Git Version Control Tutorial

What is git

Git is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency.

Git is easy to learn and has a tiny footprint with lightning fast performance. It outclasses SCM tools like Subversion, CVS, Perforce, and ClearCase with features like cheap local branching, convenient staging areas, and multiple workflows.

On a more technical level, git is a content-addressable file system. Meaning that at the core of Git is a simple **key-value data store**. What this means is that you can insert any kind of content into a Git repository, for which Git will hand you back a unique key you can use later to retrieve that content.

Source https://git-scm.com/

Target audience

The course material is targeted at technically inclined person with a high Unix proficiency and an interest of peaking under the hood bug had very little or no exposure to Git yet.

A bit of exposure to other version control systems can be beneficial but isn't a requirement.

Scope & Terminology

While working with git you almost certainly encounter the terms **`porcelain`** and **`plumbing`**. Initially git was a toolkit for a version control system many of the commands are meant for user-unfriendly low level work. The low-level commands were designed be chained together, not unlike the Unix pipe, to complete tasks. Hence they are referred to as **`plumbing`**. For most git users the more user-friendly **`porcelain`** are used nearly exclusively.

For most part of this tutorial we stay within the realm of **`porcelain`**.

Table 1. Terminology Summary

Term	Description
`plumbing`	Low-level toolkit commands.
`porcelain`	User-friendly front end commands.

Concepts

File storage

Traditionally version control systems such as Subversion, CVS and Perforce are delta-based version control systems. Meaning they are tracking files and changes to said files over time.

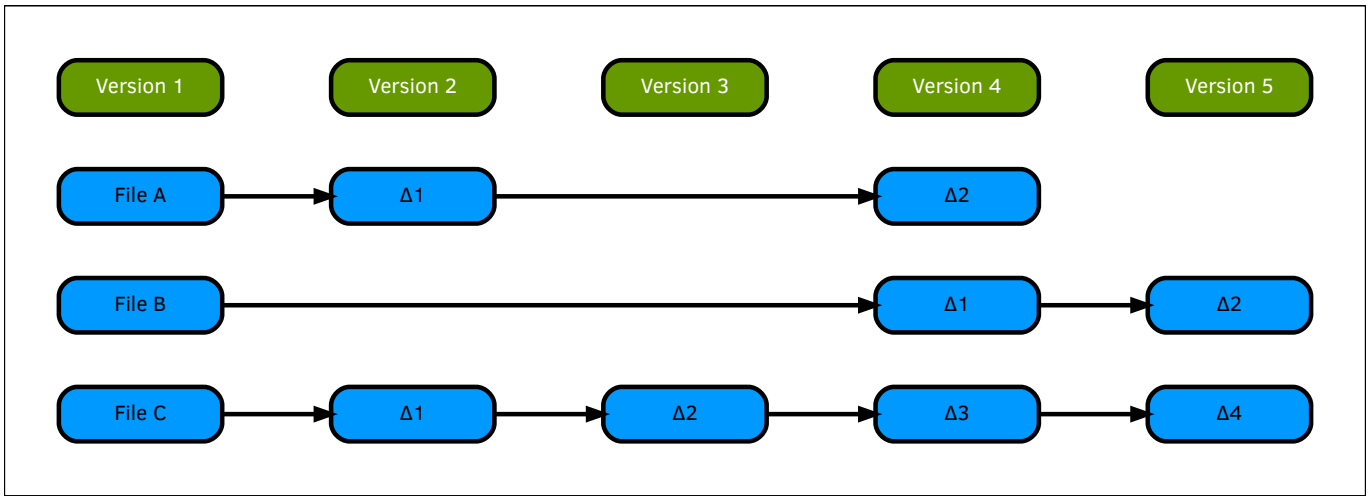

Figure 1. Delta based version control model

Git on the other hand sees it's data more as a **series of snapshots**. So for git every commit is not unlike a picture of the file content at a given time. If a files content has not change only a reference to a snapshot is stored. Git is as such like a mini file system.

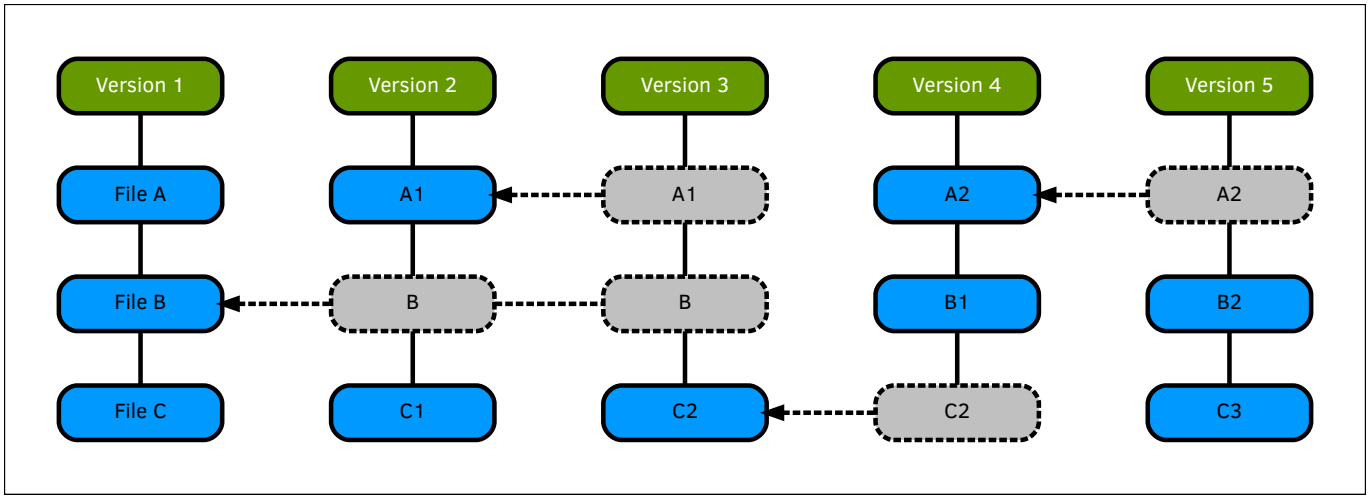

Figure 2. Git snapshots / file storage model

Location

Most git commands are performed locally. Compared to the CVCS (Centralizes Version Control Systems) such as Subversion, git is blazingly fast. Git always retains the whole repository with history locally. Only when sending changes to a remote site is network latency coming into play.

Integrity

SHA1 checksums are a integral factor in git. Everything is checksummed and then referenced by that checksum. File corruption or tampering will not go unnoticed by git.

The 40 character long hex encoded SHA1 checksums are popping up all over when working with git as they are used universally.

SHA1 hash example

```
d6a96ae3b442218a91512b9e1c57b9578b487a0b
```

Content

Git only works with files. Directories without a single file are not retained in git. Also files with the same content are only stored once and then referenced.

Stages

An important part is to understand the state or stage a file in git can be in.

Stage	Description
committed	The content of the file is safely stored in the repository.
staged	A modified file is marked for inclusion with the next commit.
modified	A file git is aware of has been modified but is not commited to the repository yet.

In a Git project the 3 stages are represented by: the .git directory (Repository), the working directory and the staging area.

This results in the following workflow.

- Checkout a repository or branch aka **`git checkout`**.
- Modify a file in the working directory.
- Stage the file to be included during the next commit aka **`git add`**.
- Commit the changed file to the repository aka **`git commit`**.

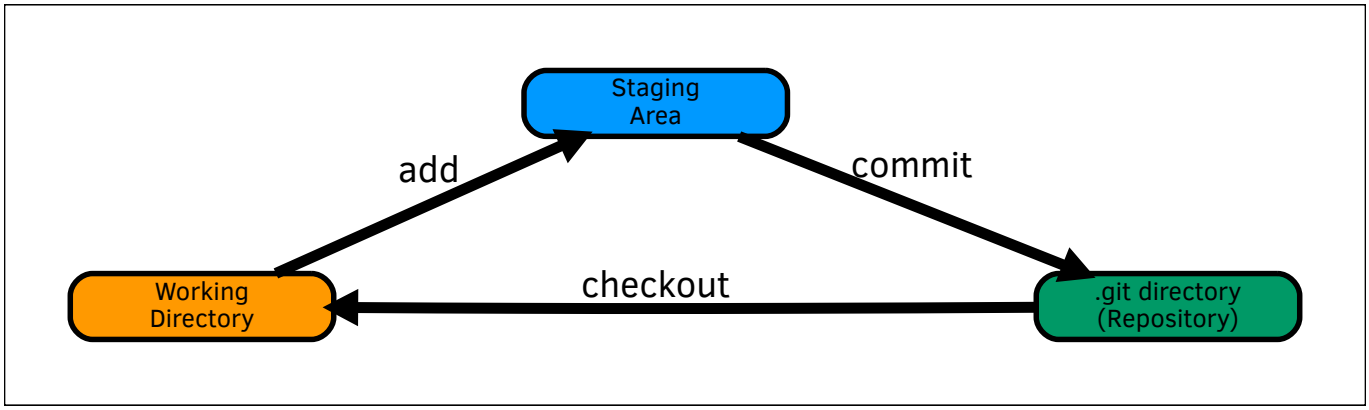

Figure 3. Git stages and workflow

Family tree

Git commits, with a few exceptions, have always at least one parent commit. Remember this concept as we dive deeper into the git universe. The parent determines where a certain commit is located in the hierarchy of the repository.

Understanding the relations between commits is essential for understanding more advanced topics like branching, rebasing and resetting among others. But it also helps with troubleshooting issues.

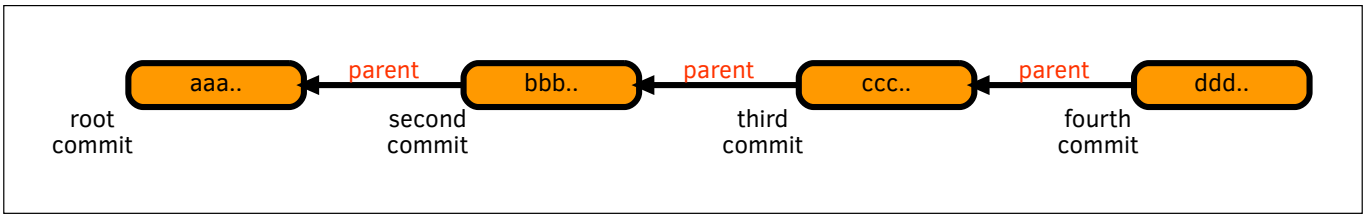

Figure 4. Git commit relation

Repositories

The repository of every git project is stored in the root of the working directory under the **`.git`** directory. This provides a small overview of the content within the directory.

Content

A git repository consists of a few directories and files. After initializing it looks like the directory tree below.

After the first **`commit`** or **`push`** to a remote repository more files and directories appear.

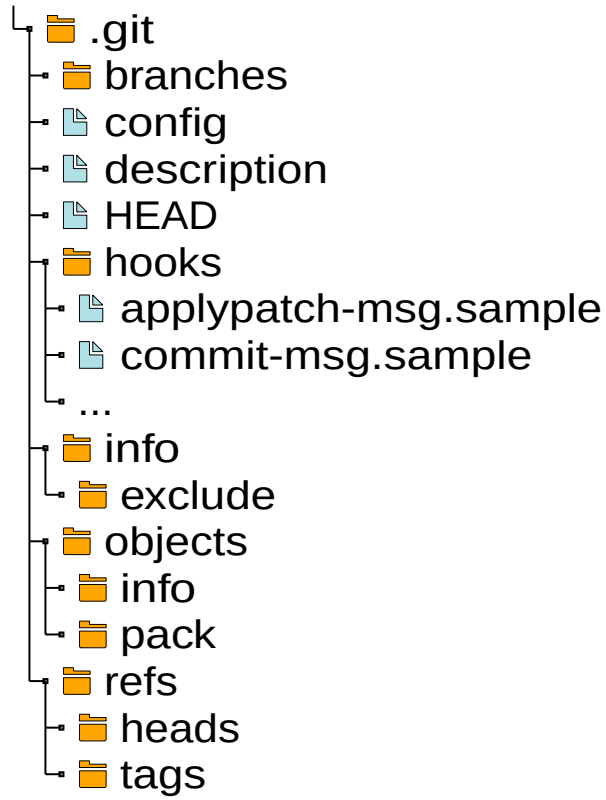

Figure 5. Listing of **`.git`** *directory after initialization*

A overview of what the files and directories are use for in Git.

`branches`	A slightly deprecated way to store URL shorthands used by **`git fetch`**, **`git pull`** and **`git push`**.
`config`	Repository specific configuration file.
`description`	Used for software like **`git-web`** to describe the repository.
`HEAD`	A reference to the **`refs/heads/`** name space describing the currently active branch.
`hooks`	Hooks are customization scripts used by various Git commands.
`info`	Additional information about the repository is recorded in this directory.
`logs`	Missing from the figure above as there are no commits yet. Records of changes made to refs are stored in this directory.
`objects`	Object store associated with this repository see objects for more information.
`refs`	References are stored in sub directories of this directory.

Source gitrepository-layout man page

For a more in depth description of the repository layout consult the **`gitrepository-layout`** manual page or read it online.

Objects

There are 4 types of git objects stored within the **`objects`** directory. Namely:

Commits	Contains the **`commit`** SHA1 hash, the **`tree`** SHA1 hash, the **`parent`** commit, the `author's name, the `committer's name and the commit message.
Trees	Contain references to **`blob`** objects (files) or other **`tree`** objects (directories) together with their file permission and their name.
Blobs	Store the content of a given file.
Tags	Are similar to commit objects but only refer to a commit object and lack the **`tree`** and **`parent`** reference. But additionally there is the **`tag`** name.

