Remote repositories

What are remotes?

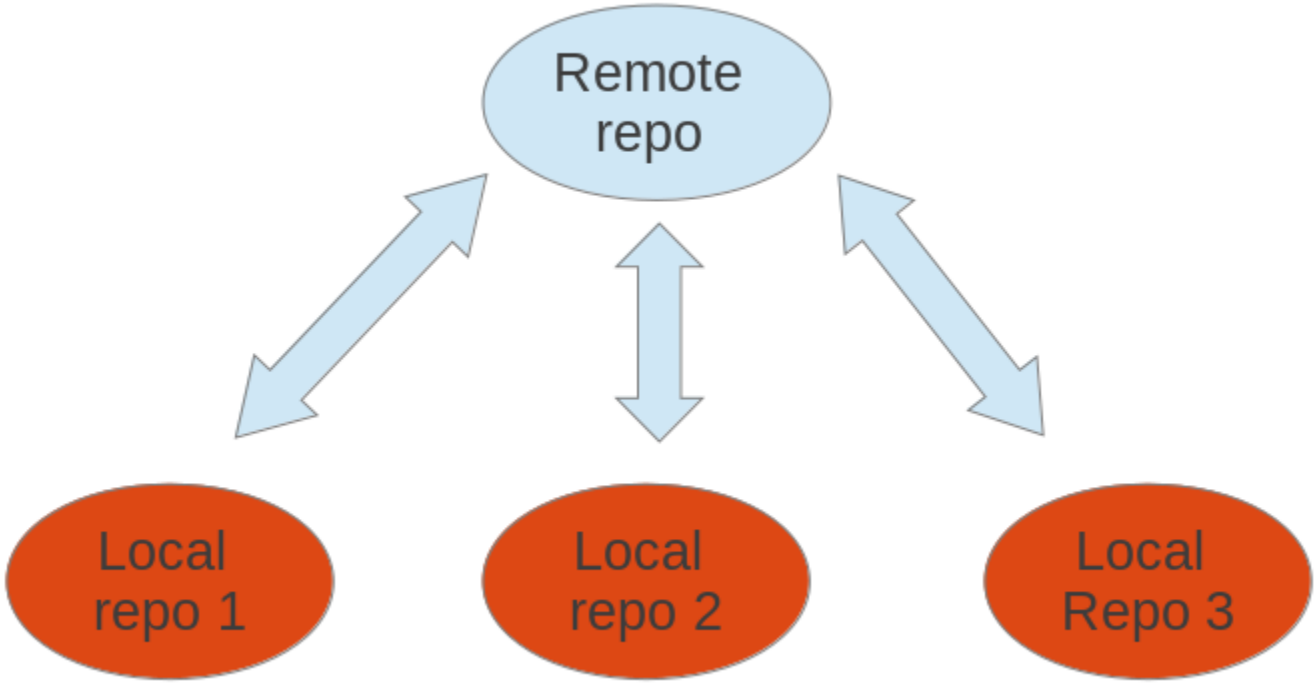
Remotes are URLs in a Git repository to other remote repositories that are hosted on the Internet, locally or on the network.

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.

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.

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.

Convert a Git repository to a bare repository

Converting a normal Git repository to a bare repository is not directly support by Git.

You can convert it manually by moving the content of the `.git` folder into the root of the repository and by 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.

As this is officially not supported, you should prefer cloning a repository with the `--bare` option.

Cloning repositories and the remote called "origin"

Cloning a repository

The `git clone` command copies an existing Git repository. This copy is a working Git repository with the complete history of the cloned repository. It can be used completely isolated from the original repository.

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.

. Exercise: Cloning to create 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.

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

If you receive a warning similar to the following: `push.default is unset; its implicit value is changing in Git 2.0 from 'matching' to 'simple'`

Adding and listing existing remotes

Adding a remote repository

You add as many *remotes* to your repository as desired. For this you use the `git remote add` command.

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

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

. 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
```

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
```

The push and pull commands

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.

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

Pull changes

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

The `git pull` command is actually a shortcut for `git fetch` followed by the `git merge` or the `git rebase` command depending on your configuration you configured your Git repository so that `git pull` is a fetch followed by a rebase.

Exercise: Clone your bare 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 .
```

Exercise: Using the push command

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
```

Exercise: Using the pull command

To test the `git pull` 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
```

Working with remote repositories

Cloning remote 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
```

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 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
```

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
```

Remote operations via HTTP

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 clone http://git.eclipse.org/gitroot/platform/eclipse.platform.ui.git
```

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

Using a proxy

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.

The following listing configures the proxy via environment variables.

