Table of Contents

Using Subversion (svn) in LHCb

The LHCb code used to kept in a Subversion repository. It has now been migrated to Git

The information in this page is kept for historical reasons. For current usage, refer to the Git4LHCb page

This set of guidelines about using SVN is specifically for **Users** and **Developers** of LHCb code.

- For the definition of these roles see SVNUsageModel
- If you are a **release manager**, specific instructions are given on the PrepareProjectRelease twiki
- If you are a **librarian** instructions are given on the SubversionSupport twiki

Information:

- These instructions do not apply to the lhcbdocs repository, only the lhcb code repository.
- For further instruction also see at Marco's svn tutorial and the FAQ.

Table of Contents

SVNUsageModel

For a description of how SVN is used in LHCb and overcoming specific problems see: SVNUsageModel.

Tools and terms

Term	Explanation
the head	The most recent software with all committed modifications. Also known as the trunk in svn.
branch	A development offshoot which is not compatible with the head and must be handled carefully.
a tag	A persistent coherent snapshot of software at a given point in time. A versioned piece of software indicated by the tag name.
an official tag	A tag with a name of the form `vXrY`.
a user tag	A tag with the form `<user>_yyyymmdd`
tagging	within the release system, this is the action of creating a tag or set of coherent tags.
tag collecting	The collecting of changesets and modifications which are expected to go into a tag. Tag collecting is not tagging.
the tag collector	a website interface to a database where revisions are collected before official tags are made

Tool	Explanation	Example
svn info	Find information about a local checked out package such as the last person to modify it and the svn path	`svn info`
getpack	Checkout from the repository by package name, no need to know the full path to the repository.	`getpack -rH LHCbSys head`
tag_package	Create a tag of a package. Usually this means copying from one svn location to another	`tag_package MyHat/MyPackage me_20110907`
branch_package	Create a branch tag of a package. Usually this means copying from one svn location to another	`branch_package MyHat/MyPackage -T v1r5 v1r5b`
svnCheckTags		`svnCheckTags Rec v12r0 --diffs`

	Check that the packages and tags requested in a given requirements file or project.cmt exist. Can also compare the contents with the trunk.	
svnDiffTags	Compare the contents of two tagged versions of a package or project and its dependent packages	`svnDiffTags Rec v11r0 v12r0`
svnProjectDeps	Get a list of all package or project and versions used by a given release. Use "-P" to see only the projects, use "-f Something" to show only matching regex.	`svnProjectDeps Rec v11r0`
cmpPackageWithTrunk	compare the content of a checked-out (and modified) version of the software with the trunk	`cmpPackageWithTrunk --all`
move_package	move a package from one project another	`move_package MyHat/MyPackage DestinationProject`

Tutorial and Guidelines for Users, Accessing Subversion

- Web access

- The LHCb, Gaudi and Dirac repositories are world readable and browsable via the two web interfaces and a statistics page
  - ♦ LHCb:
    - ◊ Trac: https://svnweb.cern.ch/trac/lhcb/browser
    - ◊ WebSVN: http://svnweb.cern.ch/world/wsvn/lhcb
    - ◊ Statistics: https://svnweb.cern.ch/stats/lhcb/
  - ♦ Gaudi:
    - ◊ Trac: https://svnweb.cern.ch/trac/gaudi/browser
    - ◊ WebSVN: http://svnweb.cern.ch/world/wsvn/gaudi
    - ◊ Statistics: https://svnweb.cern.ch/stats/gaudi
  - ♦ Dirac:
    - ◊ Trac: https://svnweb.cern.ch/trac/dirac/browser
    - ◊ WebSVN: http://svnweb.cern.ch/world/wsvn/dirac
    - ◊ Statistics: https://svnweb.cern.ch/stats/dirac

The layout of the repository is based on the rules outlined in the Gaudi twiki.

- Installing the Subversion client

The Subversion command line client `svn` is available in all Linux distributions with names like `subversion` or `subversion-client`. It is available by default on `lxplus5` nodes, but it might not be installed on other machines, but you can easily find on the web how to install it for your distribution.

