

[TWiki](#) > [AtlasComputing Web](#) > [AtlasComputing](#) > [ComputingTutorials](#) > [SoftwareTutorial](#) > [SoftwareTutorialAnalysisInGitReleases](#)  
(2019-06-06, AdamJacksonParker)

Responsible:

[KerimSuruliz](#)

## Organising Analysis Code In Git/GitLab

### [General Overview](#)

- [Creating a git repository](#)

- [Setting the repository permissions](#)

- [Giving individuals access to a group](#)

- [Using LDAP to give everyone subscribed to an atlas mailing list access to a group](#)

- [Giving individuals access to a project](#)

- [Organising the analysis code](#)

- [Using submodules](#)

- [Using submodules to upgrade/downgrade athena packages](#)

- [Upgrading/downgrading multiple athena packages](#)

- [Examples of analysis frameworks in git](#)

### [Hands' On Exercise](#)

- [Create The Repository](#)

- [Adding Files To The Repository](#)

- [Creating The Project Configuration](#)

- [Adding An Analysis Package](#)

- [Building The Project](#)

- [Picking Up ElectronPhotonFourMomentumCorrection From ATLAS-EGamma](#)

- [Committing The Submodule](#)

- [Updating The Submodule](#)

- [Removing The Submodule](#)

- [Live example](#)

- [Summary](#)

This twiki documents some instructions and recommendations on migrating ATLAS analysis code from SVN to git. It is not meant to replace the [ATLAS git tutorial](#) which explains how to use git, with a particular focus on offline software.

## General Overview

In the first section we give a general overview of some topics, to try to give people a basic understanding of how code in [Git](#)/[GitLab](#) can be organised.

### Creating a git repository

The first thing you'll want to do is to create a project in git where you will put your analysis code. The project can reside in a user, institute or group repository. To create a project in your personal repository, you can go to

<https://gitlab.cern.ch/<your cern username>>

and click on "New project". You will then need to give a name to the project and set a visibility level. For the latter, it is best if you keep this at the default value: "Private". This ensures that non-ATLAS users with CERN gitlab access are unable to access your repository. Please see the next section for more details on setting permissions.

In case you need to create a new institute or group repository, you will first need to create a gitlab group. To do this, go to

<https://gitlab.cern.ch>

and click on the button on the top left to change from "Projects" to "Groups". Then click on "New group" to create your group.

Please be sensible when naming your group - there are no restrictions, but in general it's wise to follow SVN-style conventions as closely as possible, for example something like `atlas-inst-institutename` or `atlas-phys-susy-charm` are reasonable choices. If you are migrating CP or physics group level code, make sure you contact the relevant conveners/experts to ensure you put the code in the correct location, e.g. `ATLAS-EGamma` or `atlas-phys-susy-wg`.

Note: the project `atlas-physics` (<https://gitlab.cern.ch/groups/atlas-physics>) is an attempt to bring analysis repositories under the same location. The group has many sub-groups, that likely can host your repositories. For example, the repositories for the physics group `atlas-phys-sm-ew-sww-run2` has its code under <https://gitlab.cern.ch/atlas-physics/sm/ew/sww-run2>. Please contact the relevant conveners/experts, or if in doubt, `gitlab-atlas-physics-admins@cern.ch` to create the repositories.

### Setting the repository permissions

By default, projects and groups are private and invisible to other uses. If someone tries to access a Project repository they don't have access rights to, they will (somewhat confusingly) see a "404 not found" error.

## Giving individuals access to a group

Go to a group page in gitlab (e.g. <https://gitlab.cern.ch/mygrouppage/>) and click on "Members". Typically you will want to give people "Developer" or "Reporter" access. The latter category of members can view and pull the code but cannot commit to the repository.

## Using LDAP to give everyone subscribed to an atlas mailing list access to a group

Go to a group page in gitlab (e.g. <https://gitlab.cern.ch/mygrouppage/>), click on "Settings" and then on "LDAP group". Here you can add any of the usual ATLAS mailing lists (starting with atlas- or hn-, e.g. =hn-atlas-susy-wg) access. As an example, you could give "Reporter" access to everyone on hn-atlas-EGammaWG so that anyone subscribed to the egamma WG mailing list can view your code. You could then add individuals as developers for the individual projects in the group, following the next subsection.

