Git

Using Git

Getting Started

For short and to-the-point Git tutorials see e.g. here:

- https://www.atlassian.com/git/tutorials/

Git identifies the authors of commits by their email, so when you first use git on a new system, you have to tell it yours:

```
$ git config --global user.name "John Doe"
$ git config --global user.email "john@doe.com"
```

The `--global` option puts this into your global `~/.gitconfig` file, which applies to all your local git repos. Using `--local` (the default) allows a per-repository setting.

The email you use should be the one associated with your DESY user account. That way, your commits are associated with your user profile.

To clone the main `scetlib` repository, do

```
$ git clone ssh://git@stash.desy.de:7999/scetlib/scetlib.git
```

- This creates a new local repository in the subdirectory `scetlib/` of the current directory.
- The repository data itself is in the `scetlib/.git/` directory. (The repo's main configuration lives in `scetlib/.git/config`, which can be useful occasionally.
- The `scetlib/` directory itself serves as the working area. (It starts out containing the content of the default branch, typically either `develop` or `master`.)

The possible access methods and different repositories are described below.

Workflow

We use a standard Git workflow, where all feature development happens in branches. Each module has its own branch with possible sub-branches for feature development. The integration happens through the develop branch.

Here is a summary of the conventions for branches and their relations

Branch	Purpose	Originates from	Merges into
`master`	Stable releases		`stable`
`stable`	Public access for stable releases, contains abridged (squashed) version of master		
`hotfix-xxx`	Critical fixes to stable releases	`master`	`develop` and `master`
`develop`	Integration of all features that will go into next release	`master`	
`release-x.y`	Preparation for stable release	`develop`	`develop` and `master`
`core`	Development of core library	`develop`	`develop`
`module-xxx`	Main development for module 'xxx'	`develop`	`develop`
`module-xxx-feature`	Development of 'feature' in module 'xxx'	`module-xxx`	`module-xxx`

Basic Commands

Below, we go through the most important git commands and their most relevant options.

All commands have additional options for special purposes or finetuning their behavior. There are also many additional commands. To see a detailed documentation for a command with a list of all options and usage examples, do

```
$ git help <command>
```

Page Contents

Subpages

Create new subpage

Important Note

Git commands always act on the entire repository you are in, it does not matter in which directory you are inside your working area (this is a big difference to svn)

- The current directory only matters to the extent that file paths are always given relative to the current directory.
- Git does not track directories and files by name, it only tracks files by their content. This means
  - Empty directories are simply ignored
  - You can freely copy, delete, rename, move files and directories in your working area without any danger of destroying the repo's integrity. In particular, there is no need to tell git about this, it automatically detects it based on content.

Getting information

List of all branches (with the current branch highlighted):

```
$ git branch [-vv | -a | -avv]
```

- The different options give different level of detail. The `-avv` option is the most useful. With `-a` it shows both local and remote branches. With `-vv`, for each local branch, it shows which remote branch it trackes and by how many commits it is ahead/behind.

Show the commit history on the command line:

```
$ git log [--oneline] [--graph]
```

- By default, the complete commit information (author, date, message) is shown. With `--oneline` only the first line of each commit message is shown.
- With `--graph` a text-based graphical representation of the commit tree is included.
- There are many options for formatting etc. However, there are many external programs for graphically visualizing the commit history tree, which are more convenient to use than `git log` from the command line. A standard program coming with git is `gitk`, which also shows the detailed changes.

Working with branches

Branches

in git are simply pointers to commits. Commits do not belong to (are "owned by") specific branches.

- Commits have pointers to their parent(s) and child(s) and are only associated to a branch by being part of the history of the commit a branch points to, i.e., they are reachable from the branch by going backward in history.
- Branches are important because commits that are not reachable from any branch (or the current HEAD) are eventually thrown away during garbage collection.