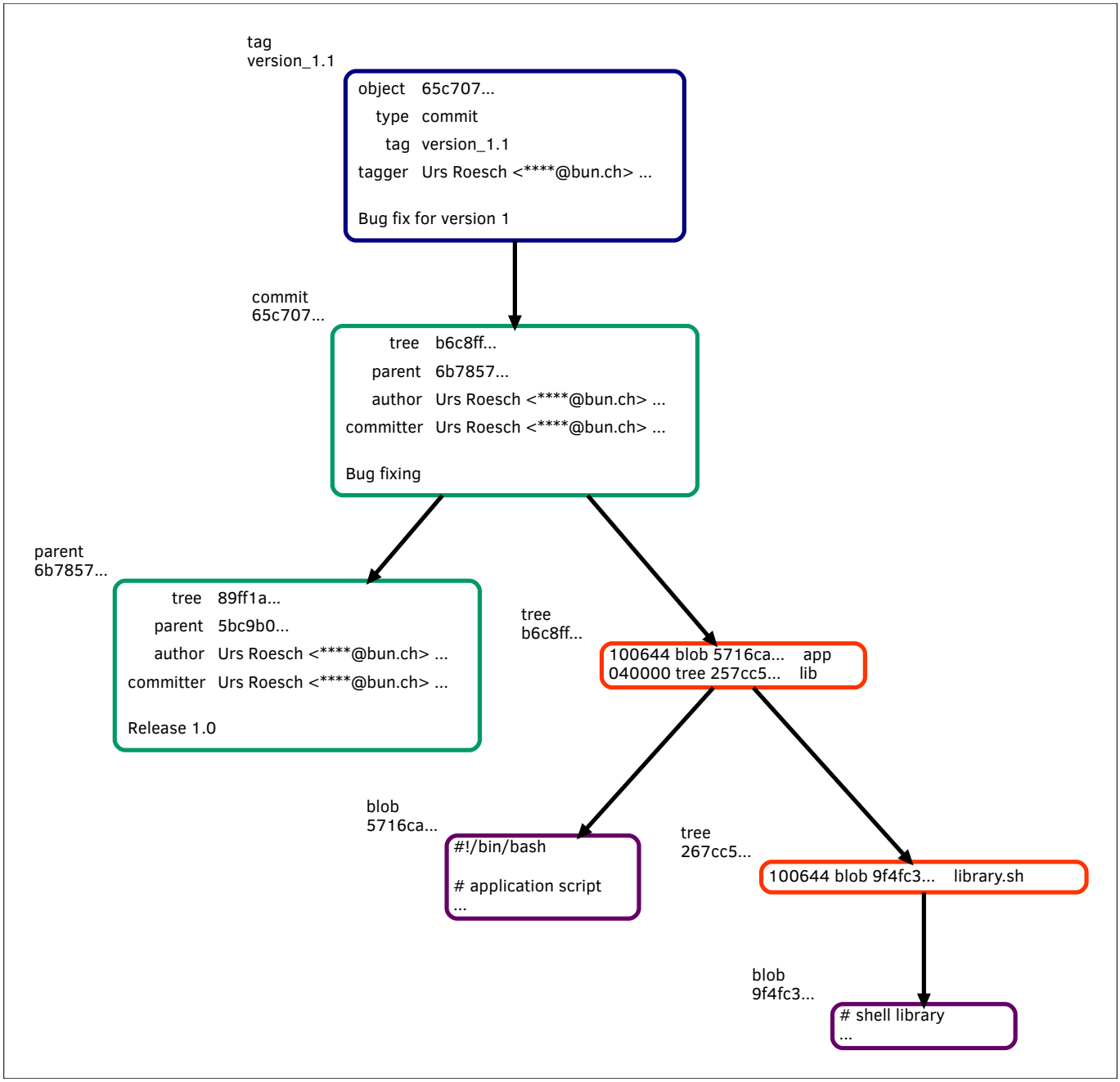

Figure 6. Graphic showing all 4 types of git objects

Getting help

Git is very well documented and comes with extensive help accessible from the command line. There are a few ways getting help with a git command.

Commands

To list all available porcelain commands **`git --help`** or **`git help`** is used.

```
$ git help
usage: git [--version] [--help] [-C <path>] [-c <name>=<value>] ❶
           [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
           [-p | --paginate | -P | --no-pager] [--no-replace-objects] [--bare]
           [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
           <command> [<args>]

These are common Git commands used in various situations: ❷

start a working area (see also: git help tutorial)
   clone             Clone a repository into a new directory
   init              Create an empty Git repository or reinitialize an existing
one

work on the current change (see also: git help everyday)
   add               Add file contents to the index
   mv                Move or rename a file, a directory, or a symlink
   restore           Restore working tree files
   rm                Remove files from the working tree and from the index
   sparse-checkout   Initialize and modify the sparse-checkout

examine the history and state (see also: git help revisions)
   bisect            Use binary search to find the commit that introduced a bug
   diff              Show changes between commits, commit and working tree, etc
   grep              Print lines matching a pattern
   log               Show commit logs
   show              Show various types of objects
   status            Show the working tree status

grow, mark and tweak your common history
   branch            List, create, or delete branches
   commit            Record changes to the repository
   merge             Join two or more development histories together
   rebase            Reapply commits on top of another base tip
   reset             Reset current HEAD to the specified state
   switch            Switch branches
   tag               Create, list, delete or verify a tag object signed with GPG

collaborate (see also: git help workflows)
   fetch             Download objects and refs from another repository
   pull              Fetch from and integrate with another repository or a local
branch
   push              Update remote refs along with associated objects

'git help -a' and 'git help -g' list available sub commands and some ❸
concept guides. See 'git help <command>' or 'git help <concept>'
to read about a specific sub command or concept.
See 'git help git' for an overview of the system.
```

❶ Git's common options.

❷ Git's porcelain commands structured by topic.

❸ Further commands to display more extensive help.

Man pages

To display documentation there are 3 equivalent commands which display the Unix **`man`** page for a given git command when invoked.

To show more help for the **`git init`** command one can either use **`man git-init`**, **`git help init`** or **`git init --help`**. They all open the Unix **`man`** page for the topic at hand.

Shell completion

As there are countless options and switches for the various git commands one can easily lose track. For various modern Unix shells such as **`bash`** or **`zsh`** completion packages exists.

When installed one can hit the `Tab` twice to find suggestions.

```
git init --Tab Tab
--bare                --no-...            --no-template       --quiet
--separate-git-dir=   --shared            --template=
```

Websites

The comprehensive documentation is also available on the net via the official git website's documentation section.

Module 1 - Configuration

As with any advanced software the default values might work for very small tasks but as proficiency rises the need to divert from the default values becomes a necessity. This module scratches only on the surface the git configuration. But the most often used values are certainly covered. After completion of this module one can:

Goals

- Show configuration values.
- Set contact information.
- Differentiate system, global and local configurations
- Useful configuration options.

With recent versions of **`git`** a base configuration for the user's email and full name is required.

List git configuration

Up till now the first repository contains only a couple of files and is used locally. The configuration is therefore in pristine condition. Let's see how to show the current state.

list

```
$ git config --list
core.repositoryformatversion=0 (1)
core.filemode=true (2)
core.bare=false (3)
core.logallrefupdates=true (4)
```

Generally this is the default configuration for a working directory repository. You may skip the explanation of each item as it is not very important at this point in time.

(1) **`0`** is the original git repository format. With git 2.7 extensions were introduced if such extensions exist the version can be bumped to **`1`**.

(2) **`filemode`** is generally set to **`true`** on Unix like systems but for say Windows where the traditional Unix file permissions are not used it should be set to **`false`**. This prevents mismatches of file permission in the index and on disk.

(3) Bare repositories are mostly used on sharing platforms such as GitHub, GitLab or Bitbucket. The structure is different from a working repository.

(4) For working directories the default is **`true`** as all changes should be logged. Bare repositories should have a value of **`false`**.

This is a very minimal configuration indeed. So let's see how one extend it.

Set contact information

The configuration items that should be set is the **`user.name`** and **`user.email`**. As the name suggests these settings are specific to the actual git user as such need to be set individually.

user.name & user.email

```
$ git config --global --add user.name "Urs Roesch" (1)
$ git config --global --add user.email "****@bun.ch" (2)
$ git config --list
user.email=****@bun.ch
user.name=Urs Roesch
core.repositoryformatversion=0
core.filemode=true
core.bare=false
core.logallrefupdates=true
```

(1) Adding the user's full name global configuration file.

(2) Adding the user's email to the global configuration file.

System, global and local?

Peeking into the man page of **`git-config`** or searching the Internet for **`git config`** values quite often the switches **`--global`** or **`--local`** appear. Shedding a bit light into this matter is the purpose of this section.

File location

system	System wide configuration located usually under **`/etc/gitconfig`**.
global	User wide configuration located under **`${HOME}/.gitconfig`**.
local	Repository only configuration under **`.git/config`**.

Important here is that configuration values in **`local`** context take precedence over **`global`** ones. And **`global`** values override **`system`** ones.

How does it work?

First with the information about file location take a peek into the two configuration that exist so far. Namely **`global`** with the email addresses and **`local`** in the repository.

```
$ cat ~/.gitconfig
[user]
        email = ****@bun.ch
        name = Urs Roesch
$ cat .git/config
[core]
        repositoryformatversion = 0
        filemode = true
        bare = false
        logallrefupdates = true
```

The format of the files is **`ini`** based. A very popular key value store. Which can be changed with a text editor. The sections **`user`** and **`core`** are being prepended to the keys like **`email`** or **`filemode`** resulting in **`user.email`** or **`core.filemode`**.

There is no overlap in the two configurations. The global configuration is simply appended to the local one.

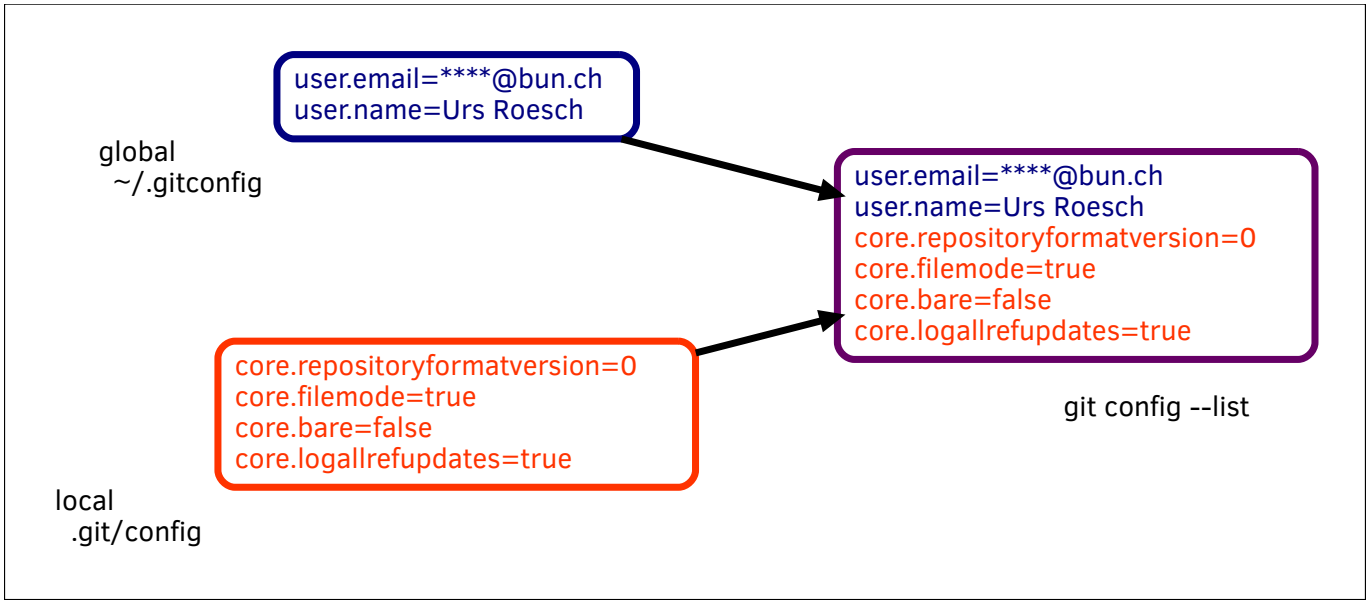

Figure 7. Combining the **`global`** *and* **`local`** *configuration with no overlap.*

To use a real world example. While having a **`user.name`** and **`user.email`** configured globally, for GitHub repositories the **`user.email`** should be matching the one registered at the site.

In the below figure the **`user.email`** key from **`local`** takes precedence over the one from **`global`**.

