Resources

- Git howto.pdf
- Git presentations (first time users, basic usage, advance usage)
- GitHub help https://help.github.com/
- Git book http://git-scm.com/book/en/v2

Basic usage

The good practice procedure consists of the following workflow:

Setup to use a GitHub repository

Fork the original upstream repository

First of all, you need to `fork` to get your own personal repository copied to your GitHub user account. When you fork a repo on Github, you're essentially making a copy of a repo at a particular point in time to your own account on Github.

- Log into GitHub and go to the directory you wish to fork.
- Click on the Fork button towards the top right.
- You will be prompted to select where you would like the copy to be made -- click on your personal icon.
- This will create a tracked copy of the `<upstream>/<repo_name>` repository in `<your_username>/<repo_name>`

The dashboard for you fork will indicate that it was forked from somewhere. Now you will be able to contribute to `<your_username>/<repo_name>` which in itself is a totally independent repository that is linked with `<upstream>/<repo_name>`. Whatever changes you make in your fork,`<your_username>/<repo_name>,` will not affect the original repository, `<upstream>/<repo_name>,` and you can build and test until you are satisfied and only then merge with the original to add your contribution to `<upstream>/<repo_name>`.

Merge Note: All users will make changes to their own personal forks, `<your_username>/<repo_name>`, once the changes are ready to be contributed to the original repository the user will request one of the `<upstream>/<repo_name>` administrators to pull your changes from `<your_username>/<repo_name>` to `<upstream>/<repo_name>`.

Clone the fork on your computer

After you have created your own copy of the repository by forking the repo, you create a local repository on your machine by cloning it. Cloning the repository will create a remote called `origin` that points to your forked repository on GitHub. Now you can make changes and do commits however you like, without any fear of breaking the original repo.

Clone Note: We recommend the use of `ssh keys` to control access to GitHub. The following help file shows how to setup key access
https://help.github.com/articles/generating-ssh-keys/.
Remember to use your GitHub login password when you create this key.

Open a terminal and create or go to a location on your local system where you want the have your local `<repo_name>` repository. For `ssh` access:

```
git clone <git@github.com:path/to/repository>
git clone
git@github.com:<your_username>/<repo_name>.git
```

This will create a directory called `<repo_name>` in your current working directory with the `master` branch of `<your_username>/<repo_name>`. Go into the directory to add your work.
Let's investigate:

```
cd <repo_name>
git status
git remote -v
git branch -v -a
```

Add the `upstream` remote pointing to the original repository

You will remember that a fork is an entirely independent repository that is separated from the original "central" repository. There aren't any built-in ways of automatically getting updates from the original repo after you forked it. One thing you need to do if you want to keep up with the main development is to add the original GitHub repository as a remote, and the naming convention for this remote is `upstream`.

```
git remote add <remote_name> <repo_url>
git remote add upstream
git@github.com:<upstream>/<repo_name>.git
```

Let's investigate:

```
git remote -v
git branch -v -a
```

Now that your fork is connected to the original repository, you keep a fork in sync with the original project by fetching and merging. You should regularly fetch information from upstream and merge those changes into your own repo. Fetch will only connect to the remote repository and download the latest commits to your history, merge will actually put those commits into one of your branches and merge them with your own commits. If two commits try to change the same piece of code, you get what is called a merge conflict and you will have to manually look at the files and determine which version you want to keep. You can use the pull command to do both fetch and merge.

```
git fetch upstream
git merge upstream/master
or
git pull upstream master
```

Alternatively do `git pull --ff-only upstream master`, the `--ff-only` option ensures that in case you accidentally did make any changes to your fork's `master` branch, the pull will not work, and you first have to clean up.

After a successful pull from `upstream` your local repository on your machine is up to date with the original repository, make sure that you to push the changes to sync your fork on GitHub.

```
git push origin master
```

This *pull/push* sequence is something you should repeat fairly frequently, in particular when you start working on a new stuff. It makes sure you're up to date with what's going on in the original repository.

Advance Usage Warning:
Git has a unique feature called `rebase` which allows a contributor to update from the `origin`, and then deliver a changeset that is easy to merge. Rebase takes away the whole history of code exchanges leading up to the delivery of the changeset and puts all of the changes into one package.

The rebase allows you to make sure your changes are straightforward, making your pulling request that more easy when you want the maintainer of the original project to include your patches in his project.

To rebase your current development on top of the remote branch you got updated from the `fetch/pull` request, the correct workflow is to `git pull --rebase upstream master` (rebase your work on top of new commits from upstream), and then `git push --force origin master`, in order to rewrite the history in such a way your own commits are always on top of the commits from the original (`upstream`) repo.

Typical Git interaction

This process will be repeated for every new development/feature, and the one after that, and after that... you get the idea.

Checkout the `master` branch

If you've just started or cloned a git project, you'll find you typically have a branch named `master`. There is nothing special about this branch other than it's the conventional way to name the main development branch. You use the `checkout` command to jump between branches.