Switch to a branch to start working on it, in this example the `develop` branch:

```
$ git checkout develop
```

- This updates the working tree to reflect the content of `develop` (more precisely, the commit `develop` points at).
- This also updates the HEAD to point at `develop`, which makes it the "current branch" which many operations (commit, merge, etc.) are relative to.
- As a special case: If the branch does not yet exist but a remote branch of the same name exists (e.g. `origin/develop`), this will automatically create a local branch `develop` which tracks `origin/develop`.

Create a new branch, say `feature-cool`, and start working on it:

```
$ git checkout -b feature-cool [<start-point>]
```

- This first creates the new branch pointing at the given `<start-point>` or if not given the current HEAD. It then switches to the new branch.

- If `<start-point>` is a remote branch, this sets up `feature-cool` to track the remote branch.
- The above is exactly equivalent to doing

```
$ git branch feature-cool [<start-point>]
$ git checkout feature-cool
```

Rename or delete a branch:

```
$ git branch -m feature-cool feature-supercool
$ git branch [-d|-D] feature-supercool
```

- The `-d` option only deletes the branch if it is merged with an upstream branch, i.e. if the commits are reachable from another branch.
- The `-D` options deletes the branch no matter what. This drops commits that are only on this branch and not reachable otherwise. (The commits are still present for a while and are only dropped during the next garbage collection. This means you can checkout and recover them if you know their commit id.)

Making changes

Show the **current status** of the working tree and the commit staging area (also called the 'index'):

```
$ git status [--ignored]
```

- The `--ignored` option also shows all files that are being ignored by git.
- In general `git status` is your best git friend, since it also tells what and how to do next.

Register changes to be part of the next commit

```
$ git add <files>
$ git add <path>
```

- This adds the given modified or new `<files>` or all modified/new files in the given `<path>` to the commit staging area (also called the index). All modifications (not just new files) have to be staged before committing. Further modifications to already staged files have to be added again.
- The staging area is useful since it selectively allows to choose which changes should be part of the next commit.

Unstage changes from the next commit

```
$ git reset <files>
$ git reset <path>
```

- Removes the given `<files>` or entire `<path>` from the staging area, so this is the oppsoite of `git add` above.
- Note that `git reset` is rather powerful, so one has to be a bit careful. (When given a branch or commit it will operate on the entire commit.)

Commit staged changes:

```
$ git commit [--amend]
```

- This turns the content of the staging area into a new commit object and updates the current branch to point to it.
- Normally, this opens an editor to compose a commit message. *Please write useful commit messages.*
- With `--amend`, the staged changes are combined with the previous commit into a new commit which **replaces** the previous commit. This is very useful to fixup the last commit when some changes were forgotten or to fix a bug. *However, see the warning on changing history below.*

Convenient graphical tools for staging and unstaging, reviewing changes, and committing are

```
$ git gui &
$ git cola &
```

Good commit messages are crucial for navigating the commit history, finding bugs, etc.

Commit messages should have this form

Write one line/sentence (in active present tense) with a short description of what the commit does.

<empty line>

The main purpose is to document the commit and the changes, not so much to discuss your reasons for doing so.

An often useful form is to give a bullet list with details:

- Include relevant things for someone to be able quickly evaluate what happened.
- Mention new features, changed behavior, possible pitfalls etc. that users should be aware of.
- Be *concise.*

Merging changes

Git has two different ways to combine the changes in different branches: merging and rebasing.

Merge changes from `<branch>` since its history diverged from the current branch into the current branch (see `git help merge` for details and examples)

```
$ git merge <branch>
```

- This creates a so-called merge commit which has both branches as childdren and thus combines the histories of both branches.
- In case of a conflict, the merge stops before committing. The conflict can then be manually resolved and the final result staged and committed as usual. Alternatively, the merge attempt can be abandoned with `git merge --abort`, which returns the current branch to its pre-merge state.
- If `<branch>` is directly ahead of the current branch, so there is no fork in the history, the merge resolves as a so-called **fast-forward merge**. In this case, the current branch is simply updated ("fast-forwarded") to point at `<branch>` without generating a separate merge commit.
- While possible, it is *strongly discouraged* to run this if you have uncommitted local changes (since in that case aborting the merge can fail to recover the pre-merge state and loose your local changes).

