fournova

Understanding the Workflow of Version Control

presented by **TOWER** > Version control with Git - made easy

The Basics

1

Start a New Project

```
$ git init
```

Executing the "git init" command in the root folder of your new project creates a new and empty Git repository. You're ready to start getting your files under version control!

Work on an Existing Project

```
$ git clone <remote-url>
```

The "git clone" command is used to download a copy of an existing repository from a remote server. When this is done, you have a full-featured version of the project on your local computer – including its complete history of changes.

2

Work on Your Files

Modify, rename and delete files or add new ones. Do all of this in your favorite editor / IDE / file browser – there's nothing to watch out for in this step!

File Status

Files that aren't yet under version control are called "**untracked**"...

...while files that your version control system already knows about are "**tracked**" files.

A tracked file can either be "**unmodified**" (meaning it wasn't changed since the last commit)...

...or "**modified**" (meaning it has local changes since it was last committed).

3

Keep the Overview

```
$ git status
```

The "git status" command tells you what happened since the last commit: which files did you change? Did you create any new ones or delete old ones?

```
$ git status
#
# Changes not staged for commit:
#     modified:   about.html
#     deleted:    robots.txt
#
# Untracked files:
#     login.html
#
no changes added to commit
```

4

Add Files to the "Staging Area"

```
$ git add <filename>
```

Only because a file was changed doesn't mean it will be part of the next commit! Instead, you have to explicitly decide which changes you want to include. To do this, you add them to the so-called "Staging Area" with the "git add" command.

```
$ git add about.html
#
# Changes to be committed:
#     modified:   about.html
#
# Changes not staged for commit:
#     deleted:    robots.txt
#
# Untracked files:
#     login.html
```

5

Commit all Staged Changes

```
$ git commit -m "message"
```

A commit wraps up all the changes you previously staged with the "git add" command. To record this set of changes in Git's database, you execute the "git commit" command with a short and informative message.

```
$ git commit -m "Updated about page"

[master 9d3f32b] Updated about page
 1 file changed, 29 insertions(+)
```

6

Keep the Overview

```
$ git status
```

Running the "git status" command right after a commit proves to you: only the changes that you added to the Staging Area were committed.

All other changes have been left as local changes: you can continue to work with them and commit or discard them later.

```
$ git status
#
# Changes not staged for commit:
#     deleted:    robots.txt
#
# Untracked files:
#     login.html
#
no changes added to commit
```

7

Inspect the Commit History

```
$ git log
```

The "git log" command lists all the commits that were saved in chronological order. This allows you to see which changes were made in detail and helps you comprehend how the project evolved.

```
$ git log

commit 9d3f32ba002110ee0022fe6d2c5308
Author: Tobias Günther <tg@fournova.c
Date:   Mon Jul 8 09:56:33 2013 +0200

    Updated about page
```

Branching & Merging

1

Start a New Feature

```
$ git branch <new-branch-name>
```

Whenever you start a new feature, a new experiment or a new bugfix, you should create a new branch. In Git, this is extremely fast and easy: just call "git branch <new-branch-name>" and you have a new, separate context.

Don't be shy about creating new branches: it costs you nothing.

Understanding Branches

We often have to work on multiple things in parallel: feature X, bugfix #32, feature Y... This makes it all too easy to lose track of where each change belongs. Therefore, it's essential to keep these contexts separate from each other.

Grouping related changes in their own context has multiple benefits: your coworkers can better understand what happened because they only have to look at code that really concerns them. And you can stay relaxed, because when you mess up, you mess up only this context.

Branches do just this: they provide a context that keeps your work and your changes separate from any other context.

2

Switch Contexts

```
$ git checkout <new-branch-name>
```

To start working on a different context, you need to tell Git that you want to switch to it. You do this by "checking out" the branch with the "git checkout" command.

Every commit you make – until you switch branches again – will be recorded in this branch and kept separate from your other contexts.

HEAD Branch

At each point in time, you can only work in one context – the context of the currently checked out branch (which is also called the "HEAD" branch in Git).

Your project's working directory contains the files that correspond to this branch. When you check out a different branch (make it "HEAD"), Git replaces the files in your working directory with the ones that match this branch.

3

Integrate Changes

```
$ git merge <branch-to-integrate>
```

When your new feature is ready, you might want to integrate it into another branch (e.g. your production or testing branch).

First, switch to the branch that is supposed to receive these changes. Then, call the "git merge" command with the name of the branch you want to integrate.

Sharing Work via Remote Repositories

1

Track a Remote Branch

```
$ git checkout --track
  <remote/branch>
```

If there's an interesting remote branch that you want to work on, you can easily get your own local copy. Use the "git checkout" command and tell it which remote branch you want your new local branch to base off.

Publish a Local Branch

```
$ git push -u <remote>
  <local-branch>
```

To share one of your local branches with your teammates, you need to publish it on a remote server with the "git push" command.

2

Stay Up-To-Date About Remote Changes

```
$ git fetch <remote>
```

When collaborating with others on a project, you'll want to stay informed about their changes. The "git fetch" command downloads new changes from a remote repository – but doesn't integrate them into your local working copy. It only informs you about what happened on the remote, leaving the decision on what to integrate to you.

Local & Remote Repositories

As Git is a so-called "decentralized" version control system, a remote repository is optional. In fact, everything we did until now happened on your local machine, in your local repository – no internet/network connection was necessary.

However, if you want to collaborate with others, you need a remote repository on a server. You don't have to share all of your work though: you can decide for each of your local branches if you want to share it or not.

3

Integrate Remote Changes

```
$ git pull
```

To integrate new changes from the remote repository, you simply call "git pull".

This will update your current HEAD branch with new data from its counterpart branch on the remote. The changes will be directly merged into your local working copy.

4

Upload Local Changes to the Remote Server

```
$ git push
```

To upload the local changes you made in your current HEAD branch, all you have to do is call "git push".