For Windows, MacOS (and for Linux distributions that do not have it packaged) you can find the sources and pre-compiled distributions linked from the official Subversion web site.

Several graphic clients to Subversion are available too. RapidSVN is a half decent free application and available on many platforms.

On the LHCb Windows Terminal Server we install the CollabNet command line client and the TortoiseSVN graphic client (a command line client is mandatory to be able to use `getpack`).

- Setting up read access

Detailed instructions about how to configure the access to the Subversion repository are provided by IT in the SVN How-To. You can find instructions for Linux and Windows.

Note: To enable the ssh tunnel for the command line client on Windows (URLs of the form `svn+ssh://`), you may need to add the line

```
ssh = $SVN_RSH "C:/Program Files/TortoiseSVN/bin/TortoisePlink.exe"
```

to the `[tunnels]` section of the subversion client configuration file (either the user or the global one), see the Subversion book for details.

- Connecting with a different user name

For windows, it is quite simple. You just have to change the autologin name in the "svn.cern.ch" session. You can do it by loading the session in putty and change the username in the Connection->Data section as described in the CERN SVN How-To.

For Linux/MacOSX, this is a little bit different because on these platforms the username is always forwarded to the ssh connection. In order to change the username that ssh is using, you have to modify or create the ssh configuration file ~/.ssh/config. In that file, the entry for svn.cern.ch should look like:

```
Host svn.cern.ch svn
   User myusername
   Protocol 2
   ForwardX11 no
   IdentityFile ~/.ssh/id_rsa
Host isscvs.cern.ch
   User myusername
   Protocol 2
   ForwardX11 no
   IdentityFile ~/.ssh/id_rsa
```

where "myusername" is the name you want to use for the ssh authentication. Note that the entry for `isscvs.cern.ch` is needed since the latest getpack also queries this server.

To be noted that this procedure for the setting of the user name prevents the explicit embedding of the user name in the URL of the repository. There is a problem in the release areas and distribution kits if a given user name is hard-coded in the URL. There should be no explicit user name.

- Kerberos authentication (Linux/

It is possible to use the Kerberos V token to connect to the Subversion server instead of the ssh key, simply modifying the content of the .ssh/config file as

```
Host svn.cern.ch
   User myusername
   GSSAPIAuthentication yes
   GSSAPIDelegateCredentials yes
   Protocol 2
   ForwardX11 no
   IdentityFile ~/.ssh/id_rsa
Host isscvs.cern.ch
   User myusername
   GSSAPIAuthentication yes
   GSSAPIDelegateCredentials yes
   Protocol 2
   ForwardX11 no
   IdentityFile ~/.ssh/id_rsa
```

as explained in the SVN How-To.

- getpack

After these settings, you are able to get the code (`check out`) from the repository:

```
svn co svn+ssh://svn.cern.ch/reps/lhcb/AProject/trunk/MyPackage

svn co svn+ssh://svn.cern.ch/reps/lhcb/AProject/tags/MyPackage/v2r3 MyPackage
```

on Linux, MacOSX or Windows, however, this is not however the usual way we check out packages in LHCb.

The tool `getpack` simplifies users' life taking care of details like the presence of several repositories and the project that contains the package. It is also able to do a recursive checkout according to the CMT dependency.

Note that to be able to use `getpack` you must be able to connect correctly to the Subversion repository without being asked for the password, so be sure that you have the correct configuration explained above.

For backward compatibility with the pre-svn version of `getpack`, if the environment variable `GETPACK_USER` is defined, getpack will embed the value of that variable in the URL when checking out the code. Since it is not a good idea, check that you do not set that variable and use the above instructions to use the correct username when accessing the repository.

- Archaeology, Tracking down changes between versions

See SVNCodeArchaeology.