## Giving individuals access to a project

Go to a project page in gitlab (e.g. <https://gitlab.cern.ch/institutes-sussex/susyanalysis/>) click on "Settings" and then on "Members". Here you can add individuals or other groups access to the project, so that you could for example give the "atlas-current-physicists" group "Reporter" access to the project. The project will then be visible to everyone in ATLAS, avoiding the 404 error mentioned above.

## Organising the analysis code

Your repository is now ready and you can start writing/migrating your code in git. If you are migrating existing code from SVN, then the `git svn clone` command can come in useful (see e.g. [this page](#)). Let us assume that your project is called `MyProject` consisting of one package, `MyUserAnalysis`, which in turn depends on an external package in a different git location - for example, [StopPolarizationRewighting](#). The project layout then looks as follows:

```
MyProject
|--- MyUserAnalysis : your analysis package
|--- StopPolarizationRewighting : external package your analysis code depends on
```

## Using submodules

In order to check out the components of your project, what you could do is to `git clone` each of `MyUserAnalysis` and `StopPolarizationWeighting` in turn. However, using the **submodule** feature of gitlab, you can actually package this into one single step, where you only perform one `git clone` on `MyProject`, and then both `MyUserAnalysis` and `StopPolarizationRewighting` will be cloned for you in one go. Using submodules in addition allows you to specify which particular commit of a package you would like to use - this is roughly equivalent to checking out a particular package tag from SVN. In the example above, let's assume we want to use commit `83fec592` of `StopPolarizationRewighting`.

You would do:

```
git submodule add https://:@gitlab.cern.ch:8443/atlas-phys-susy-wg/StopPolarizationRewighting.git
cd StopPolarizationRewighting
git checkout 83fec592
cd ..
git add StopPolarizationRewighting .gitmodules
git commit -am "Adding StopPolarizationRewighting submodule"
(...)
```

After executing `git submodule add`, the [StopPolarizationRewighting](#) will be cloned to the current directory, and the `.gitmodules` file in the current directory will be updated to contain information about the submodule (its name and location in git).

Once you added the submodule to your project, it (the particular commit specified using `git checkout` above) will be cloned together with the rest of your project if you do:

```
git clone --recursive https://:@gitlab.cern.ch:8443/<my project.git>
```

(Note the `--recursive` argument. Without this, the submodule will not be cloned.)

For example, try:

```
git clone --recursive https://:@gitlab.cern.ch:8443/akraszna/GitAnalysisTest1.git
```

This will clone the `AnalysisFramework` package which is part of the [GitAnalysisTest1](#) repository, together with two submodules: `athena` and `AnalysisFramework`.

Note that although the entire `athena` project will be cloned when you are developing your analysis, when submitting jobs to the grid, only the compiled code, available in the `build` directory will be bundled up. Therefore, the size of the job sent to the grid will not be enormous.

## Using submodules to upgrade/downgrade athena packages

In a frequent use case scenario, you need to upgrade or downgrade a particular package from the athena repository. In the past, with SVN, you would have accomplished this by checking out a particular tag of the package on top of the analysis release and then recompiling. With git, "package tags" no longer exist. You will need to figure out which athena repository commit hash you need to use; as long as the commit messages are sufficiently clear (as they ought to be!), this shouldn't be a daunting task. The list of commits for the 21.2 branch - from which the analysis releases are built - is available [here](#).

Let's say you want to use commit a55818f1 of the athena repository, as that contains an upgraded version of the [JetUncertainties](#) package. You need to do the following:

```
git submodule add .././atlas/athena.git
cd athena
git checkout a55818f1
(...)
```

Note a few things here.

- We used a "relative path" for the [atlas/athena](#) repository in the command. We could've just been very explicit and referred to it with `https://gitlab.cern.ch:8443/atlas/athena.git` as well. **But**, using this relative path formalism if a user checks out the analysis repository with an authentication mode other than Kerberos, for instance like the following, then the checkout of [atlas/athena](#) will also use the same authentication method.

```
git clone --recursive https://akraszna@gitlab.cern.ch/akraszna/GitAnalysisTest1.git
```

- You can also use the name of a branch, or a tag in your checkout command. Just like with any normal `git checkout` command.