Let's investigate:

```
git branch -v -a
git status
```

If you are on a different branch, jump to the `master` branch before sync.

```
git status
```

Pull from the `upstream` remote to merge updates

Before you start merging/pulling any changes and before you create a new branch, it is always a good idea to sync with the original repository to `fetch` and `merge` any changes from `upstream`. See Add the upstream remote pointing to the original repository for detail. You can use the pull command to do both fetch and merge.

```
git pull upstream master
```

Create a topic branch for your changes

When you are doing individual work, you can make changes, commits and merges directly in the `master` trunk. When you want to start development in some of the shared directories it is advisable that you create a branch. Since these directories are more likely to have multiple people using and contributing to them, we suggest that you always keep the `master` branch in a working version at all times.

In other words you will create a branch for your idea, commit your work there and do your testing, and when everything works as intended you merge the branch back into your `master`.

Before creating a new branch `pull` the changes from `upstream`, your `master` needs to be up to date. Then create a branch and switch to it

```
git branch <branch_name>
git checkout <branch_name>
git branch mytestbranch
git checkout mytestbranch
or
git checkout -b mytestbranch
```

you only need the `-b` flag once if you are creating a new branch that does not already exist (i.e. is not listed when you do `git branch -v -a`).

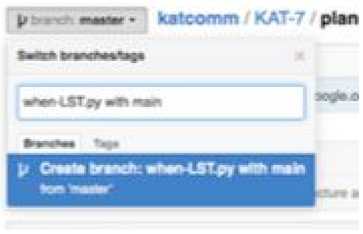
Before you start changing or adding, you can always verify you are on the right branch

```
git branch -v -a
or
git status
```

Branch Note: So far this branch only exist on your locale machine. If you look at your `<user_name>/<repo_name>` repository on GitHub you will see that the branch is not showing there, this is because you still need to `push` it if you want to have it on GitHub.

Advance Usage Warning:

When you want to contribute directly to a shared repository that you have write access to you can create a branch from GitHub



To get the branch on you local clone: `git fetch origin`
Switch to the branch: `git checkout [-b] <branch_name> [origin/<branch_name>]`
Once you have all your commits you can `push` to your branch on your fork on Github:
`git push origin <branch_name>`

Make your changes and commit to the topic branch

When you have made some changes that you want to save, you `add` and `commit` them to your repository. Commits are logical chunks of changes that gets saved in sequence, forming a history that you later can go through, so take your time and write good descriptive commit messages. Since commits can be moved around between branches and repositories it is a good idea to **be descriptive**.

```
git add <filename>
git commit -m "<descriptive message>"
```

A file is only version controlled in a branch once it has been committed to that branch. The `commit` response will show in which branch the changes have been added.

Commit Note: Remember committing is always a two step process: `add` followed by `commit`.

You always use the `checkout` command to switch between branches, you checkout one branch and then checkout the other.

Let's investigate:
Change back to the `master` branch. When you list the files in the current working directory you will see that the file you added does not appear.
Change back to your development branch. Now when you list the files the new file will be shown.

```
git checkout master
ls
git checkout mytestbranch
ls
```

Push your changes back to your fork on GitHub

Once you are happy with the status of your development, you can merge the changes from your branch back into the `master` branch without breaking the `master`.

As always, before merging we need to ensure that the `master` is in sync. This involves pulling any changes from the main repository that have been added since you last fetched and merged.

```
git checkout master
git pull upstream master
```

This should complete without any problems or conflicts. Next, rebase your local branch against the now up-to-date master:

```
git checkout mytestbranch
git rebase master
```

What `rebase` does is stash your changes, grab the latest from master, and then apply your changes on top. This effectively makes it as if rather than branching several days ago, you just branched from the tip of the master and made your commits there.

Most of the time, this will work without a hitch. However, you may occasionally get a merge conflict if the same file has been modified in the same place on both branches. If this happens, Git will complain with a very loud and verbose error message. The important thing with Git is always to read the end of the message. You will then see that it tells you to resolve the conflict.

You'll now be able to do a clean merge from your local branch to master:

```
git checkout master
git merge mytestbranch
```

When you commit to your local repository, commits only exists on your machine. If you want to share your code with others, you can `push` it to repositories where you have access. Your own GitHub fork `<your_username>/<repo_name>`, called `origin`, is of course one. Push to `origin` and your commits will now be on GitHub.

```
git push origin master
```

Send a pull request from GitHub to the original repository

If you want to contribute to the original repo where you don't have push access,you can send them a pull request on GitHub. This will request the administrators of the original repository to pull your changes from `<your_username>/<repo_name>` to `<upstream>/<repo_name>`.

After issuing a pull request, do not close the request until you have received an email indicating that your changes have been accepted and merge into the upstream repo.

Changes to any of the other directories will be reviewed by the maintainers and if needed you may be asked to make some changes.

Merge `upstream` updates with your fork

After the Pull Request has been accepted and merged into the upstream `<upstream>/<repo_name>` repository the forked repository `<your_username>/<repo_name>` will fall behind.

You need to get back in sync with the upstream repository by pulling from `upstream` and merging back into your fork.

```
git checkout master
git pull upstream master
git push origin master
```

The GitHub fork should now be up to date

After you have successfully merged with the master branch, remove the development branch because it is no longer needed:

```
git branch -D mytestbranch
```