Table of Contents

OBSOLETE

This page has been superseded by:
https://lhcb-core-doc.web.cern.ch/lhcb-core-doc/Development.html#use-of-git-in-lhcb

Introduction

The rationale and the basics of the use of Git in LHCb have been presented and discussed in a few places:

- 62nd Analysis and Software Week (2016-01-25)
- Core Software Meeting (2016-01-06)
- LHCb-INT-2016-001

This page is meant to summarize the main commands to work with Git in LHCb. Some complementary information can be found in the LHCb Starterkit. This page can also be used to document tips and tricks, and toubleshooting instructions.

About Git

Git is a distributed version control system widely used. See http://git-scm.com/ for more details.

There are a lot of resources available on the web, for example:

- Git for Subversion users, Part 1: Getting started
- Git for Subversion users, Part 2: Taking control
- Git Book
- cheatsheets (PDF)
    - www.cheat-sheets.org
    - GitHub
    - Atlassian
- Git tutorial (Google search)

CERN opted for GitLab as a Git hosting platform, at https://gitlab.cern.ch, where you can find documentation about GitLab basics.

To simplify Git use in LHCb, a few custom Git sub-commands have been developed.

Prerequisites

There's a few operations that must be done once (and only once) before start working with Git at CERN.

1. Git command line setup, n.b. for LHCb users an account has likely been created for you on https://gitlab.cern.ch. Go to https://gitlab.cern.ch and see if your CERN single sign-on properly logs you on, then click on "Profile Settings" on the left toolbar and check the account details.
2. Set Git default *push* policy
   - git < 1.7.11 (SLC6, lxplus):
     ```
     git config --global push.default current
     ```
   - git >= 1.7.11 (CentOS7, Ubuntu, Mac, ...):
     ```
     git config --global push.default simple
     ```
3. (only needed for "ssh authentication") Add SSH keys to GitLab, usually GitLab asks you to do so if it was not done yet (you can upload the key under "Profile Settings" on https://gitlab.cern.ch).
4. (only needed for "ssh authentication") `git config --global lb-use.protocol ssh`
5. (only needed for "https authentication") fix https authentication in newer git versions see here for more details
   - for work on lxplus7 (for the time being):
     ```
     git config --global http.emptyAuth true
     ```

Note that step 3 and 4 are not mandatory, because you can use Kerberos based authentication. The ssh protocol has some advantages over https (used for Kerberos) e.g. "large" transfers may fail over https. On the other hand, the ssh port is blocked in some networks (many hotels, some institutes) while the https port is usually open.

Point 2 is to avoid mistakes and to get a behavior similar to that of Git 2.0 on old versions of Git (current version at 2016-04-03 is 2.8.1).

Using Git for LHCb development

Satellite projects

Working with *satellite projects* (AKA *local projects*) is very useful for quick (a few commits) or limited (a few packages) developments.

First we need to create the satellite project

```
lb-dev Project/vXrY
cd ProjectDev_vXrY
```

then declare which project we want to get packages from

```
git lb-use Project
```

and get the package/subdirectory code, from the *master* (or another) branch

```
git lb-checkout Project/master Some/Package
```

or from a given project version

```
git lb-checkout Project/vXrY Some/Package
```

At this point, we can work on the changes we need to make, for example

```
vim Some/Package/src/MyStuff.cpp
make
make test
git add Some/Package/src/MyStuff.cpp
git commit -m 'fixing feature abc (JIRATICKET-123)'
vim Some/Package/src/MyStuff.cpp
make
make test
git commit -a -m 'improved fix to JIRATICKET-123'
```

Once we are happy with our changes, we can push them to a branch (JIRATICKET-123 in this example) in the remote repository for the project

```
git lb-push Project JIRATICKET-123
```

and create a merge request, by going to the project page on https://gitlab.cern.ch/ and clicking on + then "New merge request". If the changes you're proposing are not ready for merging, you can add 'WIP:' at the beginning of the merge request's title, and can then discuss ideas in the MR.