Now you want to make sure that only the [JetUncertainties](#) package is picked up from the athena submodule. To achieve this, you need to use the `package_filters.txt` package. This is structured as follows (taken from [this example](#)):

```
1  # Pick up EventLoop from the Athena repository:
2  + athena/PhysicsAnalysis/D3PDTTools/EventLoop
3
4  # Don't pick up anything else from the Athena repository:
5  - athena/. *
6
7  # Pick up the analysis framework code:
8  + AnalysisFramework/framework/. *
9
10 # But don't pick up anything else from that repository:
11 - AnalysisFramework.*
12
13 # Everything outside of the submodules should be picked up.
14
```

In this example, only `EventLoop` would be picked up - to change that to `JetUncertainties`, you would use `+ athena/Reconstruction/Jet/JetUncertainties` instead.

## Upgrading/downgrading multiple athena packages

To upgrade or downgrade multiple packages, you can follow the same procedure as above, except that you will need to add multiple athena submodules, one for which commit that you need to pick up. The different modules should have different names, for which you can use the following syntax:

```
git submodule add .././atlas/athena.git athena2
(...)
```

You should use `athena_JetUncertainties` rather than `athena2` (if you're using the submodule to upgrade/downgrade `JetUncertainties`) or a similar meaningful name when checking out multiple submodules to avoid confusion.

## Examples of analysis frameworks in git

In addition to the simple example described in the hands-on exercise below, you can find a working, fully fledged example of an analysis (the multi b-jet analysis) organised as a self-contained project in git using submodules here: [https://gitlab.cern.ch/MultiBJets/MBJ\\_Analysis](https://gitlab.cern.ch/MultiBJets/MBJ_Analysis).

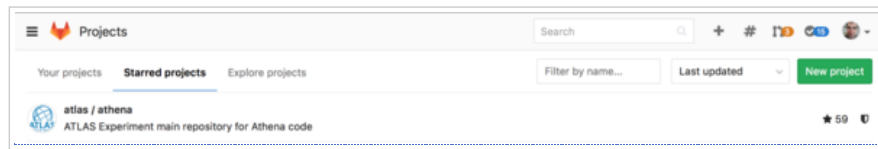
## Hands' On Exercise

The following is a step-by-step tutorial on how you would migrate some existing code from SVN, or just start writing something new from scratch.

The tutorial is designed to work as-is on lxplus. On a laptop, or personal/group desktop, depending on its CVMFS/AFS access, the instructions will of course have to be adjusted for your particular setup.

## Create The Repository

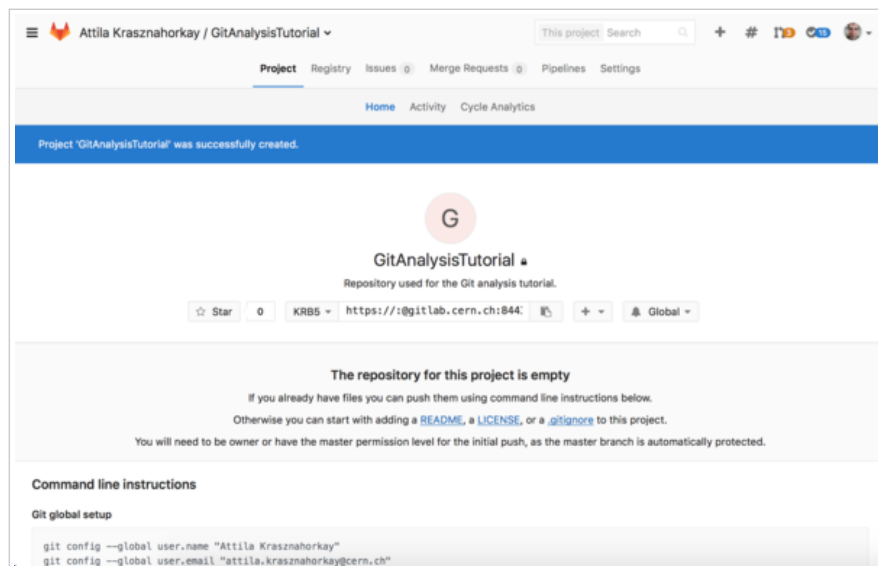
As shown before, you first need to create a new repository. For this exercise you should create a private one by clicking the "New project" button on the top-right.