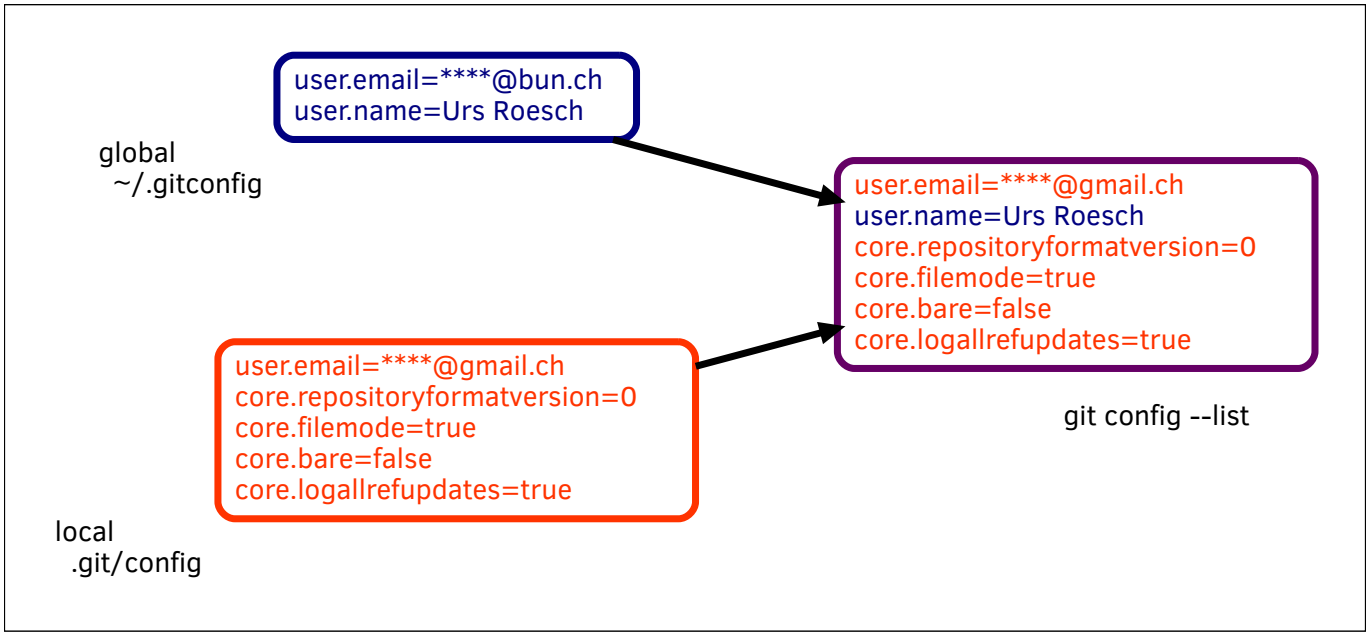

Figure 8. Combination with **`local`** *overriding* **`user.email`**.

Useful configuration options

These are basic configuration settings in git but they can make a large difference in the overall user experience.

core.editor

By default git uses the predefined system editor via the **`VISUAL`** or **`EDITOR`** environment variable. Depending on preference this may not what you like editing in.

```
git config --global core.editor emacs (1)
```

(1) Now git will use Emacs every time interactive input is required regardless of the systems editor settings.

core.pager

Git uses **`less`** as its default pager for commands that may produce multi page output. Such as **`log`**, **`diff`** or **`show`**. Depending on preference one want to use **`more`** or no pager at all.

```
git config --global core.pager '' (1)
```

(1) Switches off the pager altogheter.

color.ui

With newer versions git colors more and more of its output. To completely switch of the colored output the **`color.ui`** has to be set to **`false`**.

```
git config --global color.ui false (1)
```

(1) Switch off all colored output. The default is **`auto`** probing for terminal color support. A value of **`always`** send color to the terminal regardless of capabilities.

color.*

A more fine grained approach with color is to use the settings to only switch off certain git command's color output. Possible values for the keys listed below are **`false`** to disable, **`true`** to enable and **`always`** to enable regardless of terminal capabilities.

```
$ git config --list color.Tab Tab
color.advice
color.advice.hint
color.blame.highlightRecent
color.blame.repeatedLines
color.branch
...
```

credential.<url>

Assuming a remote repository is being used over HTTPS and the username of the repository differs from your local login on the machine there is a way to add the username to the configuration.

```
git config --local credential.https://git.bun.ch/repo/foobar.git urs.roesch (1)
```

(1) Sets the username of the URL **`https://git.bun.ch/repo/foobar.git`** to **`urs.roesch`**.

credential.helper

If git is only usable over HTTPS typing the login credentials every time becomes tiresome. There is a nifty configuration setting caching the credentials for a set period.

```
git config --global credential.helper 'cache --timeout=86400' (1)
```

(1) Caches the HTTPS credentials for 24 hours.

Summary

For brevity here are the configuration options in a tabular format.

Item	Description	Values
`list`	List the actual configuration for the repository	N/A

Item	Description	Values
`user.name`	Set the user's name used in the commit message.	String
`user.email`	Set the user's email address used in the commit message.	String
`--global`	Set the configuration for each repository of the user on the machine.	N/A
`--local`	Set the configuration only for the current repository.	N/A
`core.editor`	Change the default editor for interactive input.	String
`core.pager`	Change the pager for commands with output longer than one page.	String
`color.ui`	Change the color output for the whole of git.	false, true, always
`color.branch`	Change the color output for branching output.	false, true, always
`color.diff`	Change the color output for diff output.	false, true, always
`color.interactive`	Change the color output for interactive input.	false, true, always
`color.status`	Change the color output for status output.	false, true, always
`credential.<url>`	Add user name for a specific URL.	String
`credential.helper`	Add a credential helper for caching passwords for HTTPS URLs.	String

Module 2 - Local Repository

Goals

This is were the rubber hits the road. After all the theory about Git this is the first hands-on part. After completion of this module the student can confidently do:

- Create a new git repository.
- Create and stage files.
- Commit files.
- View the status of the repository.
- Display the log.
- Excluding files from Git.

Create a local repository

As already mentioned under location for the most part Git operates on local storage. In order to create our first repository the only thing required is the git binary. Starting with a clean slate the only other requirement is an empty directory.

The same steps performed here can be applied to an already existing project but it is better to complete module 1 before doing so.

Git commands always start with **`git`** followed by a command, in this case **`init`**, then the arguments. To get more information about a git command use **`git help <command>`**.

git init

```
$ mkdir git-repo
$ cd git-repo
$ git init (1)
Initialized empty Git repository in .../git-repo/.git/
```

(1) The command for creating or better initializing is **`git init`**. That's it, it is really that easy!

Want to know what exactly what git put into the **`.git`** directory have a look at the content of the initial tree

git add

```
$ echo "This is my first file in a git repo" > first-file.txt ❶
$ git status ❷
On branch master ❸

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        first-file.txt

nothing added to commit but untracked files present (use "git add" to track) ❹
$ git add . ❺
# git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage) ❻

        new file:   first-file.txt ❼
```

❶ Create a file with content.

❷ Execute the **`status`** command. Git has recognized the file but does not yet track it.

❸ Status' first line shows the branch name. In this case **`master`**.

❹ Git tries to inform you about possible next steps like in this case to use **`git add`**.

❺ Add all files to the staging area. The dot **`.`** recurses through all sub-directories. Only adding a single file is done with **`git add first-file.txt`**. Shell wildcards also work e.g. **`git add *txt`**.

❻ Note the message about taking a file out of cache should you have mistakenly commited to stage.

❼ Git now knows about the file **`first-file.txt`**. But remember it has not been yet commited!

Commit files

With files in staging it is time to actually put them into the repository we take now a snapshot of the repository.

git commit

```
$ git commit -m "My first git commit" ❶
[master (root-commit) 307c1a0] My first git commit ❷
 1 file changed, 1 insertion(+) ❸
 create mode 100644 first-file.txt ❹
$ git status
On branch master
nothing to commit, working tree clean ❺
```

❶ A commit message is always required. With the **`-m`** aka message argument it is passed via the command line. Otherwise an editor would appear prompting for input.

❷ The **`root-commit`** is the first commit and does not have a parent.

❸ Git is showing file and change statistics for the commit.

❹ Newly created files are displayed with the file permission.

❺ The repository is now clean, meaning there are no pending changes in the working directory or the staging area.

Inspecting log and objects

After commiting the first file one should know how to display the commit history or **`log`**. There is a multitude of options that can be used to customize the output. While showing every option is out of scope a few very useful ones are shown. Additionally the **`show`** command used for inspecting objects is briefly discussed.

git log

First up the history is inspected.

```
$ git log
commit 307c1a0a537758f3d4b6ecea98e9af2e5d0b7b88 (HEAD -> master) ❶
Author: Urs Roesch <****@bun.ch> ❷
Date:   Sun Aug 26 12:09:57 2018 +0200 ❸

    My first git commit ❹
```

With only one commit in the repository there is only a single entry shown. As the repository accumulates commits the number of **`log`** entries is also growing.

❶ The first line is the most important it shows the commit's unique identifier a hex encoded SHA1 hash. In newer version of git the log is **`decorated`**, meaning shows additional information e.g. the **`HEAD`** of the repository.

❷ The authors name. Changing this information is discussed in module 2.

❸ The date of the commit.

❹ The commit message.

To show what files are part of the commit the **`--stat`** argument is used.

```
$ git log --stat
commit 307c1a0a537758f3d4b6ecea98e9af2e5d0b7b88 (HEAD -> master)
Author: Urs Roesch <urs++git@bun.ch>
Date:   Sun Aug 26 12:09:57 2018 +0200

    My first git commit

 first-file.txt | 1 + (1)
 1 file changed, 1 insertion(+) (2)
```

(1) List of files that have been change and the number of lines added **+** or subtracted **-**.

(2) Summary of the accumulated changes. The same message shown during the commit.

While not really useful at this stage in the project with many commits one can squeeze the log output into a single line.

```
$ git log --oneline
307c1a0 (HEAD -> master) My first git commit (1)
```

(1) On one line the first 7 positions of the commit followed by the decoration and trailed by the commit message.

git show

To inspect the commmits further the **`show`** command is used. By definition if no commit is being passed **`HEAD`** is assumed.

The **`show`** command has many command line switches so the two commands discussed here are only a tiny fraction of its capabilities.

```
$ git show
commit 307c1a0a537758f3d4b6ecea98e9af2e5d0b7b88 (HEAD -> master)
Author: Urs Roesch <urs++git@bun.ch>
Date:   Sun Aug 26 12:09:57 2018 +0200

    My first git commit

diff --git a/first-file.txt b/first-file.txt (1)
new file mode 100644
index 0000000..e7e0b37
--- /dev/null
+++ b/first-file.txt
@@ -0,0 +1 @@
+This is my first file in a git repo
```

(1) The **`show`** displays the head of the commit up to the commit message the same way as **`git log`**. But then a diff command appears listing the changes commited. The assumption is made one understands the output from diff.

The **`--stat`** switch does only display statistics and skips the diff output.

```
$ git show --stat (1)
commit 307c1a0a537758f3d4b6ecea98e9af2e5d0b7b88 (HEAD -> master)
Author: Urs Roesch <urs++git@bun.ch>
Date:   Sun Aug 26 12:09:57 2018 +0200

    My first git commit

 first-file.txt | 1 +
 1 file changed, 1 insertion(+)
```

(1) Now to confuse everyone the **`git show --stat`** command displays the exact same output as the previous discussed **`git log --stat`**. This is true at this stage because there is only one commit in the repository. And while **`log`** is consulting the log **`show`** works with commit IDs passed to it. Adding one more commit and the outcome is different.

Excluding files

Depending on the project there are files that should not be included in the repository as they can be reproduced by a build script. Or temporary editor file that try to sneak into the repository.

.gitignore

The **`.gitignore`** file can be placed in any given directory within the project with the exception of the **`.git`** repository. A **`.gitignore`** file in the top-level directory of the repository applies to all files and sub-directories in the repository. Patterns from a **`.gitignore`** file further down in the tree take precedence.

```
$ mkdir tmp
$ echo temporary > tmp/tmp.txt (1)
$ git status --short
?? tmp/ (2)
echo tmp > .gitignore (3)
git status --short
?? .gitignore (4)
```

(1) Creating a dummy file in the **`tmp`** directory.

(2) Check the **`status`** of the repository in short form. The two leading question marks indicate git has found new files in the **`tmp`** directory.

(3) The **`tmp`** directory should exist but not be tracked by git. Hence a **`.gitignore`** file in the top-level with the content **`tmp`** is created.

(4) Once more running a **`status`** check shows the files in **`tmp`** are no longer considered. However the **`.gitignore`** file is not marked as untracked.

While not strictly necessary it usually makes sense to include the **`.gitignore`** file in the repository.

```
$ git add .gitignore  ❶
$ git commit -m "Excluding files with .gitignore" ❷
[master 33d3825] Excluding files with .gitignore
 1 file changed, 1 insertion(+)
 create mode 100644 .gitignore
$ git log --oneline ❸
33d3825 (HEAD -> master) Excluding files with .gitignore
307c1a0 My first git commit
```

❶ Stage the **`.gitignore`** file.

❷ Commit everything staged.

❸ Verify the with **`git log`**. Notice the **`HEAD`** has moved to the commit. And two commits are shown.

Summary

With the first module done in summary the following commands and concepts have been touched.

Item	Description	Resource
`git init`	Initialize an empty directory or existing project	man git-init
`git add`	Add file to the staging area to be included in the next commit.	man git-add
`git commit`	Send files from the staging area to the object store.	man git-commit
`git status`	Display the status of the current working directory.	man git-status
`git log`	Display the commit history of the repository.	man git-log
`git show`	Inspect various objects.	man git-show
`.gitignore`	Dot file to tell git which file / file patterns never to track.	man gitignore

Module 3 - Remote repositories

Goals

Pretty soon after using git for the first time one encounters remote repositories. This module shows the basics on how to interact with them. It is an introduction into the basic commands. To avoid conflicts each student is creating their own repository and working from it.

- Download or **`clone`** a remote repository.
- Add content to the repository.
- Learn about **`origin`**.
- Push the changes.
- Pull changes made outside of the working directory.

Create a bare repo

Creating a bare repo is an advanced topic but for the purpose of working with a local shared bare repository it is included. To distinguish bare repositories from local ones the naming convention is to suffix it with **`.git`**.

git init

```
$ git init --shared --bare remote-repo.git (1) (2)
Initialized empty shared Git repository in .../remote-repo.git/
```

(1) **`--shared`** specifies the Git repository can be shared amongst several users. Users belonging to the same group may push into that repository.

(2) The naming convention for bare repositories is to suffix it with **`.git`**.

Working with remote repository

To create a local copy of a remote or shared repository the **`clone`** command is used.