SetupProject workaround

If for some reason (which you don't have time to debug right now) you cannot use lb-dev, you can also use SetupProject

```
SetupProject --build-env PROJECT vXrY
git init
```

and then continue as above

```
git lb-use Project
```

etc.

Mixing branches

In git, one does not just commit the current content of a file, but a change history. That means whenever you commit work, git needs to know which older version of a branch your change should be applied to. When checking out files from multiple branches with git lb-checkout, this will lead to confusion (i.e. error messages about merge conflicts when doing git lb-push). In this case a combination of vanilla git and git-lb commands might be useful.

Example: You want to commit changes on top of the 2018-patches branch of a project, but pick the master version of a few files for that. You want to use the nightly that was compiled last Monday. The steps then look like:

```
lb-dev --nightly=lhcb-2018-patches Mon DaVinci/2018-patches
cd ./DaVinciDev_2018-patches
git lb-use Phys
# checkout Phys/LoKiPhys with git lb-checkout
git lb-checkout Phys/2018-patches Phys/LoKiPhys
# checkout three files from the master branch with vanilla git
git checkout Phys/master -- Phys/LoKiPhys/python/LoKiPhys/functions.py
git checkout Phys/master -- Phys/LoKiPhys/LoKi/BeamLineFunctions.h
git checkout Phys/master -- Phys/LoKiPhys/src/BeamLineFunctions.cpp
# edit Phys/LoKiPhys/python/LoKiPhys/functions.py to resolve small bug fix
git add Phys/LoKiPhys/python/LoKiPhys/functions.py
git commit
git lb-push Phys My_awesome_feature
```

The git lb-push command then should not fail in merge conflicts and is suited for a merge request into the 2018-patches branch (i.e. the branch used in the git lb-checkout command).

Whole projects

The tools to work with satellite projects have limitations. In particular they fail on binary files and do not support branch juggling.

For this reason and because working with plain git is better, it's generally suggested to work on whole projects with a workflow like:

```
git clone --recurse-submodules ssh://git@gitlab.cern.ch:7999/lhcb/Project.git
cd Project
lb-project-init
# use '-b' only if that branch does not exist in the remote repository
git checkout -b JIRATICKET-123
make
make test
# edit edit edit...
git add Some/Package/src/MyStuff.cpp
git commit -m 'fixing feature abc (JIRATICKET-123)'
vim Some/Package/src/MyStuff.cpp
make
make test
git commit -a -m 'improved fix to JIRATICKET-123'
git push -u origin JIRATICKET-123
```

Data Packages

Since December, 12 2017 data packages are in managed in Gitlab (see announcement on lhcb-core-soft).

To work on a data package is equivalent, in some sense, to working with full projects.

You can get a local clone of a data package with something like `git clone https://gitlab.cern.ch/lhcb-datapkg/Hat/Name.git Hat/Name`, or, as a handy shortcut, `git lb-clone-pkg Hat/Name`. So, for example:

```
git lb-clone-pkg Gen/DecFiles
cd Gen/DecFiles
git checkout -b ${USER}/my-changes
# edit edit edit
git add dkfiles/YourDecFile.dec
git add doc/release.notes
git commit -m "Added dkfiles for XXX"
git push -u origin ${USER}/my-changes
# create merge request
```

If you have a full stack checked out, you can do the following:

```
# from your stack directory
mkdir DBASE
cd DBASE
git lb-clone-pkg PRConfig
cd ..
make Brunel/purge
make
```

The purge is annoying but necessary. Do not purge the whole stack, just the folder with the application (Brunel, Moore, DaVinci). Then you avoid a costly recompilation.

Examples

To better understand how to apply the recipes, here we have a few specific examples.