Tutorial and Guidelines for Developers

Information:

- A **developer** is anyone who writes into the repository, so first you need to get write access (below).
- Be aware that each project has a release manager who is responsible for the maintainance of the project as a whole. It is best to discuss with the release manager of your project if you are planning major changes, discover a major bug or wish to add a new package. The release manager can also answer many of the questions you may have about svn, branch packages for you, create new packages for you, etc. in case you are not so confident with SVN. The release manager is also the person you will receive an email from if you break some production code.
- Be aware that each existing packages has a set of authors and developers who maintain the package. It is best to discuss with them if you are planning changes in that package, or discover a bug.

If instead you are looking for guidelines on how to do your development efficiently see HLTDevelopersChecklist, and/or discuss with your own release manager in advance of doing anything.

We encourage you to consider TestDrivenDevelopment using the QM-Test framework provided.

- Getting write access

To get write access to LHCb Subversion repository, go to the e-group lhcb-svn-writers and register the account you want to use when accessing the Subversion repository. A mail will be sent automatically to the librarians that will add you to the group. Once your subscription is approved *it will take few hours before you can actually commit* (the privileges are synchronized 3 times per day).

Note that when you register to the e-group lhcb-svn-writers, you will receive the emails sent on the lhcb-core-soft mailing list.

- Creating a new package or a new project

If you wish to create a new package and add it to Subversion, follow the guidelines in CreateNewPackageSVN. You can also discuss with your release manager who may do this step for you. Creation of new projects should not normally be done by developers; the necessity for a new project should be discussed and agreed at the Physics Applications Coordination (PAC) meeting. Instructions for creating a new project can be found here

- Modifying a package: Types of modification

We have three types of modification, which match the three levels of versioning in official tags, **vXrYpZ**:

- **Major**:
  - A change motivated by some major problem or improvement
  - Creating or deprecating a whole package
  - removing something from an interface, baseclass, or public inline method
  - Adding to or removing something from the DST format
  - Any non-backwards compatible change
  - Fixing a major bug which was a real showstopper
  - Changes the way a user interacts with this package
  - Major changes should be discussed with the release manager, and in the case of production software they should be presented at PAC.
- **Minor**:
  - A change motivated by improving functionality, adding functionality or improving an existing method
  - A significant change, a cosmetic change, or a small improvement
  - Code re-arrangement, juggling of code between methods, adding new private methods
  - Fixing a bug which was possibly never exposed in production
  - Does not significantly affect the users of this package or interface
  - Needs only cursory discussion with release manager or other developers
- **Patch**:
  - A change motivated by fixing a small issue in the package
  - An insignificant change, like changing the release notes or fixing a compiler warning.
  - A hot-fix of a bug which affects the production
  - A back-port of an existing bugfix to an earlier software release
  - Doesn't affect the users of this package at all
  - In the case of back-porting onto a branch, the release manager should be consulted.

For any bug which affects the production, creating a savannah bug report is considered best-practice.

- Working directories

You should only ever develop on top of:

- The HEAD, "trunk", which is the latest of all changes
- A recognized branch

Branches can be created for you by your release manager. They are used mostly to patch old software, back-porting changes to previous release stacks.

Working on top of a tag, modifying contents of a tag is forbidden. Ask for a branch or switch to the head.

- If you are not working with the HEAD revision of a package (i.e. you checked out and modified a tagged version), switch to a directory where you will be able to commit (equivalent of the old **cvs update -A** preparation before committing):
  - Since TAGGED directories are not supposed to be altered, when committing a change to a package you should, **first**, switch to the trunk directory. To do so you should
    - Issue the command `svn info`. That will tell you the actual path of your package:

```
svn info
Path: .
URL: svn+ssh://svn.cern.ch/reps/lhcb/Rec/tags/Tf/PatAlgorithms/v3r29
```

    - If the URL is of the form above (i.e. it contains the `tags` subdirectory and a version number), it means you are working against a tagged version. This needs to be altered in order to include **trunk** in place of **tags**. You also need to **remove** the version number at the end of the URL. Do this with the `svn switch` command:

```
svn switch svn+ssh://svn.cern.ch/reps/lhcb/Rec/trunk/Tf/PatAlgorithms
```