In general one would clone a remote repository over a protocol like HTTPS or ssh. A shared source tree can also reside on the local drive the syntax is only slightly different.

git clone

```
$ git clone remote-repo.git (1) (2)
Cloning into 'remote-repo'...
warning: You appear to have cloned an empty repository.
done.
```

(1) The command to create a copy locally is **`clone`**. Generally a shared repository is suffixed

with a **`.git`** extension. The cloned directory is missing said suffix.

2. To place the clone into a directory not named **`remote-repo`** append the name of the desired destination. e.g. **`git clone remote-repo.git local-repo`**.

origin

When cloning a repository per default the configuration has 4 additions not seen in a simple local repository. It is a remote URL alias as **`origin`**. While the name **`origin`** is not binding it is the default and is rarely changed. This configuration is used when pushing changes back into the shared directory.

```
$ git config --list
user.email=urs++git@bun.ch
user.name=Urs Roesch
core.repositoryformatversion=0
core.filemode=true
core.bare=false
core.logallrefupdates=true
remote.origin.url=.../remote-repo.git (1)
remote.origin.fetch=+refs/heads/*:refs/remotes/origin/* (2)
branch.master.remote=origin (3)
branch.master.merge=refs/heads/master (4)
```

1. Compared to the initial configuration in the local repository there is new the **`remote.origin.url`** values pointing to the shared repository's URL. This can be a **`file`** resource like in this case, **`http`**, **`ssh`** or **`git`**.
2. Configures the mapping between the local and remote repository when **`git fetch`** is issued.
3. Sets the branch master for remote in this case it is called **`origin`**.
4. Sets the merge branch when pulling from a repository.

To put this into perspective once more a peak into the **`.git`** repository helps. The tow important sub-directories are colored the rest is grayed out.

For the remote repository a new directories in **`logs`** and **`refs`** have been created. They correspond to the names already seen in the list of configuration items above. As before git is keeping everything local a remote branch is only synced or updated from or to the shared repository when the network commands such as **`push`**, **`fetch`** or **`pull`** are issued.

A clear understanding of how this mechanism works is key to comprehend how git works with shared repositories over the network.

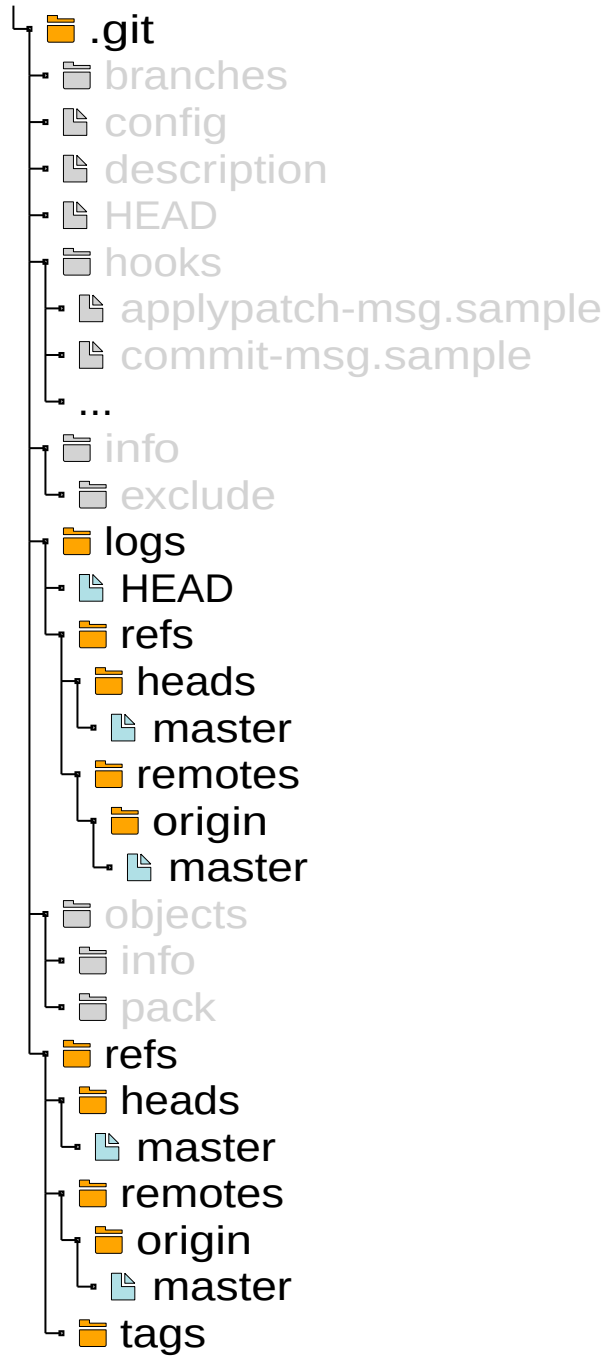

Figure 9. Remote branches in a .git directory highlighted in color.

git push

With the repository in place let's populate it with content and send or the proper term for git is **push** the changes back into the remote repository.

First off a commit must be in place.

```
$ echo 'First remote repository content!' > first_file.txt (1)
$ git add first_file.txt (2)
$ git commit -m 'Initial commit' (3)
[master (root-commit) 45d5290] Initial commit
 1 file changed, 1 insertion(+)
 create mode 100644 first_file.txt
$ git log --oneline
45d5290 (HEAD -> master) Initial commit
```

(1) Create a new file called **`first_file.txt`**.

(2) Add the file to the staging area.

(3) Commit the file to repository. Note at this stage nothing has been submitted to the remote repository.

This is the same as for the local repository. As all the actions are done locally there is no difference if you work with a remote repository.

With the commit in place the changes have to be pushed to the remote repository. This is accomplished with **`git push`**.

```
$ git push
Counting objects: 3, done. (1)
Writing objects: 100% (3/3), 250 bytes | 250.00 KiB/s, done. (2)
Total 3 (delta 0), reused 0 (delta 0)
To .../remote-repo.git (3)
 * [new branch]      master -> master (4)
```

(1) How come there are 3 objects for one commit? Actually there is the commit object, the tree object and the blob one.

(2) The actual network operation of sending the data to the shared repository.

(3) Where did it send the objects to this is by default **`origin`**.

(4) Actions done are displayed in square brackets in this case **`[new branch]`**. Also which branches are involved per default this is **`master`** for a new repository.

git fetch

Fetching is as the name states going to the shared repository and getting the data and putting it into the local git repository.

To properly show the effect how this is works a second commit outside of the current working repository has to be created. The below action is simulating a fellow git user with working on the same repository.

```
$ cd ..
$ git clone remote-repo.git remote-repo2
Cloning into 'remote-repo2'...
done.
$ cd remote-repo2/
$ git log --oneline
45d5290 (HEAD -> master) Initial commit
$ echo "Second file" > second_file.txt
$ git add second_file.txt
$ git commit -m "Second commit"
[master 885f6c7] Second commit
 1 file changed, 1 insertion(+)
 create mode 100644 second_file.txt
$ git push
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 290 bytes | 290.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To .../remote-repo.git
   45d5290..885f6c7  master -> master
$ cd ../remote-repo
```

After simulating the fellow git user and being back in the repo where the first commit was done one can issue the **`git fetch`** command to get the changes from the shared repository.

```
$ git fetch
remote: Counting objects: 3, done. (1)
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From .../remote-repo
   45d5290..885f6c7  master     -> origin/master (2)
```

(1) Fetch is downlowding the missing objects from remote into the object store.

(2) The remote branch **`master`** is put into **`origin/master`**.

```
$ ls
first_file.txt (1)
$ git log --oneline
45d5290 (HEAD -> master) Initial commit (2)
```

(1) At this point **`second_file.txt`** is not appearing in the current branch **`master`**.

(2) If the repository has been fetched why does the master branch still show the initial commit? Because fetch downloads all the objects from the shared repository and updated the **`HEAD`** for **`origin/master`** but does not merge into the local repository. This is best shown with an example.

```
$ git log origin/master ❶
commit 885f6c751abb430cb48c0903fcd82a2d35a77d25 (origin/master) ❷
Author: Urs Roesch <urs++git@bun.ch>
Date:   Thu Sep 6 06:57:50 2018 +0200

    Second commit

commit 45d52900b73a5bd461cbaef2652b9d1ed8220b3b (HEAD -> master) ❸
Author: Urs Roesch <urs++git@bun.ch>
Date:   Thu Sep 6 06:37:46 2018 +0200

    Initial commit
```

❶ Instead of simply showing the log of the current branch **`git log`** is instructed to display from branch **`origin/master`**.

❷ With a decorated log one can see the head of **`origin/master`** is one commit ahead of the current branch.

❸ Current branch HEAD points to the initial commit.

git merge

To bring the changed from the remote master branch into the working directory the **`merge`** command is used.

```
$ git merge origin/master ❶
Updating 45d5290..885f6c7
Fast-forward
 second_file.txt | 1 + ❷
 1 file changed, 1 insertion(+)
 create mode 100644 second_file.txt
$ git log
commit 885f6c751abb430cb48c0903fcd82a2d35a77d25 (HEAD -> master, origin/master) ❸
Author: Urs Roesch <urs++git@bun.ch>
Date:   Thu Sep 6 06:57:50 2018 +0200

    Second commit

commit 45d52900b73a5bd461cbaef2652b9d1ed8220b3b
Author: Urs Roesch <urs++git@bun.ch>
Date:   Thu Sep 6 06:37:46 2018 +0200

    Initial commit
```

❶ The **`merge`** command expects a source where to merge from. In this case the remote branch **`origin/master`** is used.

❷ The messages displayed during the merge are resembling the ones from the commit.

❸ Now **`HEAD`** and **`origin/master`** are in sync.

git pull

It seems an awful lot of typing for fetching and merging from remote repositories. But there is a better way one can achieve the same with the **`push`** command.

To show this both **`HEAD`** and **`origin/master`** have been reset.

```
$ git log origin/master
commit 45d52900b73a5bd461cbaef2652b9d1ed8220b3b (HEAD -> master, origin/master)
Author: Urs Roesch <urs++git@bun.ch>
Date:   Thu Sep 6 06:37:46 2018 +0200

    Initial commit
$ git pull
remote: Counting objects: 3, done. (1)
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From .../remote-repo
   45d5290..885f6c7  master     -> origin/master
Updating 45d5290..885f6c7
Fast-forward (2)
 second_file.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 second_file.txt
```

(1) Section one is the same as with **`git fetch`**.

(2) The second part is verbatim to **`git merge`**.

Summary

Finally the summary of the module 3 for remote repositories with the commands in overview.

Item	Description	Resource
`git clone`	Initialize an empty directory or existing project	man git-clone
`origin`	Sets the default location of the remote repository	N/A
`git push`	Send local changes to the shared repository.	man git-push
`git fetch`	Fetch objects from a shared repository.	man git-fetch
`git log <repo>`	Display logs from other repos without first checking it out	man git-log
`git merge`	Merge the difference between two branches.	man git-merge
`git pull`	Fetch and then merge changes into the current working branch.	man git-pull

Module 4 - Visualizing and tracking changes

Goals

This module provides a deep dive into how to work show changes and differences between certain commits or isolating files to changes. Further a view into who did change what will be given.

- Use of the **`diff`** command.
- Use **`blame`** to find who's done what change.
- Find strings in the tree with **`grep`**.

Display changed content

When working with a VCS one of the most important daily tasks is to evaluate changes made during the course of time. Git provides a variety of tools to get the job done.

git diff

Diff's list of options and switches is very lengthy so this part is only covering a tiny portions of the possibilities at hand.

Executing **`git diff`** without any other parameters will show only changes between tracked but yet unstaged files. So first a change needs to be made to an already tracked file.

```
$ echo 'second line in first file' >> first_file.txt (1)
$ cat first_file.txt (2)
First remote repository content!
second line in first file
$ git diff (3)
diff --git a/first_file.txt b/first_file.txt
index 697ffc4..f79ba9f 100644
--- a/first_file.txt
+++ b/first_file.txt
@@ -1 +1,2 @@
 First remote repository content!
+second line in first file
```

(1) Adding a new line to the tracked file **`first_file.txt`** by appending it on the command line. Alternative one can edit the file interactively with a text editor.

(2) Confirming the content has been added with **`cat`**.

(3) Invoking **`git diff`** shows the changes in diff format. To explain the format at length is not in scope of this tutorial. The important syntax is that a **`+`** in front of a line means and addition and a **`-`** is a deletion.

As mentioned basic invocation of **`git diff`** only tracks changes in the working tree. To see changes of staged files more options are required.

```
$ git add first_file.txt ❶
$ git diff ❷
$ git diff --cached ❸
diff --git a/first_file.txt b/first_file.txt
index 697ffc4..f79ba9f 100644
--- a/first_file.txt
+++ b/first_file.txt
@@ -1 +1,2 @@
 First remote repository content!
+second line in first file
```

❶ Adding the previously changed file to the stating area.

❷ Invoking the **`git diff`** shows no further changes are in the working directory.

❸ To show the changes in staging the **`--cached`** option is passed. The result from diff is now identical as in the previous example.

Spinning this thread a bit further showing differences in a already commited file requires a reference to a commit.

```
$ git commit -m "Second line in first file" ❶
[master 3fb25c5] Second line in first file
 1 file changed, 1 insertion(+)
$
urs@automatix:~/var/work/git-tutorial/remote-repo$ git diff HEAD~1 ❷
diff --git a/first_file.txt b/first_file.txt
index 697ffc4..f79ba9f 100644
--- a/first_file.txt
+++ b/first_file.txt
@@ -1 +1,2 @@
 First remote repository content!
+second line in first file
```

❶ Committing the files in stage.

❷ Invoking **`git diff`** using the **`HEAD~1`** meaning show a diff between the **`HEAD`** and commit prior to **`HEAD`**. This is nearly identical to **`git show`** with the exception that the commit message is not shown.

