Contents

Git tutorial **2**

Introduction . . . 2

Quick start guide . . . 2

Key concepts in Git . . . 3

Distributed, but local . . . 3

Where is my data? . . . 4

Commit . . . 5

Branches . . . 6

Common operations . . . 6

Creating a Git repository . . . 7

Getting a clone of the repository ATLAS Pixel DAQ . . . 7

Configuring Git . . . 8

What's going on in my workspace: `git status` . . . 8

Adding, removing and moving content . . . 9

Committing changes . . . 12

Sharing your changes with others . . . 14

Getting changes from others . . . 15

Branching . . . 16

Merging . . . 20

Conflict resolution . . . 22

Ignoring files . . . 24

Undoing things . . . 25

Stash . . . 25

Consulting the history . . . 25

Patching . . . 26

Work flow with Git . . . 26

Using git-flow . . . 30

External tools . . . 33

GUIs . . . 33

Git-new-directory . . . 33

Git tutorial

Rafael Tedin Alvarez, 2013

Introduction

Git is a distributed version control system developed by Linux Torvalds. Almost everything happens locally, so contributors can work more independently and most of the time even offline. Distributed means that no central repository is required and each team member has a **copy** of the complete repository. This encourages and makes possible that subgroups work independently from others.

This tutorial gives only a brief overview of Git. It is probably enough to get you started, but the best place to get all information about Git is probably git-scm.com. Some very helpful cheat sheets out there are for instance this one or that one.

There are also some mailing lists that you should know:

GIT-users-atlaspixeldaq : Members of this list have access to the ATLAS Pixel DAQ Git repository. If you have any questions, you can address them to this list.

git-notifications-atlaspixeldaq : Commit notifications are sent to this list.

This tutorial is organised as follows. Firstly, a quick start guide for using the ATLAS Pixel DAQ repository is provided for those that already have some knowledge of Git. Some Git key concepts are then explained. After that the most common operations in Git are shown along with the commands needed to fulfil those operations. Then the proposed Git work flow for the ATLAS Pixel DAQ is presented. This work flow will govern the way people work and cooperate using Git. Finally, some helpful external tools are listed.

Quick start guide

For getting a ready-to-use copy of the ATLAS Pixel DAQ repository you have to do the following:

1. Clone the repository.

   ```
   $ git clone https://git.cern.ch/reps/atlaspixeldaq
   ```

2. Configure your personal information giving your name and e-mail address. Use the CERN preferred way for your name and e-mail address:

```
$ cd atlaspixeldaq
$ git config user.name "FirstName LastName"
$ git config user.email "FirstName.LastName@cern.ch"
```

3. After that, if you want to use git-flow (recommended), install it (see the note at the end of the section) and then issue the following command. The defaults are fine, so you just have to hit ENTER on every question

   ```
   $ git flow init
   ```

   or use the `-d` switch to automatically accept the defaults:

   ```
   $ git flow init -d
   ```

   There you go.

Note: : For installing `git-flow` a custom script is provided in the project directory `atlaspixeldaq/scripts/ext`:

```
$ cd atlaspixeldaq/scripts/ext
$ ./install-gitflow.sh
```

Key concepts in Git

You are probably familiar with one or the other version control system, most probably Subversion. In order to painless learn how to work with Git, it is strongly recommended that you do not try to map concepts of other version control systems to Git, since this does not always work and it can be source of headaches. Beware specially from similar vocabulary: do not take for granted that two commands mean the same just because they are called the same in Git or other systems.

Instead, start fresh and open-minded and read through this small list of key concepts of Git. They are common jargon in Git and used throughout the tutorial. It's important that you understand them. More concepts are going to be introduced as soon as they are needed. Again, this tutorial only scratches at the skin of Git. For a more detailed understanding of what is going on visit git-scm.com.

Distributed, but local

First of all, let's introduce a concept common to all version control systems. The location where the data structures, files, history and other data are stored is

called *repository*. This is a very important place that you want to keep secure and clean, since all your project's data is stored there.

It was common until some time ago, that the repository was kept in a server accessible by everyone. Each team member established a connection to this server and could for instance download the source code, upload some changes or update some file with the changes of other team members. Almost every operation went through this central repository.

However, if your project has many contributors, this scheme might not scale very well. The central server becomes a bottleneck. Moreover, the server represents in fact a single point of failure. If something happens to the server, all the history of the project might be lost.

To overcome these problems, the next generation of version control systems is able to use instead a distributed scheme. In this scheme, each of the team members has a copy of the repository. Now each member operates on their own *local* repository and only shares with others what needs to be shared.

This opens a whole new way of working together with others, where each member is an independent node of a net. Nevertheless, note that the central repository scheme can still be used. In this case the central repository is just a node upon which was agreed that it should have this role.

Git is a distributed version control system. When you get a copy of a repository, it really means you get a copy of everything. That in turn means, that you can almost always work locally, since you carry all the data with you. And Git relies on this most of the time. So remember, **in Git almost everything happens locally**, in your machine. Yes, even when you commit!

The ATLAS Pixel DAQ uses a central Git repository located at `https://git.cern.ch/reps/atlaspixeldaq`.

Where is my data?

Git organises the data in different places that you should know. This is important in order for you to know what you are doing and what operations you can perform on your data. A very nice cheatsheet with available commands for each possible state of your data can be found here.

Upstream repository : The upstream repository is the repository with which you share your changes (`git push`), get the changes from others (`git pull`) or simply the one from where you want to clone the repository from (`git clone`). Note that you can have several upstream repositories, since Git is a distributed version control system. For ATLAS Pixel DAQ, the main upstream repository is `https://git.cern.ch/reps/atlaspixeldaq`. The original repository you have cloned from is normally called *origin*.