**Note:** Depending on your personal configuration for GitLab, you may see a slightly different page. The "New project" button may instead just be a green button with a plus sign on it. But it should still be on the top-right of the page.

If you're part of multiple groups, make sure that the path to the repository is just your user name. Let's give it the name "GitAnalysisTutorial" on the page that comes up next, and give it some simple description. For this exercise just set the repository to be private.

Then just press the "Create project" button. This creates a new, empty, private repository for you. You should see a page like:



At this point you could just click on "Settings" to view all the settings of your new repository. Feel free to set a custom "avatar" (icon) for the repository, give specific people access rights to it, etc.

## Adding Files To The Repository

The GitLab page for the new empty repository is very helpful with explaining how you can start adding content to your new repository. Just follow those instructions.

To clone the (empty) repository locally, just do:

```
git clone https://:@gitlab.cern.ch:8443/<username>/GitAnalysisTutorial.git
```

**Note:** Feel free to use any other "access mode" for the cloning as well. On Ixplus this Kerberos based authentication is usually very reliable. But there are many people swearing on the "SSH access mode", and I myself prefer using "HTTPS access" when working with my repositories from macOS. The URLs for all of these can be copied from the webpage of your project.

Now do something like the following to add a main README file to the project:

```
cd GitAnalysisTutorial/
echo "Readme file to be written..." > README.md
git add README.md
git commit
# Write a reasonably long commit message in the default editor that comes up
git push -u origin master
```

You need to be very explicit in the last command, since at this point the repository on GitLab will be completely empty. You'll be creating the "master" branch in your GitLab repository with just that last command. (You'll be using the same formalism later on whenever you need to push a new branch into the "main" repository.)

## Creating The Project Configuration

After the README file your next step should be to create a CMakeLists.txt file for the root directory of your repository. This will be your project configuration file, describing how your entire analysis project is to be used.

In principle you could just grab the file created by asetup for you for this purpose.

```
cd ~/
mkdir temp
cd temp/
setupATLAS
asetup AnalysisBase,21.2.75,here
# ./CMakeLists.txt is the file you're interested in
```

But these project configuration files are also pretty simple to write from scratch. So you could just do so by hand. Taking the following example:

```
1 #
2 # Project configuration for GitAnalysisTutorial.
3 #
4
5 # Set the minimum required CMake version:
6 cmake_minimum_required( VERSION 3.4 FATAL_ERROR )
7
8 # Find the AnalysisBase project. This is what, amongst other things, pulls
9 # in the definition of all of the "atlas_" prefixed functions/macros.
10 find_package( AnalysisBase 21.2 REQUIRED )
11
12 # Set up CTest. This makes sure that per-package build log files can be
13 # created if the user so chooses.
14 atlas_ctest_setup()
15
16 # Set up the GitAnalysisTutorial project. With this CMake will look for "packages"
17 # in the current repository and all of its submodules, respecting the
18 # "package_filters.txt" file, and set up the build of those packages.
19 atlas_project( GitAnalysisTutorial 1.0.0
20   USE AnalysisBase ${AnalysisBase_VERSION} )
21
22 # Set up the runtime environment setup script. This makes sure that the
23 # project's "setup.sh" script can set up a fully functional runtime environment,
24 # including all the externals that the project uses.
25 lcg_generate_env( SH_FILE ${CMAKE_BINARY_DIR}/${ATLAS_PLATFORM}/env_setup.sh )
26 install( FILES ${CMAKE_BINARY_DIR}/${ATLAS_PLATFORM}/env_setup.sh
27   DESTINATION . )
28
29 # Set up CPack. This call makes sure that an RPM or TGZ file can be created
30 # from the built project. Used by Panda to send the project to the grid worker
31 # nodes.
32 atlas_cpack_setup()
```

You should now save, commit and push this new file into your repository.

