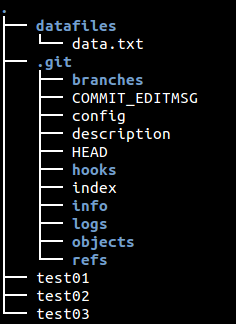
**10.2. Directory structure**

Your directory contains the Git repository as well as the Git working tree for your files. This directory structure is depicted in the following screenshot.



**11. Remove files and adjust the last commit**

**11.1. Remove files**

If you delete a file which is under version control, git add . does not record this file deletion.

You can use the git rm command to delete the file from your working tree and record the deletion of the file in the staging area.

*# create a file and commit it*

**touch** nonsense2.txt

git add .

git commit -m "more nonsense"

*# remove the file and record the deletion in Git*

git rm nonsense2.txt

*# commit the removal*

git commit -m "Removes nonsense2.txt file"

**Tip**

Alternatively to the git rm command you can use the git commit command with the -a flag or the -Aflag in the git add command. This flag adds changes of files known by the Git repository to the commit in case of the git commit command. In case of the git add command it adds all file changes including deletions to the staging area.

For this test, commit a new file and remove it afterwards.

*# create a file and put it under version control*

**touch** nonsense.txt

git add .

git commit -m "create a new file"

*# remove the file*

**rm** nonsense.txt

*# show status, output listed below the command*

git status

*# on branch master*

*# Changes not staged for commit:*

*# (use "git add/rm <file>..." to update what will be committed)*

*# (use "git checkout -- <file>..." to discard changes in working directory)*

*#*

*# deleted: nonsense.txt*

*#*

*# no changes added to commit (use "git add" and/or "git commit -a")*

*# try standard way of committing -> will NOT work*

*# output of the command listed below*

git add .

git commit -m "this does NOT remove the file"

*# On branch master*

*# Changes not staged for commit:*

*# (use "git add/rm <file>..." to update what will be committed)*

*# (use "git checkout -- <file>..." to discard changes in working directory)*

*#*

*# deleted: nonsense.txt*

*#*

*# no changes added to commit (use "git add" and/or "git commit -a")*

After validating that this command does not remove the file from the Git repository you can use the *-a*parameter. Be aware that the *-a* adds other changes, too.

*# commit the remove with the -a flag*

git commit -a -m "File nonsense.txt is now removed"

*# alternatively you could add deleted files to the staging area via*

*# git add -A .*

*# git commit -m "remove file nonsense.txt"*

**11.2. Remove a file from the staging area**

You can use the git reset [filename] command to remove a file from the staging area, which you added with git add [filename]. Removing a file from the staging area, avoids that it included in the next commit.

*# create a file and add to staging area*

**touch** unwantedstaged.txt

git add unwantedstaged.txt

*# remove it from the staging area*

git reset unwantedstaged.txt

*# to cleanup, delete it*

**rm** unwantedstaged.txt

**11.3. Correct the last commit with git amend**

The git commit --amend command makes it possible to replace the last commit. This allows you to change the last commit including the commit message.

**Note**

The old commit is still available until a clean-up job removes it. See [**Section 31.2, “git reflog”**](http://www.vogella.com/tutorials/Git/article.html#gitreflog_definition) for details.

Assume the last commit message was incorrect as it contained a typo. The following command corrects this via the --amendparameter.

*# assuming you have something to commit*

git commit -m "message with a tpyo here"

git commit --amend -m "More changes - now correct"

You should use the git --amend command only for commits which have not been pushed to a public branch of another Git repository. The git --amend command creates a new commit ID and people may have based their work already on the existing commit. In this case they would need to migrate their work based on the new commit.

**11.4. Remove files based on .gitignore changes**

Sometimes you change your .gitignore file. Git will stop tracking the new entries from this moment. The last version is still in the Git repository.

If you want to remove the last version of the files from your Git repository, you need to do this explicitly via the following command.

*# remove directory .metadata from git repo*

git rm -r --cached .metadata

*# remove file test.txt from repo*

git rm --cached **test**.txt

**Note**

This does not remove the file from the repository history. If the file should also be removed from the history, have a look at git filter-branch which allows you to rewrite the commit history. See[**Section 43.1, “Using git filter-branch”**](http://www.vogella.com/tutorials/Git/article.html#filterbranch_definition) for details.