```
# Linux and Mac
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://git.eclipse.org/gitroot/platform/eclipse.platform.ui.git
```

The following listing configures the proxy via 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

What are branches?

Git allows you to create *branches*, i.e. named pointers to commits. You can work on different branches independently from each other. The default branch is most often called *master*.

A branch pointer in Git is 41 bytes large, 40 bytes of characters and an additional new line character. Therefore, the creating of branches in Git is very fast and cheap in terms of resource consumption. Git encourages the usage of branches on a regular basis.

If you decide to work on a branch, you *checkout* this branch. This means that Git populates the *working tree* with the version of the files from the commit to which the branch points and moves the *HEAD* pointer to the new branch.

HEAD is a symbolic reference usually pointing to the branch which is currently checked out.

Commands to working with branches

List available branches

The `git branch` command lists all local branches. The currently active branch is marked with `*`.

```
# lists available branches
git branch
```

If you want to see all branches (including remote-tracking branches), use the `-a` for the `git branch` command.

```
# lists all branches including the remote branches
git branch -a
```

The `-v` option lists more information about the branches.

In order to list branches in a remote repository use the `git branch -r` command as demonstrated in the following example.

```
# lists branches in the remote repositories
git branch -r
```

Create new branch

You can create a new branch via the `git branch [newname]` command. This command allows to specify the starting point (commit id, tag, remote or local branch). If not specified the commit to which the HEAD reference points is used to create the branch.

```
# syntax: git branch <name> <hash>
# <hash> in the above is optional
git branch testing
```

Checkout branch

To start working in a branch you have to *checkout* the branch. If you *checkout* a branch, the HEAD pointer moves to the last commit in this branch and the files in the working tree are set to the state of this commit.

The following commands demonstrate how you switch to the branch called *testing*, perform some changes in this branch and switch back to the branch called *master*.

```
# switch to your new branch
git checkout testing

# do some changes
echo "Cool new feature in this branch" > test01
git commit -a -m "new feature"

# switch to the master branch
git checkout master

# check that the content of
# the test01 file is the old one
cat test01
```

To create a branch and to switch to it at the same time you can use the `git checkout` command with the `-b` parameter.

```
# create branch and switch to it
git checkout -b bugreport12

# creates a new branch based on the master branch
# without the last commit
git checkout -b mybranch master~1
```

Rename a branch

Renaming a branch can be done with the following command.

```
# rename branch
git branch -m [old_name] [new_name]
```

Delete a branch

To delete a branch which is not needed anymore, you can use the following command. You may get an error message that there are uncommited changes if you did the previous examples step by step. Use force delete (uppercase `-D`) to delete it anyway.

```
# delete branch testing
git branch -d testing
# force delete testing
git branch -D testing
# check if branch has been deleted
git branch
```

Push changes of a branch to a remote repository

You can push the changes in the current active branch to a remote repository by specifying the target branch. This creates the target branch in the remote repository if it does not yet exist.

```
# push current branch to a branch called "testing" to remote repository
git push origin testing

# switch to the testing branch
git checkout testing

# some changes
echo "News for you" > test01
git commit -a -m "new feature in branch"

# push all including branch
git push
```

This way you can decide which branches you want to push to other repositories and which should be local branches

Differences between branches

To see the difference between two branches you can use the following command.

```
# shows the differences between
# current head of master and your_branch

git diff master your_branch
```

For example, if you compare a branch called *your_branch* with the *master* branch the following command shows the changes in *your_branch* and *master* since these branches diverged.

```
# shows the differences in your
# branch based on the common
# ancestor for both branches

git diff master...your_branch
```

Tags in Git

What are tags?

Git has the option to *tag* a commit in the repository history so that you find it easier at a later point in time. Most commonly, this is used to tag a certain version which has been released.

If you tag a commit, you create an annotated or lightweight tag.

Lightweight and annotated tags

Git supports two different types of tags, lightweight and annotated tags.

A *lightweight tag* is a pointer to a commit, without any additional information about the tag. An *annotated tag* contains additional information about the tag, e.g., the name and email of the person who created the tag, a tagging message and the date of the tagging. *Annotated tags* can also be signed and verified with *GNU Privacy Guard (GPG)*.

Naming conventions for tags

Tags are frequently used to tag the state of a release of the Git repository. In this case they are typically called *release tags*.

Convention is that release tags are labeled based on the [major].[minor].[patch] naming scheme, for example "1.0.0". Several projects also use the "v" prefix.

The idea is that the *patch* version is incremented if (only) backwards compatible bug fixes are introduced, the *minor* version is incremented if new, backwards compatible functionality is introduced to the public API and the *major* version is incremented if any backwards incompatible changes are introduced to the public API.

Working with tags

List tags

You can list the available tags via the following command:

```
git tag
```

Search by pattern for a tag

You can use the `-l` parameter in the `git tag` command to search for a pattern in the tag.