At this point you're already able to point CMake at your new repository (clone), and do:

```
mkdir build
cd build/
cmake ../GitAnalysisTutorial/
make
```

It won't do anything useful, but it will do that very little successfully. 😊

## Adding An Analysis Package

---

Let's add a package actually capable of doing some analysis as the next step. You are very welcome to just use one of your existing analysis packages for this. But if you don't have something reasonably simple available at the moment (remember, we are not trying to finish writing a fully fledged analysis in this mini-tutorial), you could just pick up the example I put under:

`/afs/cern.ch/user/k/krasznaa/public/GitTutorial/ZAnalysis`

You could just copy that package into your repository, and then commit it with:

```
git add ZAnalysis/  
git commit  
# Write a reasonably long commit message in the default editor that comes up  
git push
```

Note that:

- This is supposed to be a very simple example package. It shows some good practices for how you should organise your code with [EventLoop](#), but definitely don't look at it as an "actual analysis";
- It builds 3 separate [EventLoop](#) Algorithms that together look for  $Z \rightarrow e^+e^-$  decays;
- It has one example "submission script" that can run the algorithms locally on some files on AFS.

## Building The Project

---

At this point you already have enough set up to build something meaningful. You can build the example discussed previously, like:

```
[bash][pcadp02]:build > cmake ../GitAnalysisTutorial/
-- The C compiler identification is GNU 6.2.0
-- The CXX compiler identification is GNU 6.2.0
-- Check for working C compiler: /cvmfs/atlas-nightlies.cern.ch/repo/sw/21.2/sw/lcg/releases/gcc/6.2.0/x86_64-slc6/bin/gcc
-- Check for working C compiler: /cvmfs/atlas-nightlies.cern.ch/repo/sw/21.2/sw/lcg/releases/gcc/6.2.0/x86_64-slc6/bin/gcc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /cvmfs/atlas-nightlies.cern.ch/repo/sw/21.2/sw/lcg/releases/gcc/6.2.0/x86_64-slc6/bin/g++
-- Check for working CXX compiler: /cvmfs/atlas-nightlies.cern.ch/repo/sw/21.2/sw/lcg/releases/gcc/6.2.0/x86_64-slc6/bin/g++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Found AnalysisBase: /cvmfs/atlas-nightlies.cern.ch/repo/sw/21.2/2017-06-07T2125/AnalysisBase/21.2.0/InstallArea/x86_64-slc6-gcc62-opt (version: 21.2.0)
-- Found AnalysisBaseExternals: /cvmfs/atlas-nightlies.cern.ch/repo/sw/21.2/2017-06-07T2125/AnalysisBaseExternals/21.2.0/InstallArea/x86_64-slc6-gcc62-opt (version: 21.2.0)
-- Setting ATLAS specific build flags
-- checker_gccplugins library not found
-- Package(s) in AnalysisBaseExternals: 10
-- Using the LCG modules without setting up a release
-- Package(s) in AnalysisBase: 136
-- Configuring ATLAS project with name "GitAnalysisTutorial" and version "1.0.0"
-- Using build type: RelWithDebInfo
-- Using platform name: x86_64-slc6-gcc62-opt
-- Unit tests will be built by default
-- Found 1 package(s)
-- Using the LCG modules without setting up a release
-- Considering package 1 / 1
-- No package filtering rules read
-- Configuring the build of package: ZAnalysis
-- Found ROOT: /cvmfs/atlas-nightlies.cern.ch/repo/sw/21.2/2017-06-07T2125/AnalysisBaseExternals/21.2.0/InstallArea/x86_64-slc6-gcc62-opt/include (found version "6.08/06")
-- Found ROOT: /cvmfs/atlas-nightlies.cern.ch/repo/sw/21.2/2017-06-07T2125/AnalysisBaseExternals/21.2.0/InstallArea/x86_64-slc6-gcc62-opt/include (found version "6.08/06")
-- Number of packages configured: 1
-- Using the LCG modules without setting up a release
-- Including the packages from project AnalysisBase - 21.2.0...
-- Including the packages from project AnalysisBaseExternals - 21.2.0...
-- Using the LCG modules without setting up a release
-- Generated file: /home/krasznaa/projects/git/mini-tutorial/build/x86_64-slc6-gcc62-opt/packages.txt
-- Generated file: /home/krasznaa/projects/git/mini-tutorial/build/x86_64-slc6-gcc62-opt/compilers.txt
-- Generated file: /home/krasznaa/projects/git/mini-tutorial/build/x86_64-slc6-gcc62-opt/ReleaseData
-- Generating external environment configuration
-- Using the LCG modules without setting up a release
-- Writing runtime environment to file: /home/krasznaa/projects/git/mini-tutorial/build/x86_64-slc6-gcc62-opt/env_setup.sh
-- Configuring done
-- Generating done
-- Build files have been written to: /home/krasznaa/projects/git/mini-tutorial/build
[bash][pcadp02]:build > make
[ 14%] Generating CMakeFiles/ZAnalysisLibCintDict.cxx, ../x86_64-slc6-gcc62-opt/lib/libZAnalysisLib_rdict.pcm, CMakeFiles/ZAnalysisLib.dsomap
Scanning dependencies of target ZAnalysisHeaderInstall
Scanning dependencies of target Package_ZAnalysis_tests
[ 28%] Generating ../x86_64-slc6-gcc62-opt/include/ZAnalysis
...
```