Local repository : That is your local copy of the upstream repository that you have cloned. Remember that you have a complete local copy. And almost

everything happens locally. Therefore, some operations that you might expect to be applied to the upstream repository are applied instead to the local repository. In particular, when you commit changes, you commit them to the local repository. The upstream repository remains unchanged.

Index or staging area : Git does not assume what you want to commit. You have the complete control what is going in the next commit and what does not. This is what the index or staging area is for. The index is a file that tells Git what to commit. And Git keeps religiously to the index. Sometimes you have to be careful with this. Do not assume that only because you modified a previous committed file, Git is going to add these modifications to the next commit. In fact, Git will not! You have to stage this modifications (`git add`).

Workspace : The workspace or *working tree* holds the current state of all the files and directories you are working on.

Stash : This is best explained if something similar to the following ever happened to you. You started to edit some file A adding some new feature and after some time of hard work, you realise that you have to fix some bug on file B. But in order to go on with the bug fix, you need that A is kept unchanged (for testing purposes for instance). You lose all your precious work in A! Well, you could make a copy of your work in A (call it A_copy), revert the changes to A, fix the bug in B and then rename A_copy to A. Or... you could use Git. The Git stash fits perfectly your need. The stash it's a place where you can save modifications that you want to apply later, but for the moment you are concerned with something else and those modifications, that you already made, have to wait. In the example, you would just stash A (`git stash save`), work on the bug fix and then unstash A (`git stash pop`).

Commit

This is all about version control right? That is, about keeping track of the changes that a project has undergone during the past time. The way of consolidating those changes is called *to commit* (`git commit`).

In other version control systems that use a central repository scheme, consolidating a change to the repository means that every one sees this change. But in Git almost everything happens locally, also commits. You can commit as often as you want and only share your changes when you feel to do so. So, commit often and decrease the changes of loosing some precious work!

Be aware however that commits are local. As long as you don't share your changes with others or make a backup of your local repository, you only have a single copy of your work!

Branches

A branch is a duplication of the current workspace (or an other branch), so that work can be done in parallel in the parent branch (the original branch) and in the new one. Note that with this definition, the workspace can be seen as the current branch you are working on. Normally each Git repository has a main branch called *master*.

Git calls switching to a branch *to check out* (`git checkout`). So, when you check out a branch, you make this branch your current working tree or workspace.

Remember, almost everything happens locally in Git. Branches are also local objects. But in order to share modifications with others, local branches can *track* other branches of upstream repositories. If you have a branch that tracks a remote branch, you can upload modifications to the tracked branch with `git push` or download modifications from it with `git pull` (and also `git fetch`, more on that later).

From time to time, the work done in a branch might need to be applied to the parent branch or to other branches. This is called *merging*. For instance, suppose you have one branch with a file called README. You create a new branch for starting to typeset and make corrections to this file. When you are done, you want to apply the corrections you did in the new branch to the parent branch. All you have to do is to *merge* the new branch into the parent branch (`git merge`). Git is normally smart enough to figure out how to merge your changes. If it is unable to do so, there is a *merge conflict* and manual intervention is needed.

Many version control systems have support for branches. In Git the use of branches is ubiquitous because branching is designed as a very cheap process. They can be created and deleted at will. It is very common that a developer creates one or more new branches for each new feature he or she is developing. In the example above with the README file, one might be tempted to just modify the file in the original branch. But branching really helps to separate concerns and favours independent work. While you are editing the README file, you can still use almost all features of a version control system in your own branch. For example, you can commit as often as you like without colliding with others. Branches in Git are actually only pointers to the right commit.

The correct use of branches makes a sound work flow possible and easy. The team members can work independently without to have to give up a good version control.

Common operations

In this section basic Git commands are shown that should cover most of the tasks that you will encounter in the day-to-day work using Git. It does not get into the details of each command. For that try:

```
$ git help <command-name>
```

Creating a Git repository

Creating an empty, ready-to-use repository is as easy as:

```
$ mkdir my-rep
$ cd my-rep
$ git init
Initialized empty Git repository in my-rep/.git
```

As you see, all that Git does is creating a new `.git` subdirectory. This folder contains all the repository data, including configuration, commits and snapshots of all the repository's files.

More details: `git help init`.

Getting a clone of the repository ATLAS Pixel DAQ

The command for getting a clone of a upstream repository is `git clone`. You could clone the repository created in the previous section like this:

```
$ cd ..
$ git clone my-rep my-rep-clone
Cloning into «my-rep-clone»...
warning: You appear to have cloned an empty repository.
done.
Checking connectivity... done
```

Don't bother about the warning, it is not important for this example. The syntax of the `clone` command is

```
git clone <upstream-repository> [local-name]
```

`upstream-repository` : The path or URL of the upstream repository. This is mandatory.

`local-name` : The name of the directory that is created and where the repository is cloned. You can leave this out. In this case a directory with the name of the upstream repository will be created.

In case of the ATLAS Pixel DAQ, the repository is located at `https://git.cern.ch/reps/atlaspixeldaq`. So, to clone the repository to your local machine:

```
$ git clone https://git.cern.ch/reps/atlaspixeldaq
```

Now you should have a **local repository** in the `atlaspixeldaq` directory.

More details: `git help clone`.

Configuring Git

Git uses some personal information about you and adds it to some objects, for instance the commit data, so that other people can see who did what. It is important to configure this information:

```
$ cd atlaspixeldaq
$ git config user.name "FirstName LastName"
$ git config user.email "FirstName.LastName@cern.ch"
```

Use the CERN preferred naming conventions for filling this information. This configuration gets written by default in the `atlaspixeldaq/.git/config` file under `[user]`. It is therefore only applicable to the repository where you issue the `config` command. If you want that all of your repositories have this same information you can use the `--global` option:

```
$ git config --global user.name "FirstName LastName"
$ git config --global user.email "FirstName.LastName@cern.ch"
```

You can list the configuration by:

```
$ git config --list
```

More details: `git help config`, here or there.

