GitLab

GIT CHEAT SHEET

1. GIT CONFIGURATION

```
$ git config --global user.name "Your Name"
```

Set the name that will be attached to your commits and tags.

```
$ git config --global user.email "you@example.com"
```

Set the e-mail address that will be attached to your commits and tags.

```
$ git config --global color.ui auto
```

Enable some colorization of Git output.

2. STARTING A PROJECT

```
$ git init [project name]
```

Create new local repository. If **[project name]** is provided, Git will create a new directory named **[project name]** and will initialize a repository inside it. If **[project name]** is not provided, then a new repository is initialized in current directory.

```
$ git clone [project url]
```

Downloads a project with entire history from the remote repository.

B. IGNORING FILES

```
$ cat .gitignore
/logs/*
!logs/.gitkeep
/tmp
*.swp
```

Thanks to this file Git will ignore all files in **logs** directory (excluding the **.gitkeep** file), whole **tmp** directory and all files ***.swp**. Described file ignoring will work for the directory (and children directories) where **.gitignore** file is placed.

3. DAY-TO-DAY WORK

```
$ git status
```

See the status of your work. New, staged, modified files. Current branch.

```
$ git diff [file]
```

Show changes between **working directory** and **staging area**.

```
$ git diff --staged [file]
```

Show changes between **staging area** and **index** (repository commited status).

```
$ git checkout -- [file]
```

Discard changes in **working directory**. This operation is **unrecoverable**.

```
$ git add [file]
```

Add a file to the **staging** area. Use **.** instead of full file path, to add all changes files from current directory down into directory tree.

```
$ git reset [file]
```

Get file back from **staging** area to working directory.

```
$ git commit
```

Create new commit from changes added to the staging area. Commit **must have** a message!

```
$ git rm [file]
```

Remove file from **working directory** and add deletion to **staging area**.

```
$ git stash
```

Put your current changes into **stash.**

```
$ git stash pop
```

Apply stored **stash** content into **working directory**, and clear **stash**.

```
$ git stash drop
```

Clear **stash** without applying it into **working directory**.

A. GIT INSTALLATION

For GNU/Linux distributions Git should be available in the standard system repository. For example in Debian/Ubuntu please type in the terminal:

```
$ sudo apt-get install git
```

If you want or need to install Git from source, you can get it from https://git-scm.com/downloads.

An excellent Git course can be found in the great **Pro Git** book by Scott Chacon and Ben Straub. The book is available online for free at https://git-scm.com/book.

4. GIT BRANCHING MODEL

```
$ git branch [-a]
```

List all local branches in repository. With **-a**: show all branches (with remote).

```
$ git branch [name]
```

Create new branch, referencing the current **HEAD**.

```
$ git checkout [-b] [name]
```

Switch **working directory** to the specified branch. With **-b**: Git will create the specified branch if it does not exist.

```
$ git merge [from name]
```

Join specified **[from name]** branch into your current branch (the one you are on currenlty).

```
$ git branch -d [name]
```

Remove selected branch, if it is already merged into any other. **-D** instead of **-d** forces deletion.

GIT CHEAT SHEET

5. REVIEW YOUR WORK

```
$ git log [-n count]
```

List commit history of current branch. **-n count** limits list to last **n** commits.

```
$ git log --oneline --graph --decorate
```

An overview with references labels and history graph. One commit per line.

```
$ git log ref..
```

List commits that are present on current branch and not merged into **ref**. A **ref** can be e.g. a branch name or a tag name.

```
$ git log ..ref
```

List commit, that are present on **ref** and not merged into current branch.

```
$ git reflog
```

List operations (like checkouts, commits etc.) made on local repository.

8. SYNCHRONIZING REPOSITORIES

```
$ git fetch [remote]
```

Fetch changes from the **remote**, but not update tracking branches.

```
$ git fetch --prune [remote]
```

Remove remote refs, that were removed from the **remote** repository.

```
$ git pull [remote]
```

Fetch changes from the **remote** and merge current branch with its upstream.

```
$ git push [--tags] [remote]
```

Push local changes to the **remote**. Use **--tags** to push tags.

```
$ git push -u [remote] [branch]
```

Push local branch to **remote** repository. Set its copy as an upstream.

