `TTree`, `TH1`

1. Add the service to your job options

```
from AthenaCommon.AppMgr import ServiceMgr
from GaudiSvc.GaudiSvcConf import THistSvc
ServiceMgr += THistSvc()
# specify stream and output file
ServiceMgr.THistSvc.Output += ["<STREAM> DATAFILE='MyRootFile.root'
OPT='RECREATE'"]
```

2. Initialise the service in your algorithm:

```
ServiceHandle<ITHistSvc> m_histSvc;

// add to constructor
m_histSvc("THistSvc", name),

StatusCode MyAlg::initialize() {
  ATH_CHECK(m_histSvc.retrieve());
  // register e.g. output tree
  ATH_CHECK(m_histSvc->regTree("<STREAM>/treename", myTree));
  return StatusCode::SUCCESS;
}
```

Building your own components

To integrate your own components into the framework you need to compile it against an existing athena release.

```
setupATLAS
mkdir xAODSoftwareTutorial
cd xAODSoftwareTutorial
mkdir source build run

cd build
asetup AthAnalysis,21.2.88,here
mv CMakeLists.txt ../source
cmake ../source

cmake --build $TestArea
```

Avoiding the Boilerplate — `acmd`

To integrate your C++ code, some boilerplate needs to be written — writing per hand is boring and error-prone (though educational)

`acmd` — a command line tool to generate skeleton code

```
acmd cmake new-skeleton MyAthenaxAODAnalysis
```

prepares directory structure for a new Athena package

```
acmd cmake new-analysisalg MyAnalysisAlg
```

`prepares header and source files for a new Algorithm`

Getting Help

Athena

- Athena Twiki
- Software Tutorial Twiki
- AthAnalysis Twiki

Useful mailing lists if you get stuck:

- atlas-sw-analysis-forum
- hn-atlas-PATHelp
- atlas-sw-pat-releaseannounce
- hn-atlas-dist-analysis-help
- hn-atlas-offlineSWHelp