Committing Hlt2 lines

Start from the latest Moore version with lb-dev.

```
lb-dev Moore/v<latest>
cd MooreDev_v<latest>
git lb-use Hlt
git lb-checkout Hlt/master Hlt/Hlt2Lines
```

Make your changes. Add a line describing your change in doc/release.notes

```
git diff
git status
```

to review changes and see which files need to be added to your commit

```
git add <file1> <file2>...
git commit -a -m "Describe your changes here, first line short summary, second line can be more c
git lb-push Hlt <your name>/<something describing your change>
```

Go to gitlab.cern.ch, search for the project LHCb/Hlt (https://gitlab.cern.ch/lhcb/Hlt). Create a merge request with the master branch.

Upgrade Tracking

Use case: work on a few packages from different projects in a specific nightly build slot.

Note: The branch *upgradeTracking* was merged to *master* and deleted. Therefore, from now on it is recommended to use the *master* branch.

Development:

```
#Create environment
lb-dev --nightly lhcb-head Brunel/HEAD
#for a specific nightly add Mon,Tue,Wed,Thu,Fri,Sat,Sun as i.e.: lb-dev --nightly lhcb-head Mon B
cd BrunelDev_HEAD

# Add packages from Rec that you want to modify. e.g.:
git lb-use Rec
git lb-checkout Rec/master Tr/TrackFitter
git lb-checkout Rec/master Tf/TrackSys          #contains properties of the tracking sequence
git lb-checkout Rec/master Pr/PrAlgorithms    #contains tracking algorithm ( Seeding/Forward/Matc
git lb-checkout Rec/master Pr/PrMCTools       #contains truth matching tool and tools to study de
git lb-checkout Rec/master Pr/PrKernel          #contains base classes and interfaces used by PrA

# Add Brunel for testing:
git lb-use Brunel
git lb-checkout Brunel/master Rec/Brunel

# add/change files

#Compile
make
```

```
#locally commit the code
git commit -a -m "changed something"

#Test the changes
#run all qm tests
make test
#or, for specific tests:
make test ARGS="-R brunel-upgrade-baseline"

#push to server
git lb-push Rec upgrade-newbranchname
```

Merge requests

- go to https://gitlab.cern.ch/lhcb/Rec/merge_requests/new (from https://gitlab.cern.ch/lhcb/Rec click on +, then on "New merge request").
- select *my-new-feature* as source branch and *master* as destination branch, and click on "Compare branches and continue"
- fill in the form with details and submit
- repeat for all projects (Brunel)

Update local packages **after** the merge requests are applied

```
cd BrunelDev_master
git fetch --all
git lb-checkout Rec/master Tr/TrackFitter
git lb-checkout Brunel/master Rec/Brunel
```

Upgrade Tracking Example 2

Use case: work on significant changes to the tracking code, building packages from scratch

Preparation (can be setup as a shell script for easy rebuilds):

```
source /cvmfs/lhcb.cern.ch/lib/LbEnv-stable # Only needed for non-lxplus systems
#lb-set-platform ${CMTCONFIG/opt/dbg} # Uncomment for debug builds
export CMAKE_PREFIX_PATH="$HOME/upgradeTracking:/cvmfs/lhcbdev.cern.ch/nightlies/lhcb-upgradeTrac
export CCACHE_DIR=/<somewhere>/.ccache
export CMAKEFLAGS=-DCMAKE_USE_CCACHE=ON
unset VERBOSE
```

Note that for best performance you should pick / local to the machine you run on. Also, you should increase the cache size a bit, as the default is a little small. For instance the following will set the size to 10G.

```
ccache --max-size=10G
```