**12. Remote repositories**

**12.1. What are remotes?**

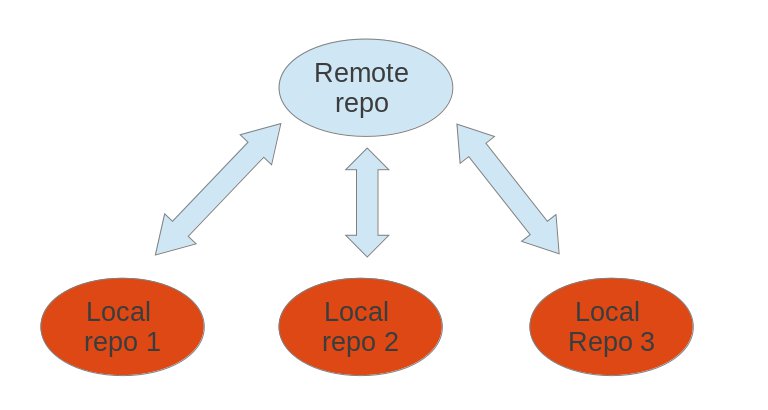
*Remotes* are URLs in a Git repository to other remote repositories that are hosted on the Internet, locally or in the network.

Such *remotes* can be used to synchronize the changes of several Git repositories. A local Git repository can be connected to multiple remote repositories and you can synchronize your local repository with them via Git operations.

**Note**

Think of *remotes* as shorter bookmarks for repositories. You can always connect to a remote repository if you know its URL and if you have access to it. Without *remotes* the user would have to type the URL for each and every command which communicates with another repository.

It is possible that users connect their individual repositories directly, but a typically Git workflow involves one or more remote repositories which are used to synchronize the individual repository. Typically the remote repository which is used for synchronization is located on a server which is always available.



**Tip**

A remote repository can also be hosted in the local file system.

**12.2. Bare repositories**

A remote repository on a server typically does not require a *working tree*. A Git repository without a *working tree* is called a *bare repository*. You can create such a repository with the *--bare* option. The command to create a new empty bare remote repository is displayed below.

*# create a bare repository*

git init --bare

By convention the name of a bare repository should end with the *.git* extension.

**12.3. Setting up a bare Git repository**

In this section you create a bare Git repository. In order to simplify the following examples, the Git repository is hosted locally in the filesystem and not on a server in the Internet.

**Note**

To create a bare Git repository in the Internet you would, for example, connect to your server via the ssh protocol or you use some Git hosting platform, e.g. Github.com.

Execute the following commands to create a bare repository based on your existing Git repository.

*# switch to the first repository*

**cd** ~/repo01

*# create a new bare repository by cloning the first one*

git clone --bare . ../remote-repository.git

*# check the content of the git repo, it is similar*

*# to the .git directory in repo01*

*# files might be packed in the bare repository*

**ls** ~/remote-repository.git

**Tip**

You can convert a normal Git repository into a bare repository by moving the content of the *.git* folder into the root of the repository and removing all others files from the working tree. Afterwards you need to update the Git repository configuration with the git config core.bare true command. The problem with this process is that it does not take into account potential future internal changes of Git, hence cloning a repository with the *--bare* option should be preferred.

**12.4. Cloning and the remote called "origin"**

If you clone a repository, Git implicitly creates a *remote* named *origin* by default. The *origin* *remote* links back to the cloned repository.

If you create a Git repository from scratch with the git init command, the *origin* remote is not created automatically.

**12.5. Adding more remote repositories**

You add more *remotes* to your repository with the git remote add command.

You created a new Git repository from scratch earlier. Use the following command to add a pointer to your new bare repository using the *origin* name.

*# add ../remote-repository.git with the name origin*

git remote add origin ../remote-repository.git

**12.6. Synchronizing with remote repositories**

You can synchronize your local Git repository with remote repositories. These commands are covered in detail in later sections but the following command demonstrates how you can send changes to your remote repository.

*# do some changes*

**echo** "I added a remote repo" > test02

*# commit*

git commit -a -m "This is a test for the new remote origin"

*# to push use the command:*

*# git push [target]*

*# default for [target] is origin*

git push origin

**12.7. Show the existing remotes**

To see the existing definitions of the remote repositories, use the following command.

*# show the details of the remote repo called origin*

git remote show origin

To see the details of the *remotes*, e.g. the URL use the following command.

*# show the existing defined remotes*

git remote

*# show details about the remotes*

git remote -v

**13. Cloning remote repositories and push and pull**

**13.1. Clone your repository**

Clone a repository and checkout a working tree in a new directory via the following commands.

*# switch to home*

**cd** ~

*# make new directory*

**mkdir** repo02

*# switch to new directory*

**cd** ~/repo02

*# clone*

git clone ../remote-repository.git .

**13.2. Push changes to another repository**

The git push command allows you to send data to other repositories. By default it sends data from your current branch to the same branch of the remote repository. See [**Section 16.6, “Push changes of a branch to a remote repository”**](http://www.vogella.com/tutorials/Git/article.html#gitpushbranch) for details on pushing branches or [**Git push manpage**](https://www.kernel.org/pub/software/scm/git/docs/git-push.html) for general information.

Make some changes in your local repository and push them from your first repository to the remote repository via the following commands.