What's going on in my workspace: `git status`

This is one of the commands that you will be using a lot to keep on the safe side. The command `git status` shows the status of the current working tree. It shows the modifications that you have made and, more important, it also gives you hints of commands that you can use for performing common tasks. Use this command before every commit!

For instance, if you create a new file called `foo`, `git status` will warn you about this modification and what you can do with it:

```
$ touch foo
$ git status
# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       foo
nothing added to commit but untracked files present (use "git add" to track)
```

If there is nothing to commit, `git status` tells you this:

```
$ git status
# On branch master
nothing to commit, working directory clean
```

Adding, removing and moving content

As the message from `git status` in the previous sections suggests, the command for adding new content to the next commit to the **local repository** in the current branch is `git add`.

```
$ cd my-rep
$ ls -l
total 0
$ touch foo
$ git status
# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       foo
nothing added to commit but untracked files present (use "git add" to track)
```

The file `foo` is untracked, which means it was not part of any previous commit. This is true since it was just created. To add the file `foo` to the staging area issue:

```
$ git add foo
```

```
$ git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file:   foo
#
```

Now Git will add this file with the next commit in the current branch (`master`). Note that `git status` tells you the changes that are going to be committed. It also tells you that if you want to remove the file from the next commit (not from the file system), that is, unstage the file, you have to issue the command:

```
$ git rm --cached foo
rm 'foo'
$ ls
foo
$ git status
# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       foo
nothing added to commit but untracked files present (use "git add" to track)
```

And now `foo` is out of the staging area and will not be committed.

Suppose now that the file `foo` is already tracked, i.e. it has been already committed:

```
$ ls
foo
$ git status
# On branch master
nothing to commit, working directory clean
```

If you want to delete this file:

```
$ rm -f foo
```

```
$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add/rm <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       deleted:    foo
#
no changes added to commit (use "git add" and/or "git commit -a")
```

The file has been deleted from the file system and Git has noticed this. But this modification was not staged so it won't get committed. `git status` tells you how to stage the change for the next commit:

```
$ git rm foo
rm 'foo'
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       deleted:    foo
#
```

Now, after a commit, the file `foo` would be deleted also in the local repository in the branch `master`. You could do all this in one step with:

```
$ git rm foo
rm 'foo'
$ ls -l
total 0
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       deleted:    foo
#
```

The `git mv` command works in a similar way. This command renames or moves files or directories. All these operations work with the staging area and `git status` tells you what is going to be committed and what you can do to undo the changes. Remember that changes only get committed if they are added to the staging area.

For further details is advisable to have a look at `git help add`, `git help rm` and `git help mv`.

Committing changes

The command for committing changes to the **local repository** is:

```
$ git commit
```

Here is a simple example. A file `foo` is created, added to the index (i.e. staged in Git's terminology), which means it is going to be included in the next commit and, finally, the file is committed.

```
$ touch foo
$ git add foo
$ git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file:   foo
#
$ git commit
```

An editor opens for editing the *mandatory* commit message. Lines starting with `#` are considered comments and thus are ignored. A summary of the commit much the same as the output of `git status` is shown in comments. Commit messages should be concise but meaningful!

After editing the commit message, the commit is performed.

```
[master 0e1b40e] Add file foo.
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 foo
```

The file is committed. Commits in Git are just pointers to snapshots of the working tree and to the immediate previous commit. The commits, as most Git objects, are referenced and uniquely identified by a SHA-1 hash. In this case `0e1b40e` are the first 7 characters of the commit hash `0e1b40e1d4c6cbf831123e146da727208fc9c079`.

A branch on the other side is just a pointer to a commit, this is why they are so cheap in Git. When committing, Git automatically moves this pointer to point to the last commit. So, for instance, the branch `master` now points to the commit `0e1b40e`, and through it, to a snapshot of the working tree, the one

which now contains a file called `foo`. Every time the user checks out this branch (switches to this branch), Git cleans the working tree (workspace) and puts in it the snapshot pointed by the commit referenced by the branch `master`.

Some caveats should be considered when committing. First, and most important of all, remember, almost everything happens locally. Also commits. This means, nobody actually has seen the previous `0e1b40e` commit. Only the user who committed it, since it is only in his local repository. This also means that only one copy of your committed data exists until you share it with others. The next section shows how to share commits.

Second, only things that are staged are committed. Thus `git add foo` is necessary not because the file `foo` is a new file, but for the very reason of including it in the commit. If you edit the file and want to commit the changes, you have to include the file:

```
$ edit foo
$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   foo
#
no changes added to commit (use "git add" and/or "git commit -a")
```

Your friend `git status` tells you this. If you want to commit the changes, use `git add`.

```
$ git add foo
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   foo
#
```

Now the changes are going to be included in the next commit. The same rationale applies to removing and moving files.

`git status` also suggests another way of including the files with changes to the next commit directly using:

```
$ git commit -a
```

It would be painful to have to add every file again and again. So for tracked files (files that were already committed before), the `-a` switch automatically includes them (with the appropriate `add`/`rm`/`mv` command) to the next commit.