To show only the summary of changes to the file one can use the **`--stat`** parameter already shown in previous commands such as **`log`** or **`show`**.

```
$ git diff --stat HEAD~2 ❶
 first_file.txt  | 1 +
 second_file.txt | 1 +
 2 files changed, 2 insertions(+)
```

❶ Again the diff command is set to work on **`HEAD`** and two commits prior to it. Stat is showing the file names and changes including a summary at the end. Compared to **`log`** or **`show`** however no further information is provided.

Where diff really shines tho is showing changes to a single file only.

```
$ git diff HEAD~2 second_file.txt (1)
diff --git a/second_file.txt b/second_file.txt
new file mode 100644
index 0000000..20d5b67
--- /dev/null
+++ b/second_file.txt
@@ -0,0 +1 @@
+Second file
```

(1) As before the last 2 commits are being considered but additionally at the end a file name is provided. This limits the diff to only show changes in the last two commits for **`second_file.txt`**.

Find who changed what

Sometimes it is important to know who changed what. For example when questions arise why a change was commited or why something was implemented a certain way. And git has a tool ready for that too.

git blame

While the name **`blame`** suggests an exercise in finger pointing it serves a much more useful purpose than just pinning the blame on someone. The command works on a single file to show which changes have been made at which revision.

While most of the git commands do not require additional parameters **`git blame`** requires a file name to work with.

```
$ git blame first-file.txt
^8070030 (Urs Roesch        2018-10-06 13:06:01 +0000  1) First file (1)
2e97d380 (Linux Torvalds    2019-12-25 08:07:28 +0000  2) Second line (2)
e5149955 (Junio C Hamano    2020-01-04 11:08:45 +0000  3) Third line (3)
e5149955 (Junio C Hamano    2020-01-04 11:08:45 +0000  4) Fourth line
e5149955 (Junio C Hamano    2020-01-04 11:08:45 +0000  5) Fifth line
00000000 (Not Committed Yet 2020-10-03 09:10:49 +0000  6) (4)
00000000 (Not Committed Yet 2020-10-03 09:10:49 +0000  7)
00000000 (Not Committed Yet 2020-10-03 09:10:49 +0000  8)
00000000 (Not Committed Yet 2020-10-03 09:10:49 +0000  9)
00000000 (Not Committed Yet 2020-10-03 09:10:49 +0000 10) Apologies, I lost count
```

(1) The first line in this example is also the initial commit of the repository and demarcate the boundary of the **`blame`**. Boundaries prefix the SHA1 a caret symbol **`^`**. For brevity the SHA1 hashes are cut at character 8. With option **`-l`** the whole hash is shown. The tabular output format provides the following fields: **`<revision hash> (<commiter name> <timestamp> <line number>) content`**

(2) Line two is a revision by a certain **`Linux Torvalds`** adding a single line to the file.

3. Line 3 is also by a different committer called **`Junio C Hamano`**, this time the change adds 3 lines.
4. Depicts a change not yet commited. Of note here is the SHA1 with all zeros and the committer name of **`Not Commited Yet`** the time stamp is the time at command execution.

As with any git command there is a slew of additional options a few of the more useful ones examined in the next few examples.

When only wanting to review changes starting at a particular revision the SHA1 can be provided followed by two dots.

```
$ git blame 2e97d380.. -- first-file.txt
^2e97d38 (Linux Torvalds    2019-12-25 08:07:28 +0000  1) First file (1)
^2e97d38 (Linux Torvalds    2019-12-25 08:07:28 +0000  2) Second line
e5149955 (Junio C Hamano    2020-01-04 11:08:45 +0000  3) Third line (2)
e5149955 (Junio C Hamano    2020-01-04 11:08:45 +0000  4) Fourth line
e5149955 (Junio C Hamano    2020-01-04 11:08:45 +0000  5) Fifth line (3)
```

1. To note here is the first line is folded into the staring commit **`2e97d380`** although in the previous example it was an independent commit. The caret symbol **`^`** demarcate the boundary of such folded commits.
2. In this example the only change made is the addition of the 3 lines by committer **`Junio C Hamano`**.
3. The keen observer the question is where are the lines not yet commited. When inspecting changes between specific commits unstaged changes are not factored in.

While the previous example shows changes between a commit and the current **`HEAD`** the same can be done between two arbitrary revisions.

```
$ git blame 80700303..2e97d380 -- first-file.txt
^8070030 (Urs Roesch        2018-10-06 13:06:01 +0000  1) First file (1)
2e97d380 (Linux Torvalds    2019-12-25 08:07:28 +0000  2) Second line (2)
```

1. The first line here is the fist commit given as an option on the command line.
2. Line two is the revision provided as the end point for the comparison.

There is also the option to view all changes up to certain revision. This is achieved by two dots followed by the revision hash.

```
$ git blame ..2e97d380 -- first-file.txt
^8070030 (Urs Roesch        2018-10-06 13:06:01 +0000  1) First file (1)
2e97d380 (Linux Torvalds    2019-12-25 08:07:28 +0000  2) Second line (2)
```

1. While the output is identical to the previous example the boundary for comparison is now the initial commit.
2. Line two is the revision provided as the end point for the comparision.

Instead of the fickle SHA1 hashes one can use tags e.g. **`git blame v2.2.0..HEAD -- first-file.txt`** or the switch **`--since`** e.g. **`git blame --since=3.weeks -- first-file.txt`**

Another options is for limiting the comparision to a range of lines in the file with the switch **`-L`**.

```
$ git blame -L 2,4 first-file.txt
2e97d380 (Linux Torvalds    2019-12-25 08:07:28 +0000  2) Second line (1)
e5149955 (Junio C Hamano    2020-01-04 11:08:45 +0000  3) Third line
e5149955 (Junio C Hamano    2020-01-04 11:08:45 +0000  4) Fourth line (2)
```

(1) The comparison starts at line 2 and since the revision boundary is outside of the limit no caret symbol is shown. Of note is also the line numbers are absolute.

(2) Although the commit **`e5149955`** spans 3 lines only the range defined on the command line is displayed.

Locate patterns in files

Locating certain text patterns in a large repository can be daunting. But **`git grep`** provides most of the options familiar from the Unix command **`grep`** tailored to work under the specifics of a repository.

git grep

The basic usage is nearly identical to the Unix **`grep`** command. Simply prefixing it with **`git`**. For brevity there is only a single file in the repository called **`first-file.txt`**.

```
$ cat first-file.txt
First file
Second line
Third line
Fourth line
Fifth line

$ git grep First
first-file.txt:First file (1)
```

(1) Like with **`grep`** operating on multiple files the result is prefixed with the file name the match was located.

There is a way to limit the search to files matching a name pattern, not unlike normal **`grep`**. However the syntax is slightly different as wildcards are required to be properly escaped.

```
$ cp first-file.txt first-file.copy (1)
$ git add first-file.copy (2)

$ git grep First
first-file.txt:First file
first-file.copy:First file (3)

$ git grep First -- '*.txt' (4)
first-file.txt:First file (5)
```

(1) Creating a copy of the existing text file with the extension **`.copy`**

(2) Adding the newly created file to the staging area.

(3) Executing the same command two result show up. Both commited changes and the ones in the staging area.

(4) To limit the search to only files ending in **`.txt`** the shell escaped wildcards is used **`'*.txt'`**.

(5) The result then only shows file matching the wildcard.

There is a way to include untracked files in the search by adding the **`--untracked`** switch.

```
$ echo "Second file" > second-file.txt (1)
$ git grep --untracked Second (2)
second-file.txt:Second file (3)
```

(1) Creating a new yet to be tracked file with content.

(2) Adding the **`--untracked`** switch to the **`git grep`** query includes all file under the git tree.

(3) Result now also includes the untracked files. There is however no visible queue for untracked files.

Descending into a sub directory within the git tree limits the search to the files under said sub directory.

Summary

With all the basic research tools under our belt here the summary of the commands.

Item	Description	Resource
`git diff`	Find differences between staged changes, commits or single files.	man git-diff
`git blame`	Visualize who has changed which line in a commited file.	man git-blame
`git grep`	Search text strings in a files recursively.	man git-grep

Module 5 - Tags

Goals

An integral part of almost all VCS' is the feature of creating tags for release or milestones. Under git to navigate the cryptic SHA1 messages can be quite daunting. Marking milestones, and releases in a more human readable format can be archived with tags.

In git 3 types of tags exists **`lightweight`**, **`annotated`** and **`signed`**. This module is looking mainly at lightweight and annotated ones. Signed tags are an advanced topic which requires the use of GPG-keys which is considered out of scope for this tutorial.

- List tags with **`tag`**.
- Create tags with **`tag <name>`**.
- Remove tags with **`tag -d`**.
- Push tags to remote repositories using **`push --tags`**
- Delete tags in remote repositories.

List tags

Tags have their own git command aptly named **`tag`**. Issuing **`tag`** without any options list all defined tags. The list of options for **`tag`** at least for lightweight and annotated tags is comparatively small.

git tag

To show the current branch the command **`branch`** without any options or switches is invoked.

```
$ git tag (1)
v1.0.0 (2)
v1.1.0
```

(1) One can also append the switch **`-l`** or **`--list`** but the output stays the same.

(2) Tags are listed without any further information such as to which revision it points. The name of the tags are arbitrary alpha numerical characters. But for version releases a lowercase **`v`** followed by the version number is a widely followed convention.

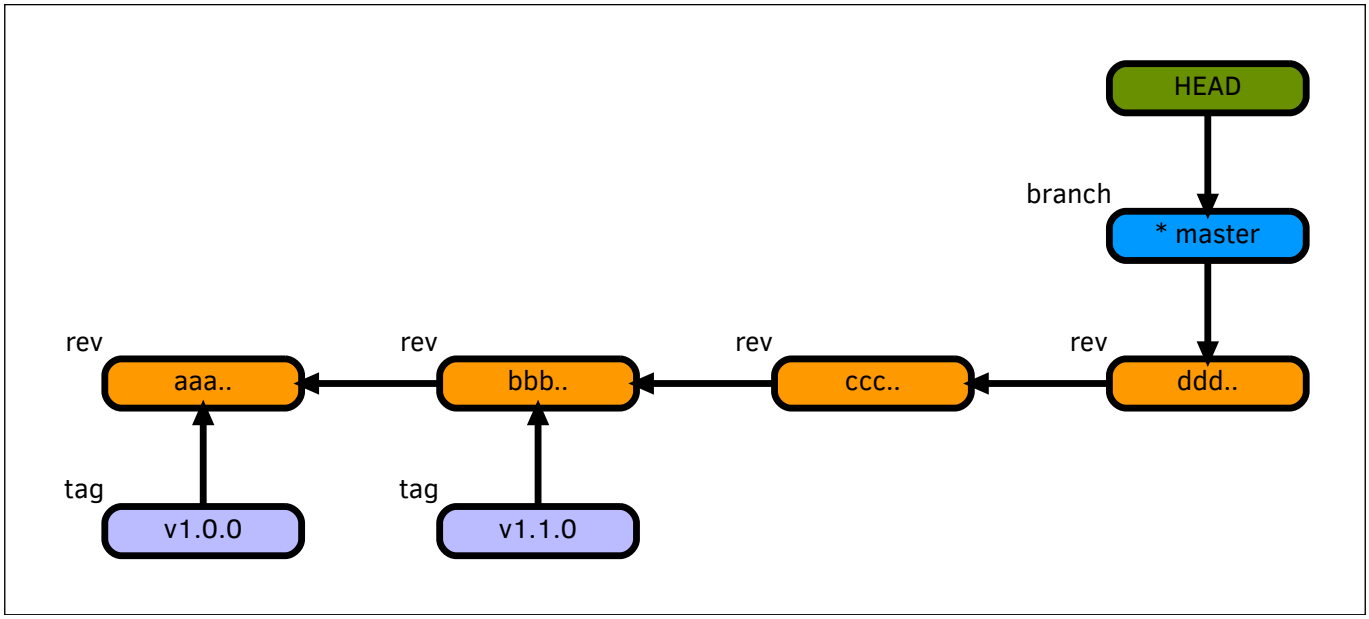

Figure 10. Example visualization of revisions, tags and branches in git.

Create tags

Creating tags for the current revision is very straight forward. The only additional option is to provide a name for the tag.

When creating tags for older revisions the first few character of the SHA1 commit have to be appended.

For annotated tags the option **`-m`** followed by a message is added.

git tag (create)

By default tags are created to point to the **`HEAD`**. The only additional option required is the tag's name.

```
$ git tag v1.3.0 (1)
```

(1) Creates tag with name **`v1.3.0`** to the current revision (**`HEAD`**) the same.