Building (using ssh for gitlab authentication):

```
mkdir ~/upgradeTracking

cd ~/upgradeTracking
git clone --recurse-submodules ssh://git@gitlab.cern.ch:7999/lhcb/LHCb.git LHCB/LHCB_upgradeTrack
cd LHCB/LHCB_upgradeTracking
git checkout -b upgradeTracking origin/upgradeTracking

cd ~/upgradeTracking
git clone --recurse-submodules ssh://git@gitlab.cern.ch:7999/lhcb/Lbcom.git LBCOM/LBCOM_upgradeTr
cd LBCOM/LBCOM_upgradeTracking
```

```
git checkout -b upgradeTracking origin/upgradeTracking

cd ~/upgradeTracking
git clone --recurse-submodules ssh://git@gitlab.cern.ch:7999/lhcb/Rec.git REC/REC_upgradeTracking
cd REC/REC_upgradeTracking
git checkout -b upgradeTracking origin/raaij-upgradeTracking

cd ~/upgradeTracking
git clone --recurse-submodules ssh://git@gitlab.cern.ch:7999/lhcb/Brunel.git BRUNEL/BRUNEL_upgrad
cd BRUNEL/BRUNEL_upgradeTracking
git checkout -b upgradeTracking origin/upgradeTracking
```

Edit each of the CMake lists in the last three packages to use the `upgradeTracking` version instead of the versions listed for the above packages. In each directory, in order, run the following commands to build:

```
lb-project-init
make configure
make install
```

Note: if you want or need to use prerelease versions of heptools you can edit the toolchain.cmake file created by the `lb-project-init` command with the additional lines at the top:

```
set(CMAKE_PREFIX_PATH /cvmfs/lhcb.cern.ch/lib/lhcb /cvmfs/lhcb.cern.ch/lib/lcg/releases /cvmfs/sf
list(REMOVE_DUPLICATES CMAKE_PREFIX_PATH)
```

which adds the location of the central heptools location as used by the nightly builds.

Building everything locally example

This example is a minor modification of the upgrade tracking example2. In this example the aim is to build everything, from `Gaudi` upwards, locally. The advantage of this approach is it decouples your development from the nightlies, the only dependencies taken from `/cvmfs` are the `LCG` releases.

First, set up your environment in a very similar way :-

```
# Only needed for non-lxplus systems
source /cvmfs/lhcb.cern.ch/lib/LbEnv-stable
# If required, set CMTCONFIG to the platform you wish to build
lb-set-platform <platform>
# Define the sub-directory you wish to use for the project checkouts
export User_release_area="/path/to/somewhere"
lb-set-workspace ${User_release_area}
export CMAKE_PREFIX_PATH=${User_release_area}:${CMAKE_PREFIX_PATH}
# Use ccache
export CCACHE_DIR=/<somewhere>/.ccache
export CMAKEFLAGS="-DCMAKE_USE_CCACHE=ON"
```

Then, starting with `Gaudi`, checkout and build each project in turn

```
cd $User_release_area
git clone --recurse-submodules ssh://git@gitlab.cern.ch:7999/gaudi/Gaudi.git
cd Gaudi
git checkout <branch>
lb-project-init
make configure
make install

cd $User_release_area
git clone --recurse-submodules ssh://git@gitlab.cern.ch:7999/lhcb/LHCb.git
cd LHCb
git checkout <branch>
```

```
lb-project-init
make configure
make install
```

Repeat for whatever project you require, up to the top level application (`Brunel`, `DaVinci` etc.). Note in each step running `make configure` explicitly before `make install` is technically not required, but its useful to do the first time as it allows you to check the configuration before continuing. In particular check that each project is correctly using those it requires from your build area.

Do not worry if, the first time you build a project it takes a long time. The likes of `LHCb` and `Rec` indeed take a short while to build at first. In subsequent builds though, you will find only the file you have touched, or those that use them, will need to be rebuilt each time. In addition, if you ever need to trigger a complete rebuild, using `ccache` will significantly improve the time those rebuilds take.

Tips & Tricks

Replacement for "svn update" in local projects

In an `lb-dev` generated project, with packages checkout out with `git lb-checkout`, it is not possible to run `git merge` to get the local copy of the packages synchronized with remote changes while keeping the local ones.