  - An alternative is to call `getpack` again on the same package asking for version `trunk` from the parent directory:

```
getpack Tf/PatAlgorithms trunk
```

- Guidelines for SVN commit

1. **Discuss** with the package owner and/or release manager:
   - Before changing any released package, you should discuss the changes with the package owner (whose name usually appears in the cmt/requirements and doc/release.notes files).
   - It is usually a good idea to inform the appropriate mailing list of the planned changes if they are major or affect a production application, you may also want to come along to PAC.
   - If it is a major bugfix which affected a production application, it is a good idea to make a savannah task, and assign it to yourself, even if it is already fixed.
2. **Document** each modification, before committing in the file `doc/release.notes`
   - The release.notes looks something like:

```
! 2009-12-15 - Marco Clemencic
 - This is the comment for the latest commit, do not touch anything below this line.

!================== GaudiConf v10r9 2008-06-04 ========================
 ! 2008-06-03 - Marco Clemencic
 - Everything up to the official tag line above documents commits which were include
 - do not touch anything below the tag line, and do not make your own tag lines.
```

     - Lines with official version numbers are made by the release manager. You shouldn't edit them or make them yourself.
     - When using the LHCb-flavour of emacs, the release.notes is templated. Just hit `Insert` to start your release comment
   - Be verbose enough in your release notes so that others can understand what has changed, "minor fixes" is not good enough.
     - But no need to document the fix to the fix that you committed a few hours ago, unless it will be confusing for the release manager
     - Please keep the lines short, max 80 characters per line, carriage return to a new line if necessary.
3. **Commit often**, but **commit only code that compiles and runs**
   - Make sure you have compiled the package against the latest software. The code should compile **without warnings** on at least one platform

- Make sure you have run any relevant qmtests and it's a good idea to write your own new qmtests if a major bug has been fixed or a major change has been made.

4. **Check what you are about to commit against the repository**:

```
svn status -u
```

This will list the files that have changes with respect to the versions in the repository. The type of difference will be marked with one or more letters. The most common are: **M** for changes in your copy, **?** for new files, ***** for files that have a new version in the repository, **C** if there are conflicts between your changes and the ones in the repository, **!** for files removed. The complete list of letters can be found in the SVN documentation. Once you're sure of your changes, you can call `svn update` (or `svn up`) to synchronize the working copy with the changes in repository.

- Make sure that there is no **C** in front of any file: **C** in front of a file means that several people are working on the same package so be careful: SVN has tried to merge your modifications with the changes already done and it has NOT succeeded. Check the merged files carefully: SVN marks with `>>>>>>` and `<<<<<<` the regions that have incompatible modifications. You **must** fix these (and remove the `<<<<<<` and `>>>>>>` before committing). Merged files only appear when you do `svn update`, but you then need to be more careful of what you do.
- Check carefully, and Eliminate or ignore any **?** in source directories: These are files that you created newly and that are not in the repository. If you wish to save them in SVN, you have to

  ```
  svn add TheFileToAdd
  ```

  but do not add any file in the cmt/ directory, or any copy of a file that should not be kept (e.g. files whose name ends with "~")
- Tell SVN to remove from the repository the files you have deleted if any: files that were marked as **lost** are files that you removed. If you wish to remove them from SVN, issue the comand

  ```
  svn rm TheFileToRemove
  ```

5. **Check for conflicts**:

```
svn update
```

- ◊ Any **C** should have been replaced by **M**
  ◊ Any **?** should have been replaced by **A**, or just ignored, you may have several ? left
  ◊ Any **!** should have been replaced by **R** 1 Check for, and fix conflicts before committing:
  ◊ If there are any remaining conflicts, discuss with your release manager.