Reapply changes of each commit of the current branch since its history diverged from `<branch>` on top of `<branch>` (see `git help rebase` for details and examples)

```
$ git rebase [-i] <branch or commit>
```

- This ports/transplants each commit after the common ancestor one-by-one onto `<branch>`, appropriately merging each one with the existing changes in `<branch>`.
- This creates new commit objects and discards the old ones (unless they belong to the history of a another branch), and therefore changes the history of the current branch.
- In case of merge conflicts, the rebase stops. After manual conflict resolution, it can be continued with `git rebase --continue` or abandonded with `git rebase --abort`.
- As with merge, if `<branch or commit>` is directly ahead of the current branch, it will simply be fast-fowarded.
- If the piece of history being rebased itself contains a nonlinear merge history, that history will be flattened out and lost. It is then better to use merge if the history should be preserved. This means, that in general only commits after the last merge commit are easy to rebase.
- The `-i` option triggers an interactive rebase. This allows the commits to be selected, combined, reordered, edited, etc. This allows to rewrite and reorder the history of the current branch by specifying a `<commit>` to rebase against which is in the direct past of the current branch, such as `HEAD~3`.

Rebasing vs. Merging

Rebasing keeps the commit tree linear, which makes future merges easier (e.g. more likely to resolve as fast-forward). *However, see the warning on changing history below.* If possible/allowed, always use rebase

- for merging local changes into a remote branch
- for merging changes in develop into feature branches

Merging preserves the exact commit and merge history, including the historical existence of branches and connection between different branches. Only use merge

- If there is a reason to record that a merge happened (a typical example is merging a feature or module branch into develop)
- If rebase is not allowed.
- If rebase would flatten out existing nontrivial merge history.

Throw away changes:

```
$ git reset --hard <branch or commit>
```

- This resets the current branch **and** working tree to the state of the given branch or commit, throwing away all commits and changes since the common ancestor.
- Basically this is useful for throwing away a failed experiment. But be careful, since this also discards any local changes to tracked files in the working area.
- The parent of the current commit is conveniently specified by `HEAD~`, and similarly `HEAD~~` or `HEAD~2` is the current commit's grandparent. This also works for branches and commit ids, so `develop~3` refers to the commit 3 steps back from the commit the `develop` branch points at.

Interacting with remote repos

> **Remote branches**
>
> are also just pointers to **locally known** commits. Their difference to local branches is in their different behaviour.
>
> - They can be checked out, which has the same effect as directly checking out the commit they point to.
> - They are kept synchronized with where the corresponding branch in the remote repo points to.

In git, updating your repo with new changes from a remote repo (the equivalent of `svn update`) is a two-step process: First you download new commits, then, if needed, you combine those with your current work using rebase or merge, just like combining changes between local branches. Separating the two steps provides much more control over nontrivial merges or conflicts, which makes handling them much easier.

Download upstream commits from the remote repo:

```
$ git fetch [-p]
```

- This downloads new commits and also updates the corresponding remote branches.
- To investigate the upstream changes, you can do `git checkout origin/<branch>`. (This does not create a local `<branch>` that tracks `origin/<branch>`, but simply checks out the commit.)
- The -p option removes any remote branches that do no longer exist on the remote repo.

Incorporate upstream commits with your local ones:

```
$ git [i-] rebase [origin/<currentbranch>]
```

- This uses rebase as described above to reapply any new local commits in the current branch on top of the remote branch.
- If the current branch is called `<currentbranch>` and is setup to track the remote branch `<origin/<currentbranch>` (which is the standard case), then the remote branch is automatically supplied and can be omitted.
- By default you should use rebase to combine the remote changes with your local (unpublished) commits.