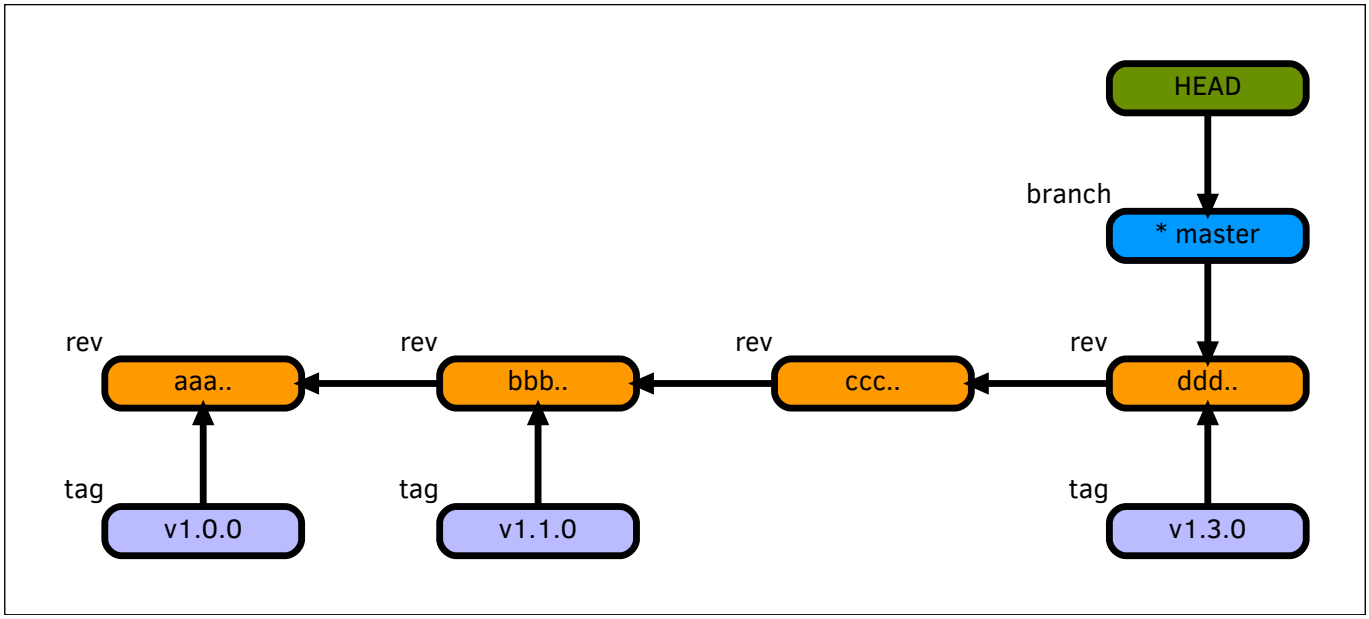

Figure 11. Git tree after adding the new new tag **`v1.3.0`** *to* **HEAD**.

git tag (create → revision)

By default tags are created to point to the **HEAD**. The only additional option required is the tag's name. This way a tag can be added to a revision earlier in the tree for example if the tagging was forgotten.

```
$ git tag v1.2.0 64b67952 (1)
```

(1) Creates tag with name **`v1.2.0`** at revision **`64b67952`**.

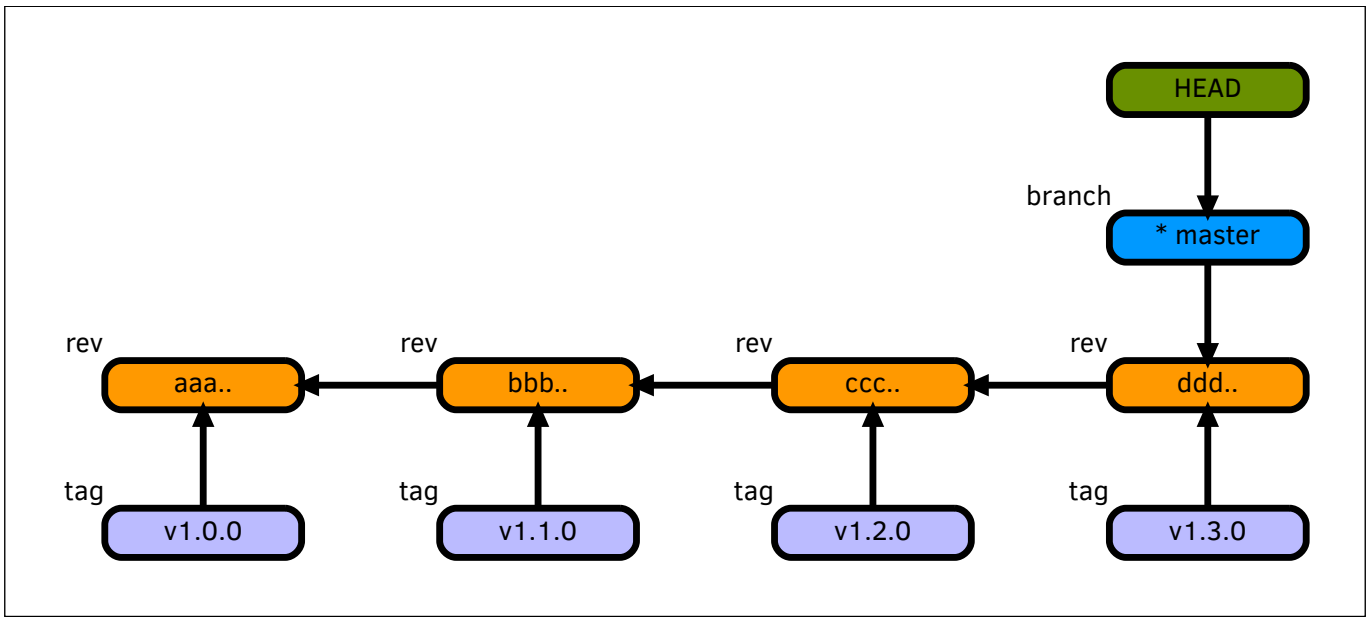

Figure 12. Tag **`1.2.0`** *was added to a revision prior to* **HEAD**.

git tag (annotated)

By default tags are created to point to the **HEAD**. The only additional option required is the tag's name. This way a tag can be added to a revision earlier in the tree for example if the tagging was forgotten.

```
$ git tag -a milestone_1 -m "This is the first milestone" (1)
$ git show milestone_1 (2)
tag milestone_1 (3)
Tagger: Urs Roesch <Urs Roesch>
Date:   Sun Nov 8 17:17:17 2020 +0100

This is our first milestone (4)

commit 45a6088dd900e5363dddbb656360661adb94c1a1 (HEAD -> production, tag:
milestone_1) (5)
Author: Urs Roesch <Urs Roesch>
Date:   Sun Nov 8 10:47:36 2020 +0100

    first-file: New milestone reached
```

(1) Creates an annotated tag named **`milestone_1`** with a message.

(2) With the **`show`** command and argument **`milestone_1`** the tag plus the referred commit is shown.

(3) Displays the content of the tag object.

(4) Message provided during tag creation.

(5) The commit referred to by the tag.

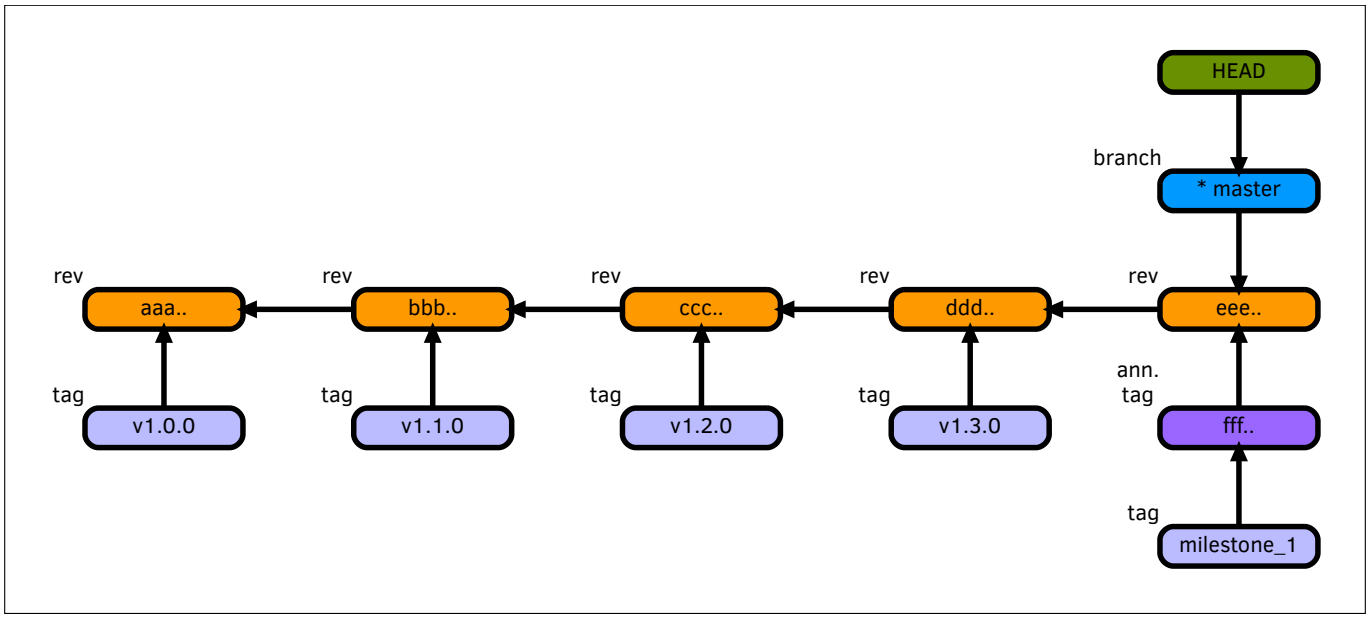

Figure 13. Annotated tags point to a tag object which points to the revision.

Push tags

When pushing changes to a remote repository tags are not being transmitted. Some automated workflows rely heavily on tags. In this rather short module the tat

git push

To push only tags to a remote repository the **`push`** command takes the switch **`--tags`** and will then push to the default remote repository usually **`origin`**.

```
$ git push --tags (1)
Enumerating objects: 9, done. (2)
Counting objects: 100% (9/9), done.
Delta compression using up to 8 threads
Compressing objects: 100% (5/5), done.
Writing objects: 100% (7/7), 748 bytes | 748.00 KiB/s, done.
Total 7 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/octocatterpillar/git-tutorial.git
 * [new tag]         milestone_1 -> milestone_1 (3)
```

(1) Issuing **`git push --tags`** pushes only tags to the remote repository.

(2) Standard push message.

(3) Pushed tags are listed one per line in a tabular format.

There is an options for **`push`** called **`--follow-tags`** which can be provided during a normal push. Said option will only push annotated tags!

Delete tags

Deleting tags is as easy as creating them. Simply provide the **`-d`** switch before the tag name.

There is deletion of a lightweight tag and an annotated tag. Although they look exactly the same on surface the difference is shown in the figure to each command.

Tags deleted locally which have already been pushed to a remote repository will be downloaded again with the next **`fetch`**.

git tag (delete)

Simply using the **`-d`** minus switch one can delete a tag. A deleted tag can easily be recreated as it simply a pointer to a certain revision.

```
$ git tag -d v1.1.0 (1)
```

(1) The **`-d`** switch will remove said tag.

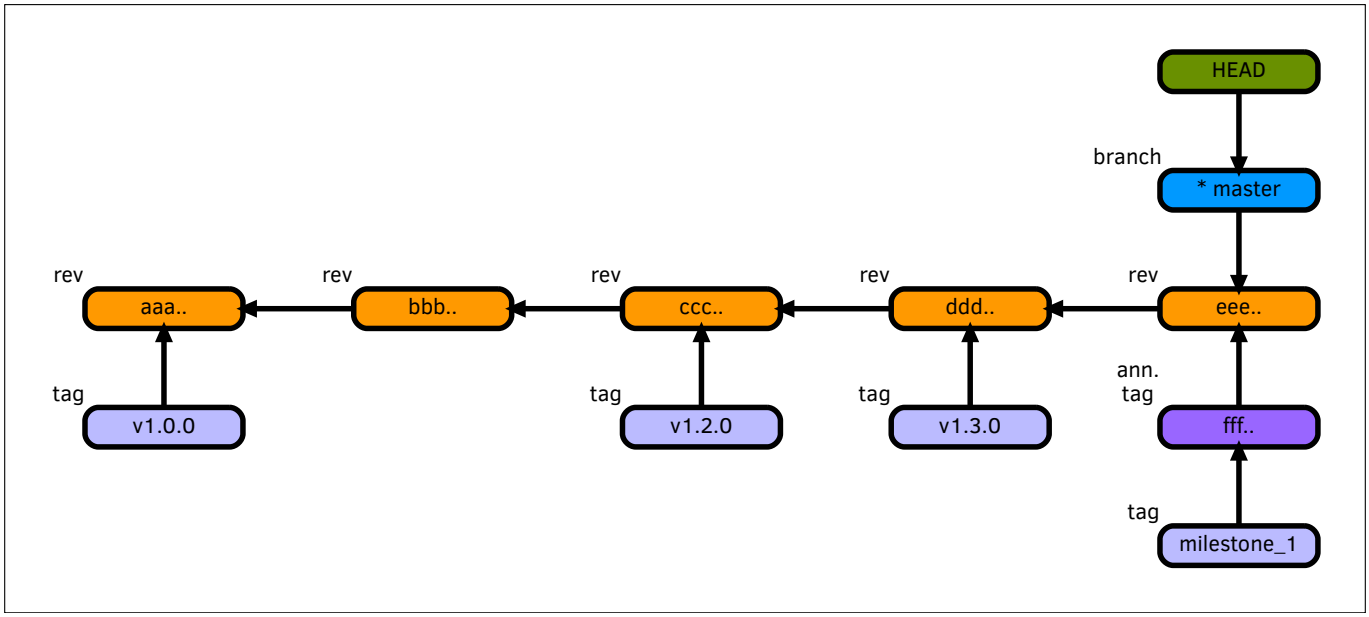

Figure 14. The git tree after removing tag **`v1.1.0`**.

git tag (delete → annotated)

Deleting a annotated tag does in no way differ from the the command used for the lightweight tag. Internally the reference file to the revision is removed for the lightweight tag. For an annotated tag the reference file to the tag object is remove only. The tag object is made an orphan.

```
$ git tag -d v1.1.0 (1)
```

(1) The same syntax applies for both lightweight and annotated tags.