6. **Commit**, always with a meaningful comment

```
svn commit
```

Always add a meaningful comment using the editor that is automatically started.
You can by-pass the editor with the `-m` switch:

```
svn commit -m "some brief and meaningful comment"
```

P.S. "minor changes" is still not good enough, and "some brief and meaningful comment" is also not such a funny joke. The comment can be the same as the release notes, often that helps the release manager

7. **Use the LHCbTagCollector**
   - If the package is part of a production project, like Rec, Phys, Analysis etc. then you need to add it to the LHCbTagCollector so that the release manager will know what to do with your changes for the release.

- This adds into a database the revision number that was printed by `svn` after the commit.
- **You do not need to provide a "user tag" as was done for CVS.** If you do, for some strange reason, want to provide your own user tag, please follow the guidelines for tags
- You will select the type of modification from amongst the options, Major, Minor, Patch, as defined above. The release manager will then pick an appropriate tag name when it comes to the release
- Q: do I really need to use the tag collector. A: It is best-practice, see the explanation here: FAQ

8. **Check the nightlies**
   - We build all our software every night to check for problems
   - The next day, check that your commit has not broken other packages
   - Check the nightly builds the day after your commit
   - Fix any packages that were broken by your commit (or ask the package owner to do it), and fix any compilation warnings (on any platform) or QMTest failures caused by your commit

- **It didn't work?**
  - Take a look at the FAQ, and email your release manager.

- Tags and Branches

Tagging in Subversion, if you are used to another version control system, may appear strange. The concept of *tag* and *branch* are not natively available in Subversion, but they can be easily emulated using a convention, thanks to the ability of Subversion of remembering the history of the whole repository (including directories).

The standard convention used in almost all the projects hosted on Subversion (a notable exception being CMT) is based on the presence of three directories called `trunk`, `tags` and `branches`. The `trunk` contains the main line of development of your code. `tags` and `branches` contains copies of the `trunk` with symbolic names (like `v1r2` or `2.3.4`). There is no technical difference between `tags` and `branches`, but the convention is that you are not supposed to commit changes to a file in the tags directory.

The convention used for the repository layout in LHCb is described in detail in GaudiSVNRepository.

- Guidelines for tags

Users and Developers do not need to make tags in SVN, since the revision number provides the same information, but you can, if necessary, or if you are the release manager.

- Tagging in subversion is a copy from one place to another in the file system
- Do **not** use an official tag of the form `vxry`
- ... i.e. Only release managers are supposed to make official tags of the form **vXrY**
- Developer and user tags are only useful in certain nontrivial cases, and are of the form `<username>_yyyymmdd`

The guidelines for tagging in LHCb were presented on 18th June 2008 (33rd LHCb software week)

- It is recommended to use a tag of the form `<username>_<yyyymmdd>` where `<yyyymmdd>` is the date of the commit. If you have to create more than one user tag during the day, you can append a lowercase letter (starting from `a`) to distinguish them. To tag the current version in the repository do:

```
tag_package Rec/Brunel marcocle_20091215
```

You can also use the low level command (note: you can use `svnpath` to get the URL of the package) to tag the trunk

```
svn cp svn+ssh://svn.cern.ch/reps/lhcb/Brunel/trunk/Rec/Brunel svn+ssh://svn.cern.ch/reps/lhcb/Br
```

or to tag according to what's presently in your directory, do in your directory (e.g. `Rec/Brunel`):

```
svn cp . svn+ssh://svn.cern.ch/reps/lhcb/Brunel/tags/Rec/Brunel/marcocle_20091215
```

See GaudiSVNRepository for a description of the tags conventions in the LHCb Subversion repository.

- How to alter a tag

It is not permitted to change the content of an existing tag. Modifying the content of a tag is **always** a **bad policy**.

If you have made a "user tag" of the form <username>_<yyyymmdd>= then you can just create a new tag with the new fix.

To remove the knowledge of the previous tag you have to delete the tag you created with something like:

```
svn rm svn+ssh://svn.cern.ch/reps/lhcb/Brunel/tags/Rec/Brunel/marcocle_20091215
```

then it is possible, but not advisable to re-create the tag with the instructions in the previous section.