It is possible, though, to emulate the behaviour of `svn update` via a `git rebase` with the following procedure:

Show example of local project set up ▸ Hide example ▾

```
lb-dev LHCb/v40r1
cd LHCbDev_v40r1
git lb-use -q LHCb
# get an old version as an example
git lb-checkout LHCb/v40r0~ GaudiObjDesc
# ... some development ... edit edit edit commit commit commit

# update the package
(
imported=$(git config -f .git-lb-checkout lb-checkout.LHCb.GaudiObjDesc.imported)
git checkout $(git config -f .git-lb-checkout lb-checkout.LHCb.GaudiObjDesc.base)
git lb-checkout $imported GaudiObjDesc
)
git lb-checkout LHCb/master GaudiObjDesc
git rebase HEAD master
# resolve conflicts (if needed) following instructions from git-rebase
```

Remove from a local project a package checked out with "git lb-checkout"

If in your local project you have a copy of a package you do not need anymore, you can safely remove it from the local repository and remove the associated metadata with:

```
cd ~/cmtuser/MyProjectDev_vXrY
git rm -r Hat/MyPackage
git config -f .git-lb-checkout --remove-section lb-checkout.TheProject.Hat/MyPackage
git add .git-lb-checkout
git commit -m 'removed Hat/MyPackage'
```

where you replace `TheProject` and `Hat/MyPackage` with the correct values (you can use `git config -l -f .git-lb-checkout` to be sure).

Commit a change to both `master` and `run2-patches` branches

`master` and `run2-patches` branches are distinct branches for run3 and run1/run2 development respectively. Both branches are in the process of being cleaned up to contain only code relevant for the supported run period, but some of the code is shared. When making a fix or adding a new feature, think whether it is relevant to both branches - if so you should commit it to both. While it is always possible to make distinct commits to the two branches, it is usually cleaner to make the commit to one branch and then cherry-pick it to the other, as this gives a more legible Git history and can avoid merge conflicts in future. Just make the merge request to one of the two branches and make a note in the MR description for the release manager to port to the other branch. Depending on the type of change it may be better to start from one branch rather than the other: for bug fixes, it is usually better to apply first to `run2-patches`; for new features, apply first to `master`

N.B. if your branch originated from anything other than the target branch, you must first "rebase" the branch, see instructions below

Commit a change to a legacy `xxx-patches` branch

Legacy branches are intended for maintenance of software versions used in official processings (e.g. a `Reco` or `Stripping` version, but also the trigger for a given year). As such, their behaviour should not be modified, so any changes should be considered carefully: it is OK to add new features, but generally not OK to modify algorithms in ways that change their performance, even if it's a bug fix. When propagating a new feature to a (set of) legacy branch(es), it is best to start from the most recent branch (generally, `run2-patches`) and back port. This is best done by a release manager, just make a note in the MR description of the MR that introduces the change, which legacy branches it should be back-ported to.

Rebase a feature branch to a different origin branch

Sometimes you may have been working on a new feature or bug fix in a branch that you created starting from e.g. `master`, but you wish to make the merge request against another branch, e.g. `run2-patches`. Before you push your branch you need to rebase it. In the following example, we assume that you are working on a feature branch called `myFeatureBranch` in `Phys` project and wish to rebase it to `run2-patches` branch. Proceeed as follows:

```
cd $TMPDIR
git clone --recurse-submodules ssh://git@gitlab.cern.ch:7999/lhcb/Phys.git
cd Phys
git checkout myFeatureBranch
git rebase -i origin/run2-patches
# an editor opens. You should keep only the lines with your commits and save
git log  # make sure the history looks right
git push --force  # much better to use --force-with-lease, but you might not have it in your git
# see e.g. https://developer.atlassian.com/blog/2015/04/force-with-lease/
```