The third thing to consider is that every commit needs a message. This message is very important, unless you are able to remember what you did only at looking at the hash `0e1b40e1d4c6cbf831123e146da727208fc9c079`. Certainly others won't, so a good commit message is essential for making the navigation in the repository's history possible. When issuing `git command` an editor opens where you can edit the message. The message can also be introduced inline:

```
$ git commit -m "Added file foo."
```

Other switches can be combined. This includes the changes in file `foo`, commits and sets the commit message, all in a single line.

```
$ git commit -am "Edited file foo."
```

Finally, commit often, very often! It's cheap, it's fun and does not hurt! If you messed up something in the last commit, it can even be amended:

```
$ git commit --amend
```

This switch overrides the last commit with the new one. CAUTION: do **not** do it if you already shared your changes with others!

For basic use, this should be enough, but it is strongly encouraged to have a look at `git help commit` for at least getting some idea of all options.

Sharing your changes with others

Almost everything is local, so for sharing changes, Git has to be instructed to do this explicitly. The command for that is `git push`. The basic syntax of the command is as follows:

```
git push [upstream-rep]
```

Where the optional `upstream-rep` parameter can be the name or the URL of the remote repository with which to share the changes. If you do not give this parameter, the changes are pushed to the remote branch that the current local branch is tracking.

If the local branch is not tracking any remote branch, a remote branch has to be created and the local branch must be set up to track the remote branch. This is

something you will likely come across the first time you want to push a branch to the upstream repository. For details on how to this, see the branching section or have a look at the `--set-upstream` or `-u` switch using `git help push`.

`git push` updates the remote repository (upstream repository) with the commits and other local objects created since the last time both repositories synchronised.

Again, this basic use should be enough. For more advanced operation, consult `git help push`. There are options for deleting remote branches and pushing other objects that are not pushed by default (for instance *tags* are not pushed by default).

Getting changes from others

For getting changes from others there are basically two commands: `git fetch` and `git pull`. The difference between the two commands is that `git fetch` only downloads the remote objects to the local repository, but the changes are not applied to the workspace. The changes take thus no effect. But they are present in the local repository and can be accessed offline. On the other hand `git pull` downloads the remote objects and tries to merge them with the workspace, applying the necessary changes.

The syntax of `git fetch` is:

```
git fetch [upstream-rep]
```

Again, as with `git push`, the optional `upstream-rep` parameter can be the name or the URL of a remote repository from where to download the changes. To fetch everything from a remote:

```
$ git fetch --all
```

The syntax of `git pull` is:

```
git pull [upstream-rep]
```

Same remarks as before for `upstream-rep`. The changes are by default incorporated only to the current branch.

For more advanced operation, consult `git help fetch` and `git help pull`.

Branching

Most people refer to branching as the killer feature of Git. It is so cheap and fast to use compared to other version control systems, that it has become common practise to branch very often in projects controlled with Git.

Normally, each Git repository has a default branch called master. To consult the branches in your local repository use:

```
$ git branch
* master
```

The current branch is marked with a `*` character. For creating a new branch that starts at the current commit of the current branch:

```
$ git branch develop
$ git branch
develop
* master
```

The current branch is still `master`. Switching to an other branch is called *check out* in Git:

```
$ git checkout develop
Switched to branch 'develop'
$ git branch
* develop
master
```

You can create and check out a branch in one single step:

```
$ git checkout -b release
Switched to a new branch 'release'
develop
master
* release
```

As said before, branches are just pointers to commits. When switching from one branch to an other, Git populates the workspace with the appropriate snapshot referenced by the commit pointed to by the branch. This makes it possible to work on several things on parallel and to separate concerns.

Branches can be configured to track other remote branches, that is, branches of other upstream repositories. If a branch `A` tracks a branch `B`, `git push` within the branch `A` pushes the changes of `A` into the branch `B`. On the other hand, `git`

`pull` within the branch `A` pulls (fetches and merges) the changes from branch `B` into branch `A`.

For instance, suppose a Git repository called `my-rep` that has been used throughout this tutorial:

```
$ git clone my-rep my-rep-clone
Cloning into 'my-rep-clone'...
done.
Checking connectivity... done
$ cd my-rep-clone
$ git branch -vv
* master b5d0dfe [origin/master] foo
```

By default the local branch `master` in `my-rep-clone` is configure to track the remote branch `master`. Also by default, the upstream repository from where the local repository has been cloned is called `origin`. Thus the tracked remote branch is `origin/master`.

To see which other branches can be tracked on the remote, the command `git branch -r` can be used:

```
$ git branch -r
origin/HEAD -> origin/master
origin/develop
origin/master
origin/release
```

The remote repository has four branches. The special branch `HEAD` is just a pointer to the currently checked out branch. To track a remote branch it is sufficient to check it out:

```
$ git checkout develop
Branch develop set up to track remote branch develop from origin.
Switched to a new branch 'develop'
$ git branch -vv
* develop b5d0dfe [origin/develop] foo
master  b5d0dfe [origin/master] foo
```

Now the local repository has a new branch called `develop` that is tracking the remote `origin/develop` branch.

In the configuration file of the local repository this information is also very easy to see:

```
$ cat .git/config
[core]
    repositoryformatversion = 0
    filemode = true
    bare = false
    logallrefupdates = true
[remote "origin"]
    url = /my-rep
    fetch = +refs/heads/*:refs/remotes/origin/*
[branch "master"]
    remote = origin
    merge = refs/heads/master
[branch "develop"]
    remote = origin
    merge = refs/heads/develop
```

There is one remote upstream repository configured called `origin` (located in `/my-rep`). Two branches are used in the local repository `master` and `develop` and both track the respective branches in the `origin`.