```
git tag -l <pattern>
```

Creating lightweight tags

To create a lightweight tag don't use the `-m`, `-a` or `-s` option.

The term *build* describes the conversion of your source code into another state, e.g., converting Java sources to an executable `JAR` file. Lightweight tags in Git are often used to identify the input for a build. Frequently this does not require additional information other than a build identifier or the timestamp.

```
# create lightweight tag
git tag 1.7.1

# see the tag
git show 1.7.1
```

Creating annotated tags

You can create a new annotated tag via the `git tag -a` command. An annotated tag can also be created using the `-m` parameter, which is used to specify the description of the tag. The following command tags the current active HEAD.

```
# create tag
git tag 1.6.1 -m 'Release 1.6.1'

# show the tag
git show 1.6.1
```

You can also create tags for a certain commit id.

```
git tag 1.5.1 -m 'version 1.5' [commit id]
```

Creating signed tags

You can use the option `-s` to create a signed tag. These tags are signed with *GNU Privacy Guard (GPG)* and can also be verified with GPG. For details on this please see the following URL: **Git tag manpage**.

Checkout tags

If you want to use the code associated with the tag, use:

```
git checkout <tag_name>
```

Warning

If you checkout a tag, you are in the *detached head mode* and commits created in this mode are harder to find after you checkout a branch again.

. Push tags

By default the `git push` command does not transfer tags to remote repositories. You explicitly have to push the tag with the following command.

```
# push a tag or branch called tagname
git push origin [tagname]

# to explicitly push a tag and not a branch
git push origin tag <tagname>

# push all tags
git push --tags
```

. Delete tags

You can delete tags with the `-d` parameter. This deletes the tag from your local repository. By default Git does not push tag deletions to a remote repository, you have to trigger that explicitly.

The following commands demonstrate how to push a tag deletion.

```
# delete tag locally
```

```
git tag -d 1.7.0

# delete tag in remote repository
# called origin
git push origin :refs/tags/1.7.0
```

Listing changed files before a commit

Listing changed files

The `git status` command shows the status of the working tree, i.e., which files have changed, which are staged and which are not part of the staging area. It also shows which files have merge conflicts and gives an indication what the user can do with these changes, e.g., add them to the staging area or remove them, etc.

Example: Using git status

The following commands create some changes in your Git repository.

```
# make some changes
# assumes that the test01
# as well as test02 files exist
# and have been committed in the past
echo "This is a new change to the file" > test01
echo "and this is another new change" > test02

# create a new file
ls > newfileanalyzis.txt
```

The `git status` command shows the current status of your repository and suggests possible actions which the user can perform.

```
# see the current status of your repository
# (which files are changed / new / deleted)
git status
```

The output of the command looks like the following listing.

```
# On branch master
# Your branch is ahead of 'origin/master' by 1 commit.
#   (use "git push" to publish your local commits)
#
# Changes not staged for commit:
```

```
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working
directory)
#
#  modified:   test01
#  modified:   test02
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#  newfileanalyzis.txt
no changes added to commit (use "git add" and/or "git commit -a")
```

Reviewing the changes in the files before a commit

. See the differences in the working tree since the last commit

The `git diff` command allows seeing the changes in the working tree compared to the last commit.

Example: Using "git diff" to see the file changes in the working tree

In order to test this, make some changes to a file and check what the `git diff` command shows to you. Afterwards commit the changes to the repository.

```
# make some changes to the file
echo "This is a change" > test01
echo "and this is another change" > test02

# check the changes via the diff command
git diff

# optional you can also specify a path to filter the displayed changes
# path can be a file or directory
git diff [path]
```

See differences between staging area and last commit

To see which changes you have staged, i.e., you are going to commit with the next commit, use the following command.

```
# make some changes to the file
git diff --cached
```