Port an existing commit to another branch

Sometimes you wish to commit a change to more than one branch. If you cannot rely on the automatic merging described in previous headings, use the `git cherry-pick` command: once your commit(s) is in a given branch, just checkout the other branch and issue

```
git cherry-pick sha1 [...]
```

where sha1 is the name of your commit, or a space separated list of commits (in the order they should be applied).

cherry-picking may trigger conflicts if the commit does not apply cleanly. In such a case, one has to resolve the conflicts, readd the fixed files to the index and say

```
git cherry-pick --continue
```

as explained by git on its output

Last remark : if you find somewhere that you could achieve something similar using git merge --ours or --theirs, this is a very bad idea ! These commands are explicitly revoking commits, so losing work. they actually mean "Keep our/their work, and revoke their/our work". This reverting will even be merged into other branch in the future.

Move packages between projects

Warning: please use this recipe with care! See this JIRA comment for the details.

Add a package to a project

```
git clone --recurse-submodules https://:@gitlab.cern.ch:8443/lhcb/TheProject.git
cd TheProject
git checkout -b ${USER}/MyNewPackage
# copy the files
git add Hat/MyNewPackage
git commit -m 'add package Calibration/Pi0Calibration'
git push -u origin ${USER}/MyNewPackage
```

where you replace `TheProject` and `Hat/MyNewPackage` with the appropriate values

Port changes from an SVN checkout to Git

If the project you work on moved to Git, and you didn't have the chance to "svn commit" your changes before the write access to SVN was closed, this is how you can take your changes from your SVN checkout and port them to a Git clone.

Let's imagine we were working on the package *Hat/MyPackage* from the project *MyProject*, so probably you have a directory called `~/cmtuser/MyProjectDev_vXrY/Hat/MyPackage`.

```
# ensure we are up to date
cd ~/cmtuser/MyProjectDev_vXrY
svn update
cd ..
# rename the old checkout to leave place for a new one
mv MyProjectDev_vXrY MyProjectDev_vXrY.svn
# initialize the local project to work with Git
lb-dev MyProject/vXrY
cd MyProjectDev_vXrY
git lb-use MyProject
git lb-checkout MyProject/master Hat/MyPackage
# import the changes
( cd ../MyProjectDev_vXrY.svn/Hat/MyPackage && svn diff ) | ( cd Hat/MyPackage && patch -p0 )
# commit the changes
git add Hat/MyPackage
git commit -m 'my changes'
```

At this point you can continue your development as described above.

WARNING: Please, ensure that all your changes have been ported before removing the old SVN checkout.

Formatting code for LHCb

As discussed at the 11th LHCb Computing Workshop and detailed in a Core Software Meeting, contributions to LHCb code have to be formatted according to common LHCb style. To avoid issues with different interpretations of the style, the rules are applied using the automatic tools `clang-format` (for C++) and `YAPF` (for Python). A helper command (`lb-format`) is available to simplify the interaction with the low level tools.

Formatting individual files

The simplest way to format a C++ or Python file is to call `lb-format` on them:

```
cd <Project>
lb-format MyPkg/src/MySource.cpp MyPkg/python/MyPkg/SomePython.py
```

`lb-format` ignores unsupported file types (printing a warning), so it's safe to call it with something like:

```
cd <Project>
git ls-files | xargs -r lb-format
find MyPkg -type f | xargs -r lb-format
```

Formatting only files modified wrt to a reference branch

When working on large projects, running a no-op formatting on all files takes a lot of time, so it's useful to run it only on the files you are working on.

Assuming you are working on a branch that you want to merge into `origin/master`, you can do something like:

```
cd <Project>
lb-format --format-patch - origin/master | git am
```

for files already committed, or

```
cd <Project>
git diff --name-only --diff-filter=MA | xargs -r lb-format
```

for files not committed yet.

Troubleshooting

Merge conflict in gitlab

You have pushed your commits to gitlab into a branch, you try to merge with the master or some other branch, and gitlab tells you there is a merge conflict.