- Working with branches

It's a good idea to let the release manager know when you want to do some changes on a branch. The release manager can create the branches for you, to reduce errors.

Creating a branch is not much different from creating a tag, except that the destination URL of the copy must contain `branches` instead of `tag` (the source URL can be the trunk or another tag) and the name of the branch should end with a 'b', like `v1r2b` or `v3b`. For example:

```
svn cp svn+ssh://svn.cern.ch/reps/lhcb/Gauss/tags/Gen/EvtGen/v11r10 svn+ssh://svn.cern.ch/reps/lh
```

or using the shortcut

```
branch_package Gen/EvtGen -T v11r10 v11r10b
```

To work on the branch, just use `getpack` to check it out (requires LbScripts >= v6r0):

```
getpack Gen/EvtGen v11r10b
```

and work as if you checked out the trunk. Note that is your branch name does not end with a 'b', you will have to add the option `--branch` when calling `getpack`.

Once you are done editing, commit as usual:

```
svn commit -m "committing a modification to the head of the Gen/EvtGen v11r10b branch"
```

To tag a branch, use the `-B` option of `tag_package` (`tag_package <package> -B <branchName> <tagName>`):

```
tag_package Gen/EvtGen -B v11r10b v11r10p1
```

- Merging changes across banches

Often, when working with branches, you need to merge in the trunk changes that are on a branch or the other way around.

Let's take the example of a fix that has to be back-ported from the trunk to an old version. Let's assume that the changes you want to back-port are the commits in the trunk after revision 100 up to 110 (included) plus the commit of revision 115. What you have to do to merge is:

```
cd Gen/EvtGen
svn merge -r 100:110 -c 115 svn+ssh://svn.cern.ch/reps/lhcb/Gauss/trunk/Gen/EvtGen
```

You can also use the tool `svnpath` to get URL for the package (see `svnpath --help`):

```
cd Gen/EvtGen
svn merge -r 100:110 -c 115 `svnpath Gen/EvtGen`
```

The exact meaning of `-r` and `-c` are described in the help page `svn merge -h`.

SVN may ask you what to do in case there is a conflict. You can:

- see the differences (df)
- edit the conflict (e)
- keep the version in your working copy (mc, mine-conflict)
- use the version in the repository (tc, theirs-conflict)
- flag the conflict as resolved after the edit (r)
- and more (s, show all options)

Once the merge is completed, just commit the merged version with `svn commit`.

If the merge has to be done the other way around, i.e. from the branch to the trunk, swap the trunk and the branch:

```
getpack Gen/EvtGen trunk
cd Get/EvetGen
svn merge `svnpath Gen/EvtGen v11r10b`
svn commit
```

Note: if you do not specify the revisions to be merged, it will pick up all the changes that have been applied since the branch (or the last merge), so it is better to use this feature when merging from a branch into the trunk.

- Moving a package from one project to another

To migrate a package from one project to another, the trunk, the tags and the branches of the package have to be moved and the packages property have to be updated. Everything should be done in a single commit operation to avoid inconsistencies. We have provided an easy to use tool for the task: `move_package`. The command line looks like this:

```
move_package MyHat/MyPackage DestinationProject
```

Don't forget to edit also the `xxSys/cmt/requirements` and `xxSys/CMakeLists.txt` files of the source and destination projects

Subversion documentation

- Basic SVN commands

See SVN quick reference card and SVN cheat sheet

A summary of SVN commands for CVS users can be found in the Gaudi twiki.

Some simple instructions to migrate an existing CVS checkout to SVN are available at HowToMigrateACvsCheckoutToSvn.

- Subversion manual

The complete manual is available from http://svnbook.red-bean.com

- Instructions for LHCb librarians

Instructions for LHCb Subversion librarians are available at SubversionSupport.

-- MarcoClemencic - 15-Dec-2009

This topic: LHCb > SVNUsageGuidelines
Topic revision: r50 - 2017-01-30 - MarcoCattaneo