6. TAGGING KNOWN COMMITS

```
$ git tag
```

List all tags.

```
$ git tag [name] [commit sha]
```

Create a tag reference named **name** for current commit. Add **commit sha** to tag a specific commit instead of current one.

```
$ git tag -a [name] [commit sha]
```

Create a tag object named **name** for current commit.

```
$ git tag -d [name]
```

Remove a tag from a local repository.

7. REVERTING CHANGES

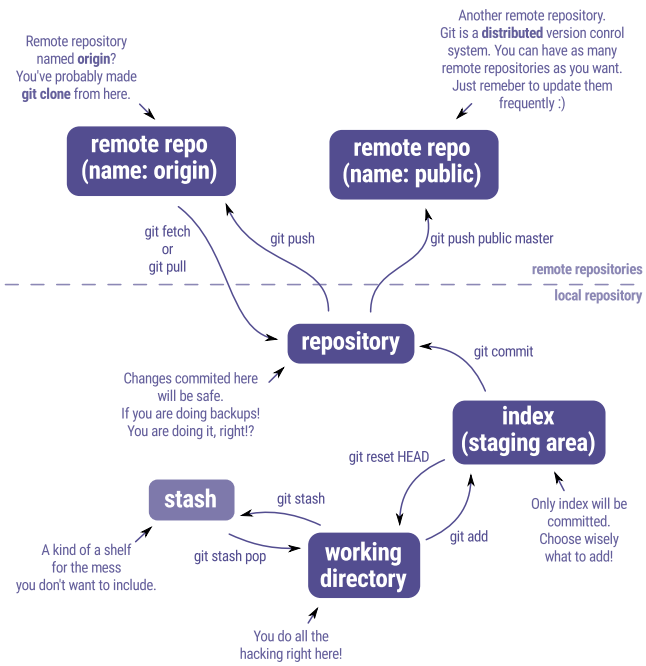
```
$ git reset [--hard] [target reference]
```

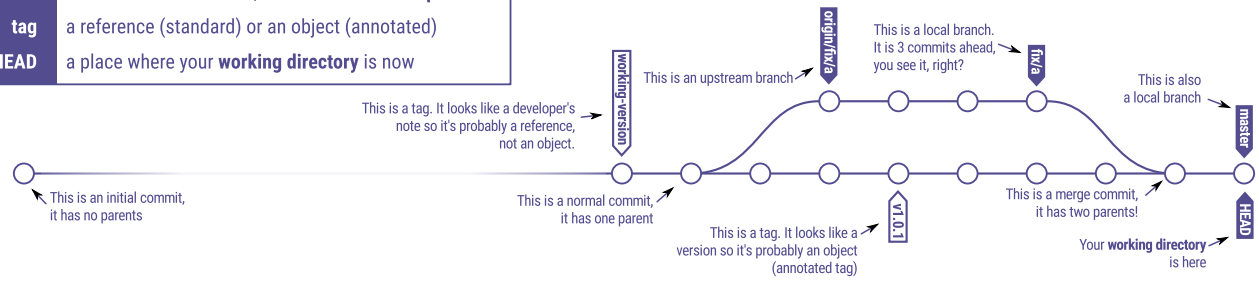
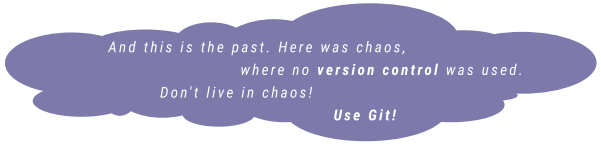
Switch current branch to the **target reference**, and leaves a difference as an uncommited changes. When **--hard** is used, all changes are discarded.

```
$ git revert [commit sha]
```

Create a new commit, reverting changes from the specified commit. It generates an **inversion** of changes.

C. THE ZOO OF WORKING AREAS



D. COMMITS, BRANCHES AND TAGS

commit	an object
branch	a reference to a commit; can have a **tracked upstream**
tag	a reference (standard) or an object (annotated)
HEAD	a place where your **working directory** is now