Local branches do not have to have the same name as the remote ones and there can be several branches tracking the same remote or an other local branch:

```
$ git checkout -b develop-fork
Switched to a new branch 'develop-fork'
$ git branch -vv
develop        b5d0dfe [origin/develop] foo
* develop-fork b5d0dfe foo
master         b5d0dfe [origin/master] foo
```

The new `develop-fork` branch does not track any other branch. To track an other branch:

```
$ git branch --set-upstream-to=origin/develop
Branch develop-fork set up to track remote branch develop from origin.
$ git branch -vv
develop        b5d0dfe [origin/develop] foo
* develop-fork b5d0dfe [origin/develop] foo
master         b5d0dfe [origin/master] foo
```

Now the `develop-fork` branch tracks the remote `origin/develop` branch in the remote repository, just as the local `develop` branch does.

This can be also unset:

```
$ git branch --unset-upstream
$ git branch -vv
develop      b5d0dfe [origin/develop] foo
* develop-fork b5d0dfe foo
master       b5d0dfe [origin/master] foo
```

What if you want to track a remote branch that does not exists yet in the remote? Suppose for instance that you want to work in a feature `f1`. You create a branch for that and do some work:

```
$ git checkout -b feature/f1
Switched to a new branch 'feature/f1'
$ touch f1
$ git add f1
$ git commit -m "Added f1"
[feature/f1 876770d] Added f1
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 f1
```

As always, most things are local, so are branches. Now you want to share this feature with others that will perhaps also contribute to it. You can not just push it:

```
$ git push
fatal: The current branch feature/f1 has no upstream branch.
To push the current branch and set the remote as upstream, use

    git push --set-upstream origin feature/f1
```

It does not work since the branch `feature/f1` does not track any remote branch. You must force the creation of a remote branch that is going to be tracked by your local branch. You can do this using the `--set-upstream` or `-u` switch:

```
$ git push --set-upstream origin feature/f1
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (2/2), 220 bytes | 0 bytes/s, done.
Total 2 (delta 0), reused 0 (delta 0)
To /my-rep
 * [new branch]      feature/f1 -> feature/f1
Branch feature/f1 set up to track remote branch feature/f1 from origin.
$ git branch -r
```

```
origin/HEAD -> origin/master
origin/develop
origin/feature/f1
origin/master
origin/release
```

A new branch is created on the remote, the changes of the local branch are pushed to the remote branch and the local branch is set to track the just created remote branch.

A branch can be deleted locally:

```
$ git checkout master
Switched to branch 'master'
$ git branch -d develop-fork
Deleted branch develop-fork (was b5d0dfe).
```

And can be deleted also remotely (be careful with that...):

```
$ git push origin :develop
To /my-rep
- [deleted]         develop
```

This removes the remote branch:

```
$ git branch -r
origin/HEAD -> origin/master
origin/feature/f1
origin/master
origin/release
```

But leaves the local one untouched:

```
$ git branch
develop
* feature/f1
master
```

Merging

Merging is the process of incorporating changes of one branch into an other. Suppose that starting in the `develop` branch, you want to add a feature by editing the file `foo`. Of course, you create a branch for that:

```
$ git checkout develop
Switched to branch 'develop'
$ git checkout -b feature/edit-foo
Switched to a new branch 'feature/edit-foo'
$ edit foo
$ git commit -am "Edited foo."
[feature/edit-foo 5f5a181] Edited foo.
1 file changed, 1 insertion(+)
```

The changes are only in the `feature/edit-foo` branch. Now you want to merge them into the development branch:

```
$ git checkout develop
Switched to branch 'develop'
$ git merge feature/edit-foo
Updating b5d0dfe..5f5a181
Fast-forward
foo | 1 +
1 file changed, 1 insertion(+)
```

Now the changes have been also applied to the development branch. The "Fast-forward" means that the commits of `feature/edit-foo` could be applied on the top of the last commit of the `develop` branch as they were, one after another, since the `develop` branch has not moved since the `feature/edit-foo` was branched. No new commit is needed.

If the two branches diverge, a new commit is needed to fusion the branches again. For example, the next sequence of commands creates a new branch called `feature/create-bar` out of the current state of `develop`:

```
$ git branch feature/create-bar
```

Still in the `develop` branch, a new file `README` is created and committed:

```
$ touch README
$ git add README
$ git commit -m "Added README."
[develop 2735f0b] Added README.
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 README
```

The `feature/create-bar` branch is now checked out and a new file `bar` is created and committed:

```
$ git checkout feature/create-bar
Switched to branch 'feature/create-bar'
$ touch bar
$ git add bar
$ git commit -m "Added bar."
[feature/create-bar f5d424c] Added bar.
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 bar
```

By now, both branches have diverged: both contain data that is not contained in the other. If you try now to merge the changes of `feature/create-bar` into `develop`:

```
$ git checkout develop
Switched to branch 'develop'
$ git merge feature/create-bar
Merge made by the 'recursive' strategy.
bar | 0
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 bar
```

Now the merge is done by the "recursive" strategy. This means that Git has to recursively search for the most common commit of both branches and apply the changes of both branches from there on. Additionally, a new commit (a "merge commit") must be added to join both branches. In fact the last command above opens an editor that lets you introduce the message for this commit.

Recursive merges consume more space in the repository since they have to add additional commits. But in turn, they can be more legible in the history of the repository. The branches appear more visible using the recursive strategy, since there is an additional node that joins two branches. This makes keeping track of developed features more easy. Because of that, there are people that suggest that the recursive strategy should be enforced. This can be done using the `--no-ff` switch:

```
$ git merge --no-ff feature/edit-foo
```

In this section it was assumed that Git could figure out how to do the merge. There are situations where this is not possible. The next section deals with this.