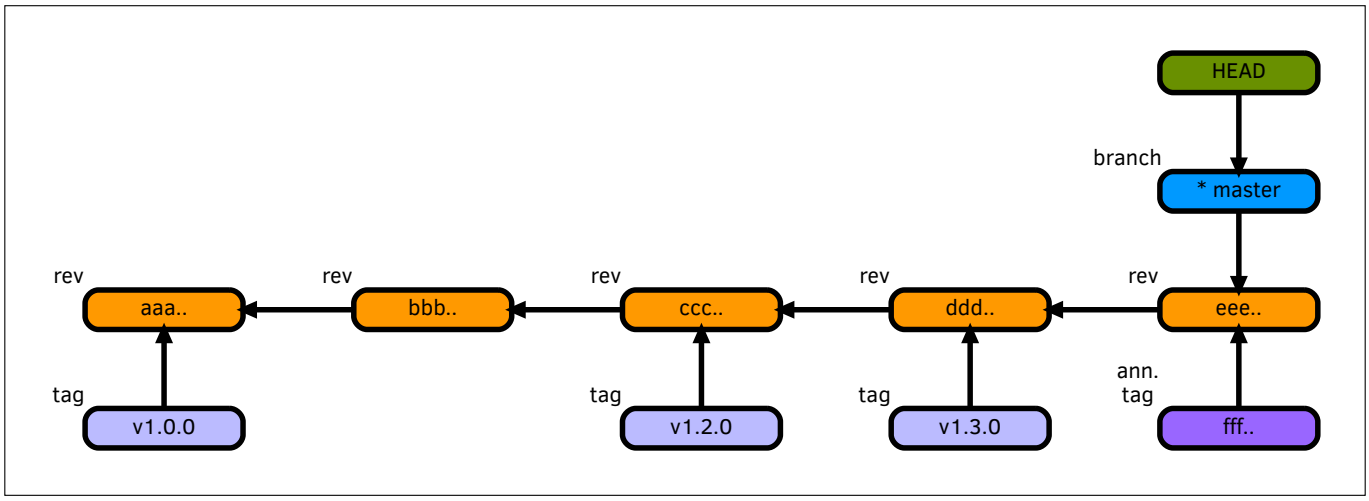

Figure 15. The tag has vanished but the tag object remains.

Summary

Tags are a big help in locating important milestones and releases and easy navigating a repository without constant lookup of hashes.

Item	Description	Resource
`git tag`	Create and manage tags.	man git-branch
`git push`	Push tag changes to a remote repository.	man git-push

Module 6 - Branches

Goals

One of the defining concepts of git from the outset was to provide cheap and fast branching of a repository. How to branch a repository and how rebase with changes that occur outside of the branch.

- Show branches in repository with **`branch`**.
- Create new branches with **`checkout -b`**.
- Delete branches with **`branch -d`** or **`branch -D`**
- Use **`rebase`** to keep branches up to date.
- Rename branches with **`branch -m`**.
- Solve merge conflicts.

Show branches

Per default with **`git init`** a branch named **`master`** is created. The **`master`** branch is just a convenience name pointing to a SHA1 revision.

The master branch can be renamed or deleted as it does not have any relevance to git. It is just an alias.

git branch

To show the current branch the command **`branch`** without any options or switches is invoked.

```
$ git branch
* master (1)
```

(1) The currently active branch is prefixed with an asterisk **`*`**. As there is only one branch present in the repository this is not instantly visible. As the module progresses the difference becomes more obvious.

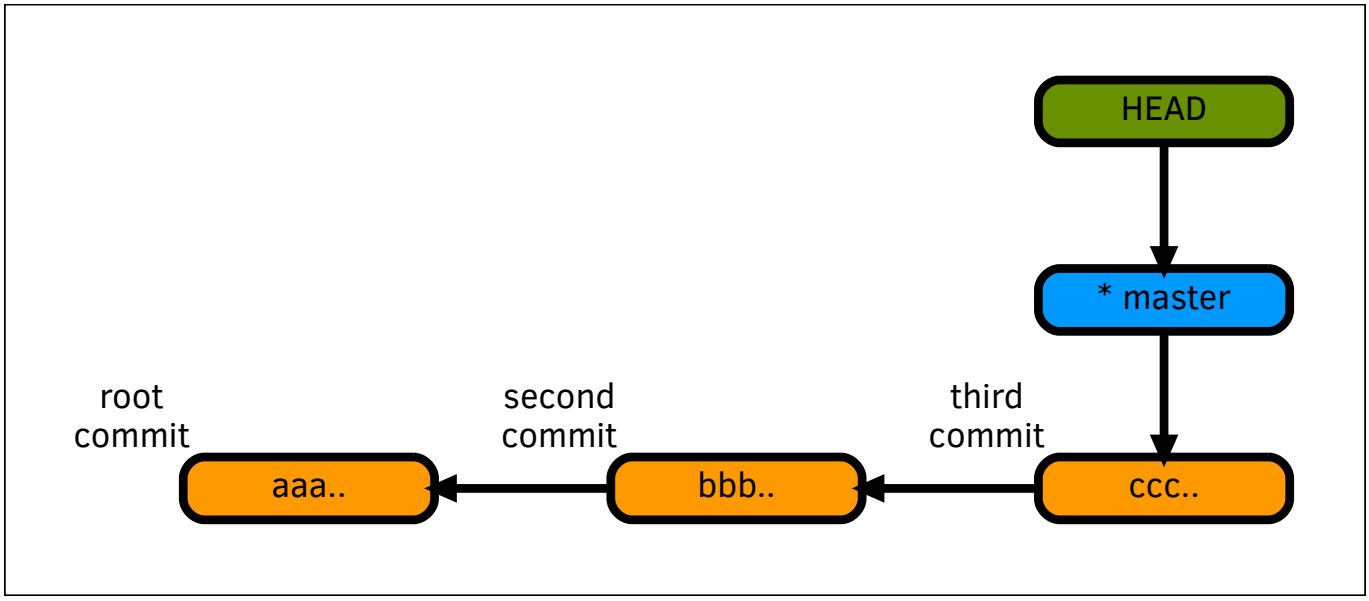

Figure 16. Repository with 3 commits and branch **`master`** *pointing to revision* **`ccc..`**.

Create a new branch

One of the stated goals of the git creator Linus Torvalds was to make branching as **cheap** and **easy** as possible. When working with large repositories and many contributors branching is a necessity. But with a good understanding and a bit of practice it quickly become second nature.

git branch

To create a new branch one the **`branch`** is used followed by the name of the new branch.

```
$ git branch testing (1)
$ git branch
* master (2)
  testing
$ git checkout testing (3)
Switch to branch 'testing'
$ git branch
  master
* testing (4)
```

(1) During branch creation the **`HEAD`** of currently active branch is used as initial pointer.

(2) **`master`** is still the current branch. The previous command only created the new branch but did not switch to it.

(3) Using the checkout command we move the **`HEAD`** to branch **`testing`**

(4) Now **`testing`** is the current working branch visible by the prefixed asterisk **`*`**.

Branch names can contain the Unix directory separator **`/`** for grouping changes e.g. **`bugfix/ticket-123`** or **`release/v1.2.3`**.

There is a shortcut for creating and switching to the newly minted branch all at once. The command **`git checkout -b <branchname>`** will achieve the same as **`git branch <branchname>`** followed by **`git checkout <branchname>`**.

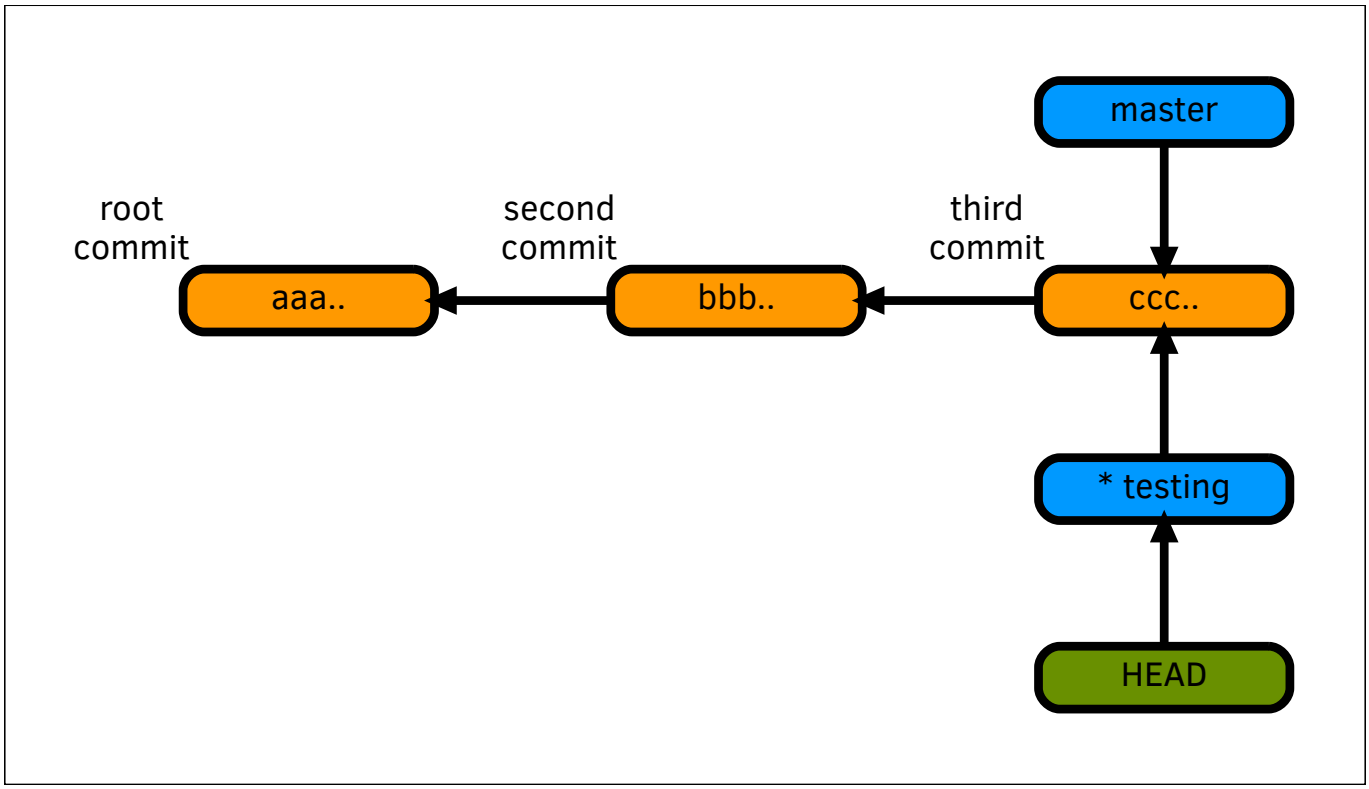

Figure 17. After creation of branch **`master`** *and* **`testing`** *pointing to the same revision.*

git add & commit

When adding a new commit under branch **`testing`** the pointer moves to the latest revision.

```
$ echo 'Branched file' > branched-file.txt (1)
$ git add branched-file.txt
$ git commit -a -m  "First file under branch testing" (2)
[testing 4b9b86d] First file under branch testing
 1 file changed, 1 insertion(+)
 create mode 100644 branched-file.txt
```

(1) Working under branch **`testing`** a new file is created.

(2) File is added and commited. Noteworthy, there is no visual hint from the commit command that this is branch testing.

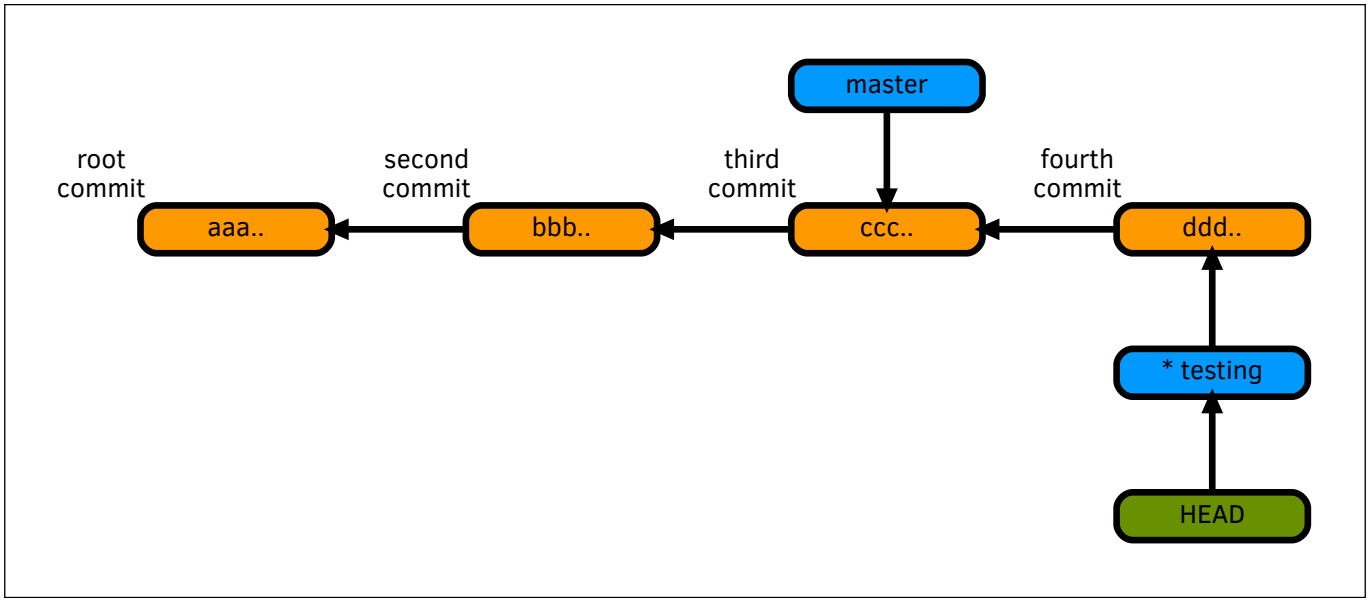

Figure 18. With the new commit the **`testing`** *pointer moved to the new commit.*

Working with multiple branches

For this section the assumption is made that that there is a bug in the current **`master`** branch that has not been addressed under **`testing`** as it is used to develop new features. To fix the bug a new branch is created called **`bugfix`** starting out with the same revision as **`master`**.

git checkout

To move between branches once more the **`checkout`** command followed by the branch name is issued.

```
$ git checkout master
Switched to branch 'master'
$ git branch
* master (1)
  testing
```

(1) **`master`** is now again the current branch being prefixed with the asterisk.