Merge upstream changes with your local ones:

```
$ git merge [origin/<currentbranch>]
```

- This uses mege as described above to merge the remote branch with your local one. As with rebase, if the current branch tracks the remote branch, the remote branch is automatically supplied and can be omitted.
- By default, you should use rebase instead of merge here, since it avoids cluttering the history with typically meaningless merge commits (unless part of your local history being merged is for some reason already public.

Single Step Pull:

```
$ git pull [--rebase]
```

- This precisely performs the above fetch and merge, or with the `--rebase` option fetch and rebase. (So it behaves similar to `svn update`.)
- This sounds convenient, but it is **discouraged**, because you cannot see the fetched changes and loose control over what gets merged.

Upload your changes to the remote repo:

```
$ git push [--all]
$ git push -u origin <newbranch>
```

- This uploads new local commits for the current branch (or with `--all` for all local branches) to the remote repo together with updating the remote branches.
- The push is only accepted if no nontrivial merge is required on the remote end, i.e. all remote branches can be fast-forwarded. Otherwise, it means the remote history has diverged from yours, in which case you first have to fetch and incorporate the two histories and then push.
- The second form is used to push a newly created local branch, where the `-u` is to setup the local branch to henceforth track the newly created remote branch.

And finally, here is the promised

> **Warning on Changing Public History**
>
> Commits that have been pushed become public history and once someone has pulled them they become shared history. Shared history should normally not be changed, which means:
>
> - Do not amend commits that have already been pushed (unless you can be sure nobody has pulled them yet).
> - Do not rebase branches that have been pushed (unless you can be sure nobody has pulled them yet).

The reason is that anyone having pulled the history, by pulling again their remote branch will now point to the changed history but their corresponding local branch will still point to the previous history. This can be confusing and requires them to reset the local branch to the new remote one. If someone already made changes, they will have to rebase their changes onto the new remote branch.

Stash

Repositories

The repositories are hosted on DESY Stash.

Repository	Purpose	Access
`scetlib`	main development (master, develop, core, module branches)	*clone/pull*: all developers *browse/comment*: all developers *push*: module developers only
`scetlib-data`	stores module data	
`public`	public releases (stable branch)	*clone/pull/browse*: public
private modules	private modules/extensions	*read/write*: module developers only

Access Methods

There are different methods for accessing the Stash repositories summarized here:

Method	How to access	Operations
ssh	`ssh://git@stash.desy.de:7999/scetlib/scetlib.git`	clone, pull, push
https	`https://username@stash.desy.de/scm/scetlib/scetlib.git`	clone, pull, push
web interface	`https://stash.desy.de/projects/SCETLIB/repos/scetlib/browse`	browse, comment, pull request

There are three ways to authenticate with Stash:

- **authenticated access** requires a DESY Stash account and allows all three methods above
- **unauthenticated access** is possible for ssh access only

- **anonymous access** is possible via https and web for public repositories only

Here is a summary of allowed operations for the different levels of authentication:

Operation	authenticated	unauthenticated	anonymous
clone, pull, browse public	✅		✅
clone, pull	✅ (ssh/https)	✅ (ssh)	❌
browse, comment	✅ (web)	❌	❌
push to unrestricted branch	✅ / ❌	❌ (✅ via ssh)	❌
push to write-restricted branch	✅ / ❌	❌	❌

Here ✅ / ❌ means that access can be configured for specific users if needed.

Configuring ssh access

ssh is the recommended way for command-line access, since it uses ssh keys and no passwords are required.

To configure **authenticated ssh access**, go to your Stash profile and install your public ssh key. When accessing the git repos via ssh using a so-installed ssh key, the Stash server automatically authenticates and associate the access with your Stash account.

For **unauthenticated ssh access**, your public ssh key can be installed directly in the repo (you have to email it to Frank). The advantage is that this works without requiring a Stash account, but it has some limitations as shown in the table above.

> ⚠ **Note**
>
> *Some networks block port 7999. In this case https is the only fall-back option.*