## Picking Up ElectronPhotonFourMomentumCorrection From ATLAS-EGamma

Let's say that at this point in our analysis we realise that we need to pick up the bleeding edge version of [ElectronPhotonFourMomentumCorrection](#). Before merging the latest version of a CP package into the 21.2 (analysis) branch of [atlas/athena](#), CP groups will likely keep them in personal/group repositories for a while first. (While the package's new version is undergoing some testing.)

The E/Gamma group has a group-level fork of [atlas/athena](#) under [ATLAS-EGamma/athena](#) exactly for this purpose. Unfortunately at the time of writing this tutorial they don't have any analysis packages in there newer than their versions in the 21.2 nightlies, but in the future this is very likely going to change.

For this exercise let's say that we want to pick up ElectronPhotonFourMomentumCorrection from the `upstream/svn_pull_of_ElectronPhotonFourMomentumCorrection_trunk` branch of [ATLAS-EGamma/athena](#). For this we first need to make that repository a submodule of our analysis repository. We can do it with:

```
git submodule add .././ATLAS-EGamma/athena.git ElectronPhotonFourMomentumCorrection_update
cd ElectronPhotonFourMomentumCorrection_update/
git checkout 21.2
cd ..
```

Now, be careful. At this point you cloned the **entire** offline repository as part of you analysis repository. Meaning that if you tried to build your project now, it would try to build >2000 packages. (And due to CMake configuration errors would die anyway.) But we only want to build ElectronPhotonFourMomentumCorrection from the git submodule, and ZAnalysis from inside our own repository. To do this, we need to create a file called `package_filters.txt`, and put it in the root directory of our repository. Beside the project `CMakeLists.txt` file. With this content:

```
#
# Package filter file for GitAnalysisTutorial.
#

# Pick up just one package from the e/gamma submodule:
+ ElectronPhotonFourMomentumCorrection_update/PhysicsAnalysis/ElectronPhotonID/ElectronPhotonFourMomentumCorrection
- ElectronPhotonFourMomentumCorrection_update.*

# Every other package (which is part of this repository) will be picked
# up for the build.
```

When you run your CMake configuration now, you'll get a **lot** of output lines like:

```
...
-- Considering package 1 / 2168
-- Package filtering rules read:
-- + ElectronPhotonFourMomentumCorrection_update/PhysicsAnalysis/ElectronPhotonID/ElectronPhotonFourMomentumCorrection
-- - ElectronPhotonFourMomentumCorrection_update.*
-- Considering package 11 / 2168
-- Considering package 21 / 2168
-- Considering package 31 / 2168
-- Considering package 41 / 2168
...
```

This is normal. (For the moment. We have an open ticket for reducing the number of lines printed in this setup.) At the end of the configuration you should still see:

```
-- Number of packages configured: 2
```








Executing `make` now will first build ElectronPhotonFourMomentumCorrection, and then ZAnalysis against it.