*# make some changes in the first repository*

**cd** ~/repo01

*# make some changes in the file*

**echo** "Hello, hello. Turn your radio on" > test01

**echo** "Bye, bye. Turn your radio off" > test02

*# commit the changes, -a will commit changes for modified files*

*# but will not add automatically new files*

git commit -a -m "Some changes"

*# push the changes*

git push ../remote-repository.git

**Note**

By default you can only push to bare repositories (repositories without working tree). Also you can only push a change to a remote repository which results in a fast-forward merge. See [**Section 34, “Merging”**](http://www.vogella.com/tutorials/Git/article.html#gitmerge) to learn about fast-forward merges.

**13.3. Pull changes**

The git pull command allows you to get the latest changes from another repository for the current branch.

To test this in your example Git repositories, switch to your second repository, pull in the recent changes from the remote repository, make some changes, push them to your remote repository via the following commands.

*# switch to second directory*

**cd** ~/repo02

*# pull in the latest changes of your remote repository*

git pull

*# make changes*

**echo** "A change" > test01

*# commit the changes*

git commit -a -m "A change"

*# push changes to remote repository*

*# origin is automatically created as we cloned original from this repository*

git push origin

You can pull in the changes in your first example repository with the following commands.

*# switch to the first repository and pull in the changes*

**cd** ~/repo01

git pull ../remote-repository.git/

*# check the changes*

git status

**Tip**

The git pull command is actually a shortcut for git fetch followed by the git merge or git rebasecommand depending on your configuration. In [**Section 6.4, “Avoid merge commits for pulling”**](http://www.vogella.com/tutorials/Git/article.html#setup_rebase) you configured your Git repository so that git pull is a fetch followed by a rebase. See [**Section 33.1, “Fetch”**](http://www.vogella.com/tutorials/Git/article.html#gitfetch_intro) for more information about the fetch command.

**14. Online remote repositories**

**14.1. Cloning online repositories**

Git supports several transport protocols to connect to other Git repositories; the native protocol for Git is also called git.

The following command clones an existing repository using the Git protocol. The Git protocol uses the port 9148 which might be blocked by firewalls.

*# switch to a new directory*

**mkdir** ~/online

**cd** ~/online

*# clone online repository*

git clone git://github.com/vogella/gitbook.git

If you have SSH access to a Git repository, you can also use the ssh protocol. The name preceding @ is the user name used for the SSH connection.

*# clone online repository*

git clone **ssh**://git@github.com/vogella/gitbook.git

*# older syntax*

git clone git@github.com:vogella/gitbook.git

Alternatively you could clone the same repository via the http protocol.

*# the following will clone via HTTP*

git clone http://github.com/vogella/gitbook.git

**14.2. Add more remote repositories**

As discussed earlier cloning repository creates a *remote* called origin pointing to the remote repository which you cloned from.

You can push changes to this origin repository via git push as Git uses *origin* as default. Of course, pushing to a remote repository requires write access to this repository.

You can add more *remotes* via the git remote add [name] [URL\_to\_Git\_repo] command. For example, if you cloned the repository from above via the Git protocol, you could add a new remote with the name *github\_http* for the http protocol via the following command.

*# add the HTTPS protocol*

git remote add github\_http https://vogella@github.com/vogella/gitbook.git

**14.3. Rename remote repositories**

To rename an existing remote repository use the git remote rename command. This is demonstrated by the following listing.

*# rename the existing remote repository from*

*# github\_http to github\_testing*

git remote rename github\_http github\_testing

**14.4. Remote operations via HTTP and a proxy**

It is possible to use the HTTP protocol to clone Git repositories. This is especially helpful if your firewall blocks everything except HTTP or HTTPS.

Git also provides support for HTTP access via a proxy server. The following Git command could, for example, clone a repository via HTTP and a proxy. You can either set the proxy variable in general for all applications or set it only for Git.

This example uses environment variables.

*# Linux*

**export** http\_proxy=http://proxy:8080

**export** https\_proxy=https://proxy:8443

*# Windows*

**set** http\_proxy http://proxy:8080

**set** https\_proxy http://proxy:8080

git clone http://dev.eclipse.org/git/org.eclipse.jface/org.eclipse.jface.snippets.git

**Note**

For secured SSL encrypted communication you should use the SSH or HTTPS protocol in order to guarantee security.

This example uses the following Git config settings.

*# set proxy for git globally*

git config --global http.proxy http://proxy:8080

*# to check the proxy settings*

git config --get http.proxy

*# just in case you need to you can also revoke the proxy settings*

git config --global --**unset** http.proxy

**Tip**

Git is able to store different proxy configurations for different domains, see *core.gitProxy* in [**Git config manpage**](http://git-scm.com/docs/git-config).

**15. What are branches?**