Conflict resolution

Suppose that you create a new branch `feature/edit-bar` out of `develop` to edit the file `bar`:

```
$ git checkout -b feature/edit-bar
Switched to a new branch 'feature/edit-bar'
$ echo "Hello" > bar
$ cat bar
Hello
$ git commit -am "Edited bar."
[feature/edit-bar e3da8ab] Edited bar.
2 files changed, 2 insertions(+)
```

Now you also edit the same file in the `develop` branch:

```
$ git checkout develop
Switched to branch 'develop'
$ echo "Bye" > bar
$ cat bar
Bye
$ git commit -am "Edited bar"
[develop bbe3383] Edited bar
1 file changed, 1 insertion(+)
```

And finally, you want to merge `feature/edit-bar` into `develop`:

```
$ git merge feature/edit-bar
Auto-merging bar
CONFLICT (content): Merge conflict in bar
Automatic merge failed; fix conflicts and then commit the result.
```

Git can not figure out how to merge the branches, since both have a conflict.

```
$ cat bar
<<<<<<< HEAD
Bye
=======
Hello
>>>>>>> feature/edit-bar
```

Standard conflict markers are added by Git to show the conflict. You can manually edit the file, include it in the next commit and commit to resolve the conflict:

```
$ edit bar
$ cat bar
Hello & Bye.
$ git add bar
```

```
$ git status
# On branch develop
# All conflicts fixed but you are still merging.
#   (use "git commit" to conclude merge)
#
# Changes to be committed:
#
#       modified:   bar
#
```

`git status` tells you that the conflicts are going to be resolved if you commit and this commit is still the *merge commit* that could not be done before.

```
$ git commit -m "Resolved conflict in bar."
```

A graphical tool can also be used to resolve conflicts. For that have a closer look at `git help mergetool`.

Ignoring files

Sometimes it's useful to ignore some files that won't make part of any commit. This is done in git by creating a file called `.gitignore`. The patterns in this file apply from the directory on where it is located downwards to any subdirectory.

Only untracked files are ignored this way. If you want to ignore uncommitted changes to a tracked file use:

```
git update-index --assume-unchanged [file]
```

If you want to stop tracking a file that is currently tracked, use:

```
git rm --cached <file>
```

The `.gitignore` file can itself be ignored in case you do not want to share it. But probably, in this case it is best to use instead the `.git/info/exclude` file, that has the same syntax and is always local, does not appear in `git status` and hence is never committed.

For more info: `git help gitignore`.

Undoing things

To discard the changes in a tracked file in the working directory:

```
$ git checkout -- <file>
```

Which actually checks out the file again from the local repository, effectively discarding any change, even if the file was deleted.

If you want to remove a file from the next commit, i.e. unstage the file:

```
$ git reset HEAD <file>
```

Git also permits to reset the working tree and the index to any previous commit:

```
$ git reset [option] <commit-sha1-hash>
```

There are different options depending on the `option` parameter. If you want to reset everything to the exact state of a particular commit:

```
$ git reset --hard <commit-sha1-hash>
```

See `git help reset` for more options.

To remove untracked files from the working tree:

```
$ git clean
```

This command has several useful options. See `git help clean` for details. A nice switch is `-n`, that actually only shows what would be deleted, but does not do it. It's advisable use it before cleaning.

Stash

As said before, the stash is an area where so save changes that you do not want to apply yet, but want to keep for the future. The stash is managed with `git stash`. An easy to follow tutorial on it is this one.

Consulting the history

For viewing the commit history, the command `git log` can be used. This command has many options, some of the most useful can be seen applied here.

Patching

Patches are very useful to share changes circumventing the repository. Suppose you have found an error in a file that you know how to correct and also know that somebody else needs this correction in order to be able to work properly. Probably, the best practice is to create a new feature branch, correct the error, push the branch upstream and inform your colleague to merge the change. This is not always possible though, for instance if you need certain permissions for some step of the process that you do not have.

In this case, you could create a patch ans send it via e-mail. This post shows how to do it in a comprehensive way.

Work flow with Git

In the last time, a very successful work flow with Git has emerged starting with this post. It is strongly advised that you read this post since it explains very well the philosophy behind it and how it is done with Git commands.

The ATLAS Pixel DAQ aims to use this work flow. Some key concepts, as a summary:

- There are only two main branches that exists for ever in the origin (central repository): `master` and `develop`.
  - `master` has always a production-ready state.
  - `develop` is the latest state of development that can go to a release.
- There are three types of supporting branches: `feature`, `release` and `hotfix`.
  - They receive their names based on how their are used. They are not "special".
- `feature` branches are used to develop new features.
  - They are typically local to a developer, but can be also shared among several in the central repository.
  - They start normally from `develop`.
  - They must be merged into `develop`.
- `release` branches are there to support the creation of the new release before merging into `master` since normally some files must be changed, such as CHANGELOG, version numbers, etc.
  - They start from `develop`.
  - They must be merged into `develop` and `master`.
- `hotfix` branches are there to fix some urgent bug in the production code, i.e. in the `master` branch.

- Normally they branch off from `master`.
- They must be merged back into `master`, `develop` and if a `release` branch exists, also into this one.

Some naming conventions are used in the ATLAS Pixel DAQ for the branches:

master : The master branch.

develop : The develop branch.

feature/name : A feature branch. The `name` is free to choose, but should be something related to the feature being developed.

release/name : The current release branch. The `name` might be the next version number.

hotfix/name : A hotfix branch. The `name` is free to choose, but should be something related to the fix being developed.

Everyone can check out the branches in `origin`. The branch `master` is writable only by members of the **GIT-librarians-atlaspixeldaq** mailing list though. Release and hotfix branches should only be used by certain persons in charge of performing the tasks the branches are supposed to be used for.

Feature branches can be created by anyone developing a new feature. They are probably most of the time local branches, but can be pushed to `origin` if a feature is too big for one single developer. In this case, each of the developers that contribute to this big feature should have local sub-branches to isolate their work from the others and merge their changes to the feature branch when they are ready to do so.

