LHCb migration from Subversion to Git

To cite this article: M Clemencic et al 2017 J. Phys.: Conf. Ser. 898 072024

View the article online for updates and enhancements.
LHCb migration from Subversion to Git

M Clemencic, B Couturier, J Closier and M Cattaneo
CERN, CH-1211 Geneva 23, Switzerland
E-mail: marco.clemencic@cern.ch

Abstract. Due to user demand and to support new development workflows based on code review and multiple development streams, LHCb decided to port the source code management from Subversion to Git, using the CERN GitLab hosting service. Although tools exist for this kind of migration, LHCb specificities and development models required careful planning of the migration, development of migration tools, changes to the development model, and redefinition of the release procedures. Moreover we had to support a hybrid situation with some software projects hosted in Git and others still in Subversion, or even branches of one projects hosted in different systems.

We present the way we addressed the special LHCb requirements, the technical details of migrating large non standard Subversion repositories, and how we managed to smoothly migrate the software projects following the schedule of each project manager.

1. Introduction
With Git\[1\] gaining more and more popularity among High Energy Physics developers, and given requests for Git support in the LHCb development workflow, the LHCb Core Software Group decided to migrate our source code repositories from the existing CERN Subversion\[2\] hosting service to the new CERN Git hosting service based on GitLab\[3\].

The layout of the LHCb Subversion repository\[4\] required a special migration strategy, and the development tools and workflow required adaptation.

2. Requirements
The main requirements for migration of the code were formalized by the LHCb Core Software Group in the following list:

- smooth transition, allowing coexistence of the two version control systems
- integration with the LHCb Nightly Builds System\[5\]
- support for development with partial checkouts
- documentation of new development workflows

3. Transition
To allow a smooth transition we started by producing Git mirrors for the software projects in Subversion. Since in the Subversion layout used in the LHCb environment the coupling between folders and released projects was loose (i.e. the release tags were not equivalent to a revision of a well defined folder in the tree), the mirrors were initially created as a time line of releases,
with one Git commit per released version, where the content of the commit was a dump of the tagged code.

After a clean up of each software project trunk directory, to make its content was equivalent the HEAD version of it (as understood by our wrapper around the svn checkout command), we bound the Git mirror to the Subversion trunk directory via the tool git-svn[6]. Since the Git repository was not created using git-svn, the binding was achieved tweaking the repository configuration to inject the right git-svn metadata.

Once the Git-Subversion links were established, we activated an automatic cross synchronization procedure where commits in Subversion were automatically pushed to the Git mirror hosted on the CERN GitLab instance, while merge requests (see section 4) created in GitLab, targeting the master branch of the repository, were automatically applied and committed to the Subversion repository. The automatic synchronization allowed early adopters of the Git based workflow to collaborate seamlessly with developers still using Subversion. At this stage, releases of the software projects could be prepared either in Subversion or GitLab.

The last phase of the transition was to declare the Git repository to be the reference repository and block write access to Subversion on a project-by-project basis. This final step was transparent for users that moved already to the Git based workflow.

4. Development and Release Workflows

The development workflow adopted with the Subversion repositories implied that developers commit their changes directly to the trunk folders of the projects. Afterwards they were to notify the project maintainers via a dedicated web site (the Tag Collector[7]) where they could record some explanation of the change they made.

LHCb software projects are organized in packages, and for Subversion we adopted a tag policy for which each package was tagged independently and the project release was defined by a list of packages and their versions (thus the loose coupling of the revision of the trunk and the releases code). This tagging policy implied that entries in the Tag Collector had to be grouped by package and that the project maintainers had to tag each modified package (updating their version numbers) before the release.

One of the reason of the popularity of Git is that it allows many different development workflows and tools like the GitLab hosting service make it possible to easily and efficiently implement workflows that include, for example, code review (which was one of arguments in favour of the LHCb migration to Git).

The workflow we adopted for Git is based on topic branches (short-lived branches used to group changes aiming the introduction of a new feature or a bug-fix) that are proposed for integration in the mainstream branch (the master branch) via the so called GitLab merge requests. Merge requests provide a staging area for the changes to be integrated, where they can be documented, reviewed and validated before they are accepted. The use of merge requests basically remove the need for the home made Tag Collector, and prevents situations where where, to prepare a release, some changes, not meant to be released yet, had to be reverted in the trunk, just to be reintroduced shortly after the software project release was out.

Since Git does not allow tags on specific folders of the project, users cannot tag anymore individual packages, but on the other hand it means that there is no need to update the version numbers of each package, and the tag of the release of a project is reduced to a couple of click on a web interface. The drop of version numbers for the packages might seem a reduction of functionality in the workflow, but their use was was bringing additional complications, for example it was difficult to find which version of a package was used for a release of a software project.
5. Integration with the LHCb Nightly Builds System

The Nightly Builds System is used in LHCb to perform continuous integration builds and tests, so it required adaptation to get the versions of the software projects to build either from Subversion or Git.

With the software projects hosted in Subversion we were used to test the changes developers committed directly to the project trunk folder, but the change of the development workflow implies that the master branch does not include the proposed changes until they are validated. We thus implemented a checkout mechanism by which the version of the software being tested corresponds to the master branch with applied all the relevant merge requests. Since often the code being build does not correspond directly to a commit in the Git repository, we created a special fork of the official Git repositories, where we record the actual version of the code build day by day in the nightly builds.

6. Development with Partial Checkouts (Satellite Projects)

LHCb developers, thanks to the build tools adopted[8, 9] and the possibility of checking out subdirectories of a Subversion repository, have always been able to build a small local project (or satellite project) that was based on a released version of an official software project, and was overriding some of the released packages (e.g. to pick up bugfixes) or extending it with new packages (e.g. to develop new feature for later inclusion in the official project). This way of developing is very effective in several circumstances and had to be preserved in the migration to Git.

The main constraining factor to the partial checkout needed for satellite projects is that the subdirectories to checkout may come from different software projects, so from different repositories. The existing Git tools to work with a reduced number of directories (sparse checkout) or with code imported from multiple repositories (submodules and subtrees) cannot work out of the box in our case. We thus decided to develop three small Git subcommands working on the same basic principles of Git subtrees[10]:

- \texttt{git lb-use}: add the remote repository of a software project to the local Git repository (similar to \texttt{git remote add})
- \texttt{git lb-checkout}: checkout a subdirectory from a given version of a software project (locally added with \texttt{git lb-use}), and store some metadata
- \texttt{git lb-push}: leveraging on the metadata recorded by \texttt{git lb-checkout}, extract from the local history the changes relevant to a specified software project, apply them to a new branch of the official project repository and push it to the remote repository (for later integration via a merge request)

The custom Git subcommands are not meant to cover all the source code management features of Git, but they are very effective in short-lived developments, like bug-fixes, where a developer creates a satellite project directory, checks out the handful of packages to be overridden, makes a few changes, pushes them to a new branch, creates the merge request, and finally removes the satellite project directory. Nevertheless, despite their limitations, several developers are using these satellite projects for long-lived developments, and share them with other developers for validation testing and contribution.

7. Conclusions

The migration of LHCb Software Projects from Subversion to Git started in March 2015 with the first mirrors of a few projects. Early 2016, we defined development workflows and prepared the required custom tools. At present, only a few special cases have not migrated yet.

Allowing users to develop with Git increased productivity (number of contributions, new projects) and code quality (testing and reviews).
References