## Committing The Submodule

With all of this done, you should commit your latest updates like:

```
git add ElectronPhotonFourMomentumCorrection_update .gitmodules package_filters.txt
git commit
# Write a reasonably long commit message in the default editor that comes up
git push
```

Note that when you check your repository on the GitLab webpage, you'll see something like the following:

Name	Last commit > 7381375d  about 4 hours ago - Updated the example job to use LocalD...	History	Last Update
 ElectronPhotonFourMomentumCorrection_... @ f8ff67e0			
 ZAnalysis	Updated the example job to use LocalDriver instead of DirectDriver.		about 4 hours ago
 .gitmodules	Added the e/gamma fork as a submodule of the repository. To show how in t...		about 5 hours ago
 CMakeLists.txt	Added a simple project configuration file to the repository.		about 5 hours ago
 README.md	Updated the README a bit.		about 4 hours ago
 package_filters.txt	Added a package filtering file to the repository, since we only want to pick		about 4 hours ago

Stating that my analysis repository is set up to pick up hash `f8ff67e0` from [ATLAS-EGamma/athena](#). This is the intended behaviour. Even though we "checked out" the head of the `upstream/svn_pull_of_ElectronPhotonFourMomentumCorrection_trunk` branch, Git just stored the hash that we actually ended up checking out, not the name of the branch. Because Git intends to check out the same version of the code the next time that we picked up now. Not some possibly updated version of the code, which may not even work with our analysis code anymore.

## Updating The Submodule

To get the latest version of the `upstream/svn_pull_of_ElectronPhotonFourMomentumCorrection_trunk` branch, after it was updated by the E/Gamma group, one would do something like:



```
cd ElectronPhotonFourMomentumCorrection_update
git checkout 21.2
git pull
cd ..
```

If this pulled in a different version of the code, which we now want to pick up for our analysis, we can commit the update (after some testing) like:

```
git add ElectronPhotonFourMomentumCorrection_update
git commit
# Write a reasonably long commit message in the default editor that comes up
git push
```

Removing The Submodule

Once you don't need the submodule anymore, for instance because the updates have been merged into the 21.2 branch of [atlas/athena](#), you need to remove it from your analysis repository. You can do this with:

```
rm -rf ElectronPhotonFourMomentumCorrection_update
git rm ElectronPhotonFourMomentumCorrection_update
git commit
# Write a reasonably long commit message in the default editor that comes up
git push
```

Yes, you see it correctly. You need to first remove the directory from disk. Make sure that you double-check that you execute the correct command with `rm -rf!!`

You should also remove the corresponding lines from `package_filters.txt`. Although this is not terribly urgent. Even with the lines still present, your Git project should still be functional, even if the submodule is no longer there.

Live example





As you can imagine, I tried this all out for myself while writing the tutorial. You can find the repository that I used, under [akraszna/ZAnalysisTest](#). You can just have a look at that if you're not completely sure how your own `GitAnalysisTutorial` repository should have ended up looking.

Summary

This concludes the mini-tutorial. As was written before, Git submodules can be used to pick up multiple packages for an analysis project from multiple different repositories like this. You should just remember to remove the submodules once the latest version of the package was merged into the 21.2 branch of [atlas/athena](#). (Since at that point you could just pick up the latest 21.2 nightly for your analysis, even before a new 21.2.X release is tagged.)

More advanced topics may be tacked on to this TWiki in the future, until then you should contact the ASG group if your analysis needs some special setup that you don't know how to implement in Git/GitLab.

Major updates:  
-- [KerimSuruliz](#) - 2017-06-06  
Responsible: [KerimSuruliz](#)  
Last reviewed by: **Never reviewed**

I	Attachment	History	Action	Size	Date	Who	Comment
	<a href="#">Screen_Shot_2017-06-08_at_14.26.14.png</a>	r1	<a href="#">manage</a>	81.2 K	2017-06-08 - 14:30	<a href="#">AttilaKrasznahorkay</a>	New project creation
	<a href="#">Screen_Shot_2017-06-08_at_14.38.07.png</a>	r1	<a href="#">manage</a>	178.3 K	2017-06-08 - 14:39	<a href="#">AttilaKrasznahorkay</a>	New project page
	<a href="#">Screen_Shot_2017-06-08_at_14.43.50.png</a>	r1	<a href="#">manage</a>	247.2 K	2017-06-08 - 14:44	<a href="#">AttilaKrasznahorkay</a>	Empty project
	<a href="#">Screen_Shot_2017-06-08_at_17.51.29.png</a>	r1	<a href="#">manage</a>	142.8 K	2017-06-08 - 17:51	<a href="#">AttilaKrasznahorkay</a>	

Topic revision: r14 - 2019-06-06 - [AdamJacksonParker](#)