For example Alice and Bob will work together on a big feature `f1`. Some of the two developers creates a feature branch in `origin`:

```
alice $ git checkout -b feature/f1
alice $ git push --set-upstream origin feature/f1
```

Now Alice can work on a part of the feature in one local branch independently of Bob:

```
alice$ git checkout -b feature/f1-module-a
alice$ edit module-a.c
alice$ git add module-a.c
alice$ git commit -m "Created a module A."
alice$ git checkout feature/f1
alice$ git merge --no-ff feature/f1-module-a
alice$ git push
```

The work is done, the branches are not needed any more and could be deleted.

```
alice$ git branch -d feature/f1-module-a
alice$ git branch -d feature/f1
```

Bob can check out the common feature branch and then do the same as Alice:

```
bob$ git fetch --all
bob$ git checkout feature/f1
bob$ git checkout -b feature/f1-module-b
bob$ edit module-b.c
bob$ git add module-b.c
bob$ git commit -m "Created a module B."
bob$ git checkout feature/f1
bob$ git merge --no-ff feature/f1-module-b
bob$ git push
bob$ git branch -d feature/f1-module-b
bob$ git branch -d feature/f1
```

Note that the branches `feature/f1-module-a` and `feature/f1-module-b` are only local to Alice and Bob respectively. Working in this way should minimise possible conflicts.

Someone has then to merge the feature `f1` into the upstream `develop` branch. It would be good if a third person would review the changes before introducing them into the upstream `develop` branch:

```
carol$ git fetch --all
carol$ git checkout feature/f1
carol$ review
carol$ git checkout develop
carol$ git merge --no-ff feature/f1
carol$ git branch -d feature/f1
carol$ git push --prune
```

Bob and Carol start by fetching all remote references before doing any work (`git fetch --all`). Note that the last `push` is needed to push the changes upstream. The option `--prune` removes remote branches that have no local counterpart. This means that the `feature/f1` branch is going to be deleted also remotely, since it was deleted locally. Of course, if the feature is going to be developed further, there is no need to remove the branch at this point. If the people are allowed to merge into the upstream branch, Alice or Bob could do the upstream merge instead of Carol.

A scheme of this work flow can be seen on the following picture.

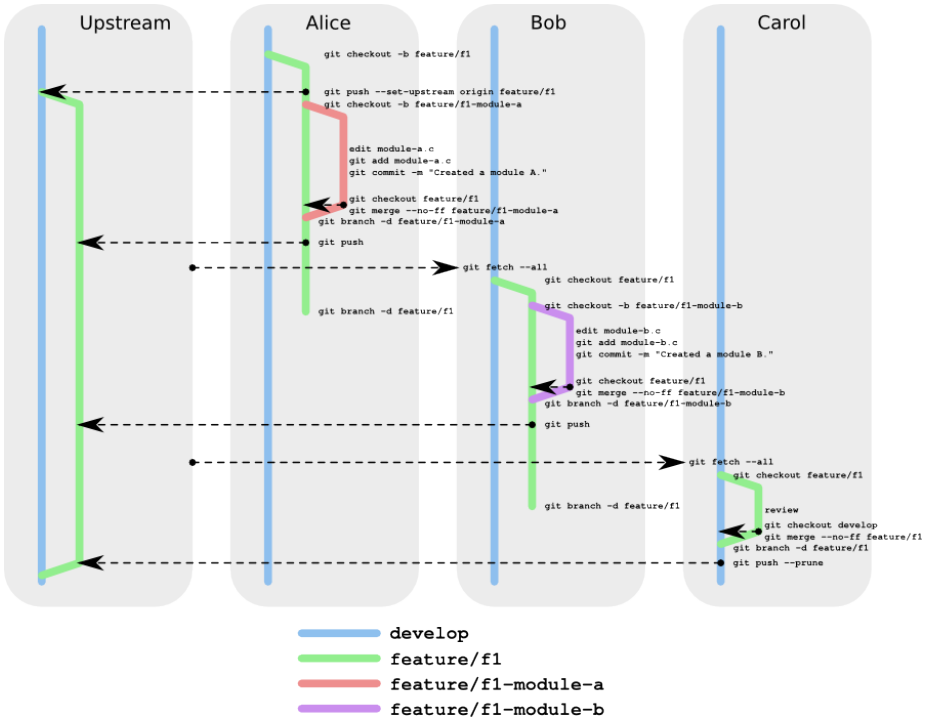

Figure 1: Scheme of the work flow between Alice, Bob and Carol.

Using git-flow

Although the work flow above can be followed using just Git commands, there is an arguably more comfortable way to do it using `git-flow`. `git-flow` are a set of git extensions to provide high-level repository operations for the discussed branching model that also ATLAS Pixel DAQ uses. A good cheatsheet can be found here. Be aware though that the designer of the cheatsheet considered `git flow feature pull` to be more important than `git flow feature track`, which it's probably what most people want.

For installing git-flow use:

```
$ cd atlaspixeldaq/scripts/ext
$ ./install-gitflow.sh
```

Most likely, as a developer, you will only need these commands to use the branching model:

`git flow init` : Initialises git-flow for your local repository. The defaults are fine, so just hit `ENTER` on each question. Do this before anything else. You can also use `git flow init -d` to accept all defaults.

`git flow feature start NAME [BASE]` : Starts a feature. Creates a new *local* branch called `feature/NAME` based on `develop` and switches to this branch. If the optional argument `BASE` is given, the feature `feature/NAME` will be created based on the branch `BASE` instead of `develop`. `BASE` is thus a complete branch name, not just a feature name.

