1. Introduction

Definition 0.1 Revision: Represents a version of the source code. Git implements revisions as **commit objects** (or short commits). These are identified by an SHA-1 hash.

Definition 0.2 Commit: Commit holds the current state of the repository. A commit is also named by SHA-1 hash. When you commit your changes into a repository this creates a new *commit object* in the Git repository. This commit object uniquely identifies a new revision of the content of the repository. Every commit object has a pointer to the parent commit object. From a given commit, you can traverse back by looking at the parent pointer to view the history of the commit. If a commit has multiple parent commits, then that particular commit has been created by merging two branches.

Definition 0.3 Branch: A branch is a named pointer to a commit. Selecting a branch in Git terminology is called to **checkout** a branch. If you are working in a certain branch, the creation of a new commit advances this pointer to the newly created commit.

Definition 0.4 Master: Is the default main branch that automatically gets created.

Definition 0.5 Staging Area/Index: The staging area is the place to store changes in the working tree before the commit. The staging area contains a snapshot of the changes in the working tree (changed, new or deleted files) relevant to create the next commit.

Definition 0.6 HEAD: Is a pointer, which (usually) always points to the latest commit *in a branch*. Whenever you make a commit, HEAD is updated with the latest commit. Think of the HEAD as the "*current checked-out branch*".
When you switch branches with `git checkout`, the HEAD revision changes to point to the tip/latest commit of the new branch.

How to see where HEAD points to?

```
cat .git/HEAD
```

Definition 0.7 Detached HEAD mode: Normally, when checking out a proper branch name, Git automatically moves the HEAD pointer along when you create a new commit and sets it to the lastest commit.
But: if we checkout a specific commit by a *SHA1 hash*, Git will not do this for us and thus if we want to commit new changes the no longer belong to any branch ⇒ detacht HEAD. In order to fix this we simply need to checkout a branch: e.g.

```
git checkout master
```

Notes

- There can exist multiple heads but the alias HEAD is the currently selected head.
  Hence: HEAD determines which branch we are currently on.

2. Git Config

Allows to get and set repository or global options.

cpp dfsdf

Creation

0.1. Creating a gitlab project from the commandline

Creating remote Project

```
git push --set-upstream remote:namespace/project_name.git
↪  branch
```

- remote: e.g. git@gitlab.ethz.ch
- namespace: e.g. groups and subgroups
- branch: e.g. master

Note

Afterward simply add

```
git remote add origin remote/namespace/project_name.git
```

1. History

Log

Get a list of commit history:

```
magit-log [--all]
```

1.1. Undo local changes

Of local unstaged file

```
git checkout [--] filename
```

When checking out previous commit

```
git checkout -f master
```

1.2. Undo Commited Changeds

Undo without leaving a trace of the commit

1. Reset current HEAD to the specified state:
   ```
   git reset previous label or sha1
   ```
2. Do local work and commit it
3. Push and force remote to consider this push and remove the previous one (specifying remote-name and branch-name is not mandatory but is recommended to avoid updating all branches with update flag).
   ```
   git push -f remote-name branch-name
   ```

Note

Don't do this if someone allready pulled your change (I would use this only on my personal repo).

Ignoring Files

1. Adding Ignored Files

```
git -f files
```

Local

1. Editing

1.1. Updating locals remote repository location

```
git remote set-url origin new_url
```

Unstaging a File

Unstage single file

```
git checkout file
```

Unstaging all Files

Unstage all files

```
git checkout --
```

Reset local unstaged changes to the last commit

```
git reset --hard HEAD
```

or equivalently

```
git checkout [-f/--force] HEAD
```

Note

While git checkout does only delete local changes, git reset will discard also commits if we go back in the history.

Overwrite local unstaged files with remote

1. Download content from remote without merge or rebase:
   ```
   git fetch --all
   ```
2. Reset the branch branch to what we just fetched from the remote:
   ```
   git reset --hard origin/branch
   ```

Attention hard

Will overwrite all local unstated and uncommitted changes.

Remote

List all remote's

```
git remote options
```

Options:

- --verbose/-v: show fetch&pull url's

Branches

List branches

```
git branch branch
```

Options:

- --all/-a: list both remote-tracking and local branches

Good Branch Naming Conventions

- `feature/feature-name`: [feature/user-authentication]
  Used for developing new features.
- `bugfix/bug-description`: [bugfix/fix-login-error]
  Used for fixing bugs.
- `hotfix/hotfix-description`: [hotfix/critical-security-patch]
  Used for urgent fixes in production.
- `release/release-version`: [release/1.0.0]
  Used to prepare for a new production release.
- `experiment/experiment-name`: [experiment/new-ui]
  Used for experimental features or spikes.
- `chore/task-name`: [chore/update-dependencies]
  Used for routine tasks like updates or cleanups.
- `docs/description`: [docs/add-api-documentation]
  Used for updating or creating documentation.