The currently easiest way to see where the merge conflict is:

```
git clone --recurse-submodules ssh://git@gitlab.cern.ch:7999/lhcb/<Project>.git
cd <Project>
git checkout <your branch>
git merge <target branch>
<solve conflict> (Remove every >>>>> or <<<<<< )
git commit
git push origin <your branch>
```

Go back to gitlab and check if you can merge now.

Conflict after the global re-format of the project

This is a pretty rare case... I'll fill the blank ASAP.

Warning about push.default not being set

When calling `git push` from a CentOS7 machine, you may get this warning:

```
warning: push.default is unset; its implicit value is changing in
Git 2.0 from 'matching' to 'simple'. To squelch this message
and maintain the current behavior after the default changes, use:

  git config --global push.default matching

To squelch this message and adopt the new behavior now, use:

  git config --global push.default simple

See 'git help config' and search for 'push.default' for further information.
(the 'simple' mode was introduced in Git 1.7.11. Use the similar mode
'current' instead of 'simple' if you sometimes use older versions of Git)
```

To fix it you need to call `git config --global push.default xyz`, as described in #Prerequisites.

Warning about not being able to push to the repository

There are three ways to access the git repostory, for example for the Rec package they are:

Access	Git URL
Kerberos	`https://:@gitlab.cern.ch:8443/lhcb/Rec.git`
HTTPS	`https://gitlab.cern.ch/lhcb/Rec.git`
ssh	`ssh://git@gitlab.cern.ch:7999/lhcb/Rec.git`

The trailing `.git` is not always necessary but push errors can occur due to the redirection from the url w/o `.git` to the complete one. Ensure the `.git` is present with `git remote -v`.

A remote repository url can in anyway be changed with

```
git remote set-url Rec <new url>
```

The command `git lb-use project` sets the Kerberos method, so anyone with access to their CERN AFS area should be able to push changes to gitlab.cern.ch. If you are working outside CERN the ssh access may be a better option, you must first register your ssh key with CERN (see https://cern.service-now.com/service-portal/article.do?n=KB0003136 for details).

To either set or reset the remote site use the command

```
git remote -v
```

to view the current remote repositories. Then

```
git remote set-url Rec ssh://git@gitlab.cern.ch:7999/lhcb/Rec.git
```

to adjust a repository or

```
git remote add -f Brunel ssh://git@gitlab.cern.ch:7999/lhcb/Brunel.git
```

to add a new repository. The official repository links can be found on the gitlab.cern.ch home page for each package.

You can globally switch between ssh and Kerberos authentication with the following entry in your `~/.gitconfig` file

```
[url "ssh://git@gitlab.cern.ch:7999"]
  insteadOf = https://:@gitlab.cern.ch:8443
```

a downside is that with Kerberos, for public repositories, authentication is skipped for pull actions. With ssh one needs to unlock the ssh keyring for pushing and pulling. An alternative is to finetune

```
[url "ssh://git@gitlab.cern.ch:7999"]
  pushInsteadOf = https://:@gitlab.cern.ch:8443
```

which globally replaces all gitlab.cern.ch access from Kerberos to ssh for all push actions but leaves pull actions unchanged.

Dealing with submodules when changing branches (incl. checkout, pull, merge, cherry-pick)

We are using git submodules. The above `git clone --recurse-submodules` mostly takes care of that, except for changing branches with `git checkout` for branches that use different versions of a submodule. In this case, updating the submodule manually can be done with

```
git submodule update --recursive
```

`git submodule update --recursive` can also be run when a project was cloned without the `--recurse-submodules` option.

Known Issues

- `git lb-push` does not handle correctly binary files
- after a `git lb-push Project my-branch` you have to call `git lb-checkout Project/my-branch My/Package` to continue working on the same branch

-- MarcoClemencic - 2016-04-25, MarcoCattaneo - 2017-07-10

This topic: LHCb > Git4LHCb
Topic revision: r78 - 2021-01-19 - MarcoClemencic