`git flow feature publish NAME` : Publishes a feature in the remote. The branch `feature/NAME` must exists. It creates a branch called `feature/NAME` in the `origin`, pushes the local changes to it and the local branch `feature/NAME` now tracks the remote branch.

`git flow feature checkout NAME` : Checks out a feature called `NAME`. It's probably faster to just use `git checkout feature/NAME` instead.

`git flow feature track NAME` : Gets a published feature from the remote. Creates a local branch called `feature/NAME`, checks it out, pulls the remote `feature/NAME` branch and configures the local branch to track this remote one.

`git flow feature pull ORIGIN-NAME NAME` : Gets the published feature called NAME from the upstream repository called ORIGIN-NAME (for ATLAS Pixel DAQ normally *origin*). A new local branch `feature/NAME` is created and checked out. But note that this branch does *not* track any remote branch as opposed as using `git flow feature track NAME`.

`git flow feature finish NAME` : Finishes a feature. Switches to the **local** develop branch, merges the contents of `feature/NAME` into develop and removes the **local** branch `feature/NAME`. Note that nothing is pushed to the origin, you

have to push the local develop branch manually. No branch is deleted on the origin either. Thus this command (as others not shown here) is most likely to be performed by the person in charge of integrating the features into the develop branch in the origin.

The work flow between Alice and Bob shown before would be as follows with `git-flow` (assume both have issued `git flow init` on their respective local repositories.

Alice (or Bob) starts a new feature and publishes it to the central repository:

```
alice$ git flow feature start f1
alice$ git flow feature publish f1
```

Then Alice starts a new branch locally based on the feature `f1`, only for her, and works on it:

```
alice$ git flow feature start f1-module-a feature/f1
alice$ edit module-a.c
alice$ git add module-a.c
alice$ git commit -m "Created a module A."
alice$ git flow feature checkout f1 (or shorter: git checkout feature/f1)
alice$ git merge --no-ff feature/f1-module-a
alice$ git push
alice$ git branch -d feature/f1-module-a
alice$ git checkout develop
alice$ git branch -d feature/f1
```

Note that Alice deletes her local branches using Git standard commands and not using `git flow feature finish`, since this last command would try to merge into the `develop` branch.

Bob starts to track the common feature `f1` and works in his own local branch:

```
bob$ git fetch --all
bob$ git flow feature track f1
bob$ git flow feature start f1-module-b
bob$ edit module-b.c
bob$ git add module-b.c
bob$ git commit -m "Created a module B."
bob$ git flow feature checkout f1 (or shorter: git checkout feature/f1)
bob$ git merge --no-ff feature/f1-module-b
bob$ git push
bob$ git git branch -d feature/f1-module-b
bob$ git checkout develop
bob$ git branch -d feature/f1
```

The same comment on deleting the local branches applies here as for Alice before. Then Carol (or Alice or Bob if they are allowed) reviews and pushes the changes the upstream `develop` branch, possibly cleaning up things (`--prune`).

```
carol$ git fetch --all
carol$ git flow feature pull origin f1
carol$ review
carol$ git flow feature finish f1
carol$ git push --prune
```

Carol in this case uses just `git flow feature pull` since she is not interested in tracking the branch, only reviewing it. It she wanted to make some changes to the remote `f1` branch, she would use `git flow feature track f1" instead ofgit` flow feature pull origin f1`. She could also keep usinggit` flow feature pull`and use`git branch –set-upstream-to`orgit push –set-upstream“. The following three examples are equivalent for pushing changes to the upstream `feature/f1` branch.

Using `git flow feature track`:

```
carol$ git fetch --all
carol$ git flow feature track f1
carol$ review
carol$ git push
carol$ git flow feature finish f1
carol$ git push --prune
```

Using `git flow feature pull` and `git branch --set-upstream-to`:

```
carol$ git fetch --all
carol$ git flow feature pull origin f1
carol$ review
carol$ git branch --set-upstream-to=origin/feature/f1
carol$ git push
carol$ git flow feature finish f1
carol$ git push --prune
```

Using `git flow feature pull` and `git push --set-upstream`:

```
carol$ git fetch --all
carol$ git flow feature pull origin f1
carol$ review
carol$ git push --set-upstream origin origin/feature/f1
carol$ git flow feature finish f1
carol$ git push --prune
```

External tools

GUIs

The command line is fine, but some GUIs for working with Git are also helpful:

- Sourcetree (Only for Mac OS)
- Smartgit (Windows, Mac OS, Linux)

Git-new-directory

When you want to switch to another branch in Git, you simply say `git checkout branchname`. But there's a small problem with this approach: If you want to switch to another branch, but still have a lot of unsaved changes, then the switch might fail. Or you might want to have two separate branches checked out at the same time.

One solution to this is to simply create another local clone of your repository. Git automatically uses hard links when you clone locally, so cloning is very fast. But there is one problem with this: You now have another, separate repository you need to keep up to date.

This is where `git-new-workdir` comes in. Instead of doing a full-blown clone of your repository, it simply sets up a new working directory (with its own index) for you. The actual repository itself is shared between the original and the new working directory. This means: If you update one repository, the new commits are instantly visible in all other working directories as well. Create a new commit or branch in one of your working directories, they're instantly available in all working directories.

Note: Even though the commits are automatically there, Git won't update the working copy if you've got the same branch checked out. You'll have to do that for yourself.

To install, simply stick this script somewhere in your `$PATH` and make it executable. Run it without any arguments to see its usage, and run

```
git-new-workdir path/to/repository/ path/to/new/workdir
```

to get a new working directory. You can optionally specify a branch which will then be automatically checked out after setting everything up.

(This text about `git-new-directory` has been derived with minor changes from here.)