Switching branches

```
git checkout branch
```

Options:

- -b: create a new branch if it does not exist

Tracking remote branches

```
git checkout -b branch --track remote/branch options
```

this is equivalent to to

```
git checkout branch
```

if branch does not exist locally but there exists exactly one remote (e.g. origin) with a matching name.

Options:

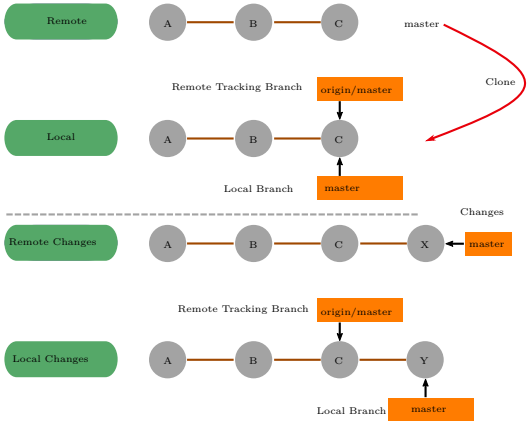
- --track/-t: specifies which remote branch to track if there exist multiple remotes with the desired branch. (implies -b for convenience).

1. Remote Tracking Branches

Definition 7.1 remote/branch
Remote Tracking Branch:
Is a local branch, that is connected to a *branch* on *remote repository* and shows the local state of a remote branch:

```
git checkout --track [local_branch] remote/branch
```

Explanation 7.1.
A remote tracking branch shows us the local state (i.e. after the last fetch) of a given remote branch:



1.1. origin/master

Definition 7.2 origin/master**:** Is the remote tracking branch[def. 7.1] of the branch `master` on the remote `origin`.

Reset

Revert

Fetching

Definition 7.3 Fetching:
Downloads commits, objects and refs from a remote repository and updates `origin/master`[def. 7.2] and `FETCH_HEAD`[def. 7.4]:

```
git fetch [options] [remote] [refspec]
```

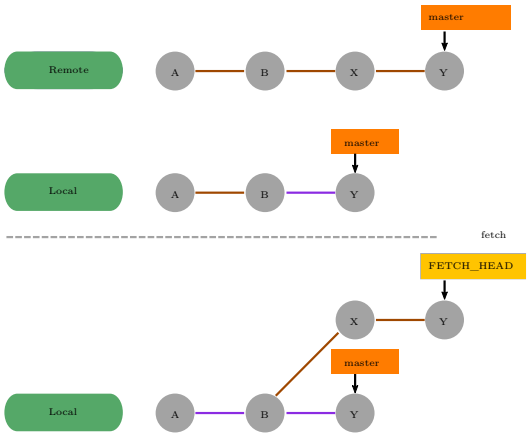


Figure 1: git fetch

Definition 7.4 FETCH_HEAD:
FETCH_HEAD is a shortlived reference to the HEAD of the last fetch and its value is stored inside `.git/FETCH_HEAD`.

Options

--all: Fetches all branches from all remotes.

Merging

1. Merging Remote Changes

Merging branches

Definition 8.1 Merging Branches:
Is the process of merging another branch into our current branch by creating a new commit:

1. checkout a branch into which we want to merge the changes from another branch:
```
git checkout branch
```
2. Merge another branch into branch
```
git merge another_branch
```

2. Merge Conflicts

Conflict Markers

Definition 8.2 Conflict Markers:
When an automatic merge fails we need to resolve the conflicts by hand.
Those conflicts that cannot be resolved by git are then indicated by so called conflict markers.

Definition 8.1 HEAD:
Indicates what we already have locally, as `HEAD` points to the current branch [def. 0.6].
```
<<<<<<< HEAD:file.txt
something
=======
```

Definition 8.2 Pulled Changes: Indicate what the pulled commit on the remote branch (with the given commit number) would introduce in our local repository
```
=======
something_else
>>>>>> 77976da35a11db4580b80ae27e8d65caf5208086:file.txt
```

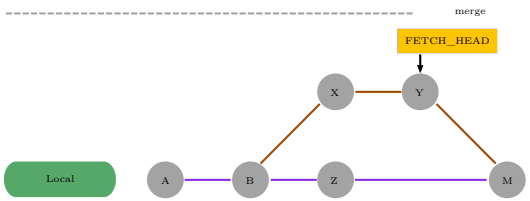
Pulling

Definition 9.1 git pull:
Performs a git fetch followed by a merge and creates a new commit:

1. A `git fetch`[def. 7.3].
2. A `git merge FETCH_HEAD`.

```
git pull [options] [local_branch] remote/branch
```

git fetch fig. 1



Options

`--rebase`: Is similar to:
```
git fetch
git rebase origin/master
```
thus we first apply our remote changes and rebase our local changes onto it.

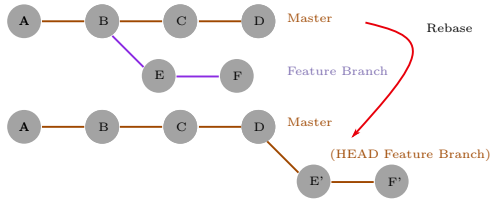
`--fast-forward`: Is similar to:
```
git fetch
git merge --ff-only origin/master
```
thus we git aborts if our a local and remote branches have diverged.

Rebasing

Rebasing a branches

Definition 10.1 Git Rebase:
Allows us to rebase a feature branch onto another branch by rewriting its history:

1. Checkout the feature branch we want to rebase:
```
git checkout feature_branch
```
2. Rebase the feature branch onto another branch:
```
git rebase [options] new_parent [feature_branch]
```
3. In case of merge conflicts resolve them and continuoe:
```
git rebase --continue
```



Note
We either change to the feature branch `feature_branch` that we want to rebase or we specify it explicitly.

Attention:
- Git rebase rewrites the commit history, thus we have to be careful with remote changes.
- Rebase cannot be undone unless we start a new backup branch before starting.

Cherry Picking

Submodules

1. Adding a new submodule

Run from main project repository:
```
git submodule add <path-to-external> [name]
```

2. Pushing new changes to the Submodule

Add, commit, pull, push normally from within the submodule.
```
1: cd [path-to-submodule]
2: git add [files]
3: git commit -m <message>
4: git pull [remote e.g origin] [branch e.g master]
5: git push [remote e.g origin] [branch e.g master]
```

3. Telling the project that the submodule changed

After making changes to the submodule normally add, commit, push submodule form main-project.
```
1: cd ..
2: git add <submodule>
3: git commit -m ``Updated reference to <name> submodule''
4: git pull [remote e.g origin] [branch e.g master]
5: git push [remote e.g origin] [branch e.g master]
```

4. Updating And Existing Submodule

To Update a project to the newest commit of the submodule run: `git submodule update [--init]` sometimes it is necessary to run
`git submodule update [--init --force --remote]`

Note
All git commands executed from within a submodule are with respect to that submodule repo and are not seen/affacted by the main repository.

5. Changing a Submodules Path

1. edit path inside `.gitmodules`
2. Re-sync module `git submodule sync`
3. Update submodule
   `git submodule update --init --recursive --remote`

Magit

Maggit Status C-x g

- **Magit Status**:

Useful Commands

- `magit-remote-add`: add new remote to git repository.
- `magit-log`: show commit log.

1. Show Remote Branches

`magit-show-refs-popup` or y during maggit status

2. Pull from Upstream

F: to display pull popup (`magit-pull-popup`)

3. Push to Upstream

1. **Staging**:
   s: to add the file under the cursor to the stage (`magit-stage`)
   S: to add all tracked files to the stage (`magit-stage-modified`)
   u: to unstage the file under the cursor to the stage (`magit-unstage`)
   U: to unstage all staged files (`magit-unstage-all`)
2. **Committing**:
   c c: write a commit message
   C-c C-c: Finish/close message and commit (`magit-commit-popup`)
3. **Pushing to upstream**:
   P u: push upstream (`magit-push-popup`)
4. **Pulling from upstream**:
   F: to pull (`magit-pull-popup`)

4. Branches b

- b b – `magit-checkout`: checkout a branch
- b n – `magit-branch`: create a new branch
- b c – `magit-branch-and-checkout`: create a new branch and check it out
- b s – `magit-branch-spinoff`:
- b x – `magit-branch-reset`:
- b k – `magit-branch-delete`: deletes one or multiple branches
- b r – `magit-branch-rename`: renames a branch

4.1. Submodules

`magit-submodule` or o during maggit status

- a: add a submodule to the repository (`magit-submodule-add`)