Since **`git`** version 2.23 the **`switch`** command can be used instead of **`checkout`**. To create a new branch **`git switch --create <branch name>`** is used. The short option for **`--create`** is **`-c`**. With **`git switch -`** one can toggle between current and last used branch. This is not unlike **`cd -`** under the Bash shell.

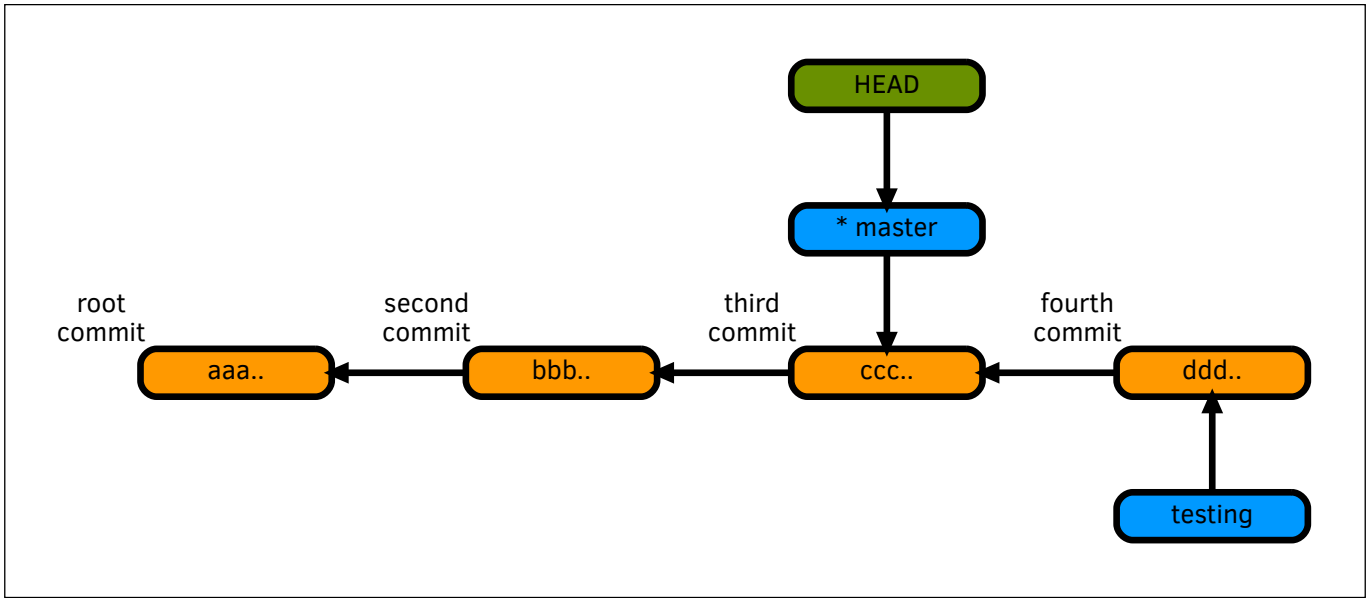

Figure 19. Moving back to **`master`** *the* **`HEAD`** *is now again at commit* **`ccc`**…

git checkout -b

With the **`HEAD`** back on commit **`ccc..`** the new branch **`bugfix`** is created.

```
$ git checkout -b bugfix ❶
Switched to a new branch 'bugfix'
$ git branch
* bugfix ❷
  master
  testing
```

❶ Create a new branch called **`bugfix`**

❷ Branch **`bugfix`** is now the **`HEAD`**.

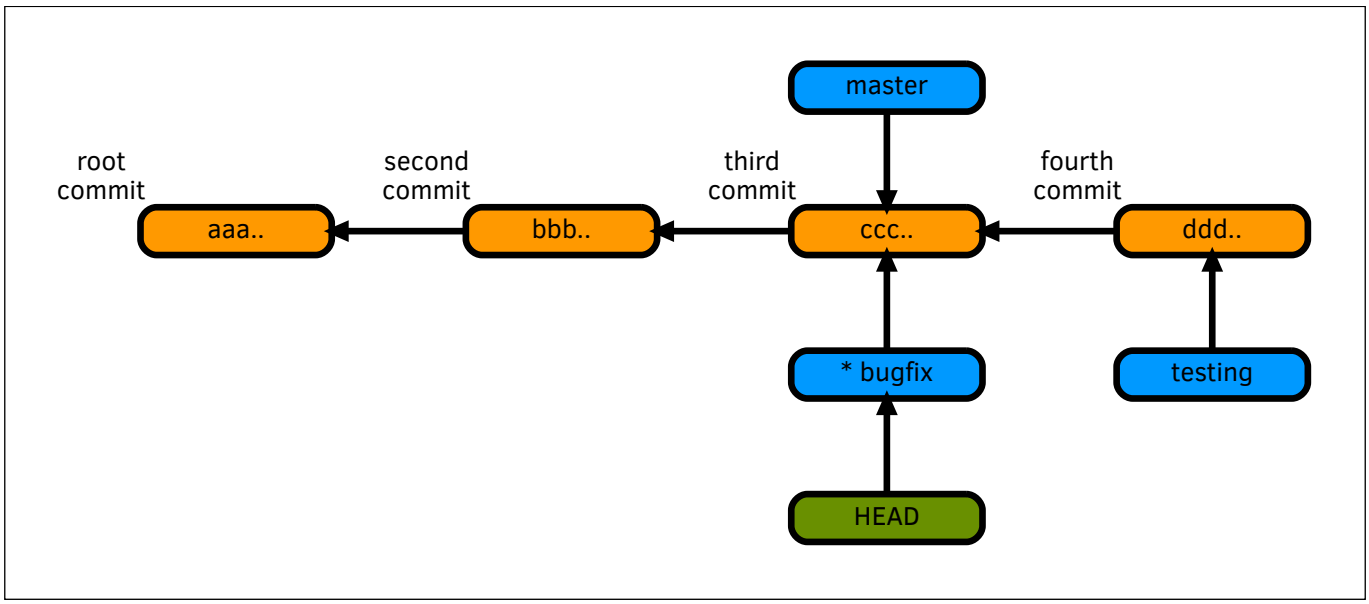

Figure 20. After checking out **`bugfix`** *it points to the same revision as* **`master`**.

git commit

With branch **`bugfix`** created and checked out the fix is developed and commited.

```
$ git diff
diff --git a/first-file.txt b/first-file.txt
index 4c5fd91..aa24abd 100644
--- a/first-file.txt
+++ b/first-file.txt
@@ -1 +1 @@
-First file
+First file with bugfix (1)
$ git commit -a -m "Bugfix for first file"
[bugfix a27a927] Bugfix for first file
 1 file changed, 1 insertion(+), 1 deletion(-)
```

(1) The first line in **`first-file.txt`** has been modified adding **`with bugfix`** to the first line.

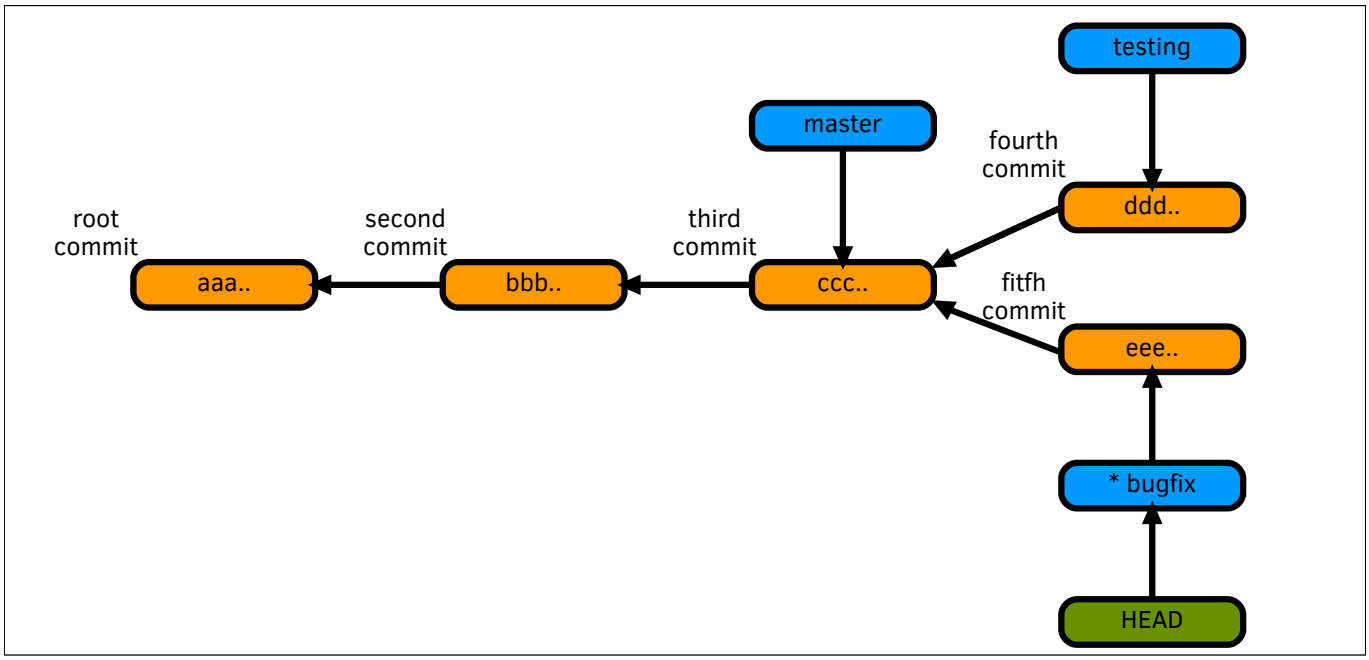

Figure 21. With the new commit to **`bugfix`** *the branches start diverging.*

Merging branches

With the bug fix in place the task is to merge it back into the master branch So other users could can use it as well. Assuming the master branch is then pushed to a remote repository that is.

git merge

Merge is described a replaying the changes of a named commit (aka branch) to a another branch since they diverged. For this to work one changes first to the target branch. In this case this is **`bugfix`** changes are to be replayed to **`master`**. As the target is **`master`** the first step is to change to that branch.

```
$ git checkout master (1)
Switched to branch 'master'
$ git branch
* master (2)
  testing
$ git merge bugfix (3)
Updating e303af7..a27a927
Fast-forward
 first-file.txt | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

(1) Switch to the target branch (**`master`**).

(2) Confirm being on the target branch. This step is optional.

(3) Replay or merge the changes from **`bugfix`** into **`master`**.

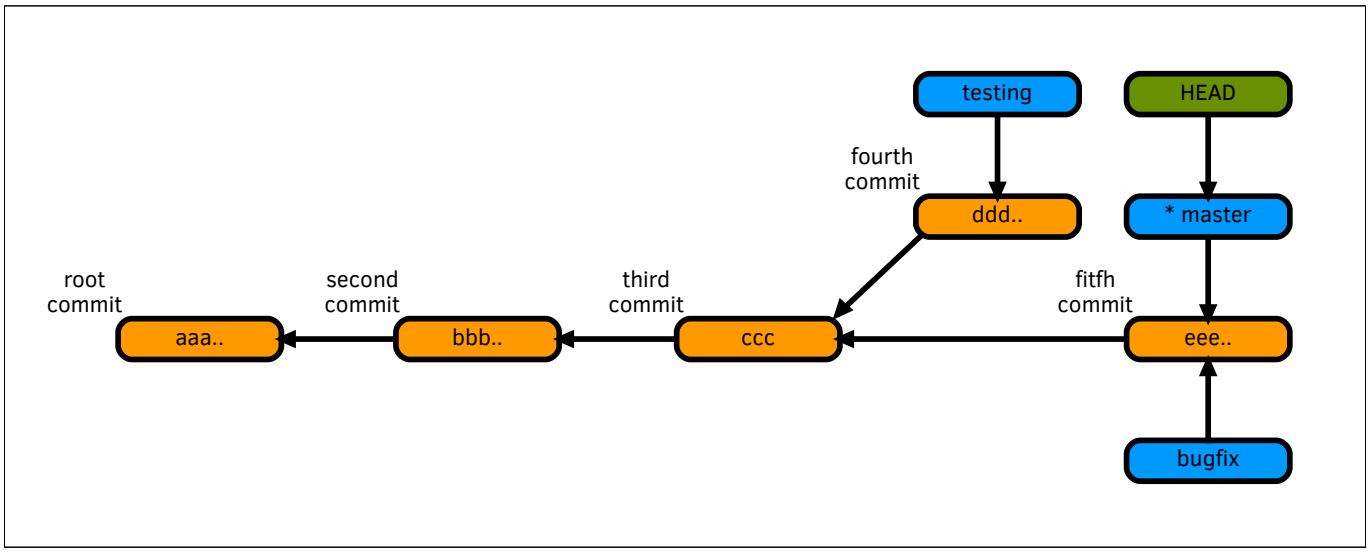

*Figure 22. After the merge **`bugfix`** and **`master`** point to the same revision.*

git branch -d

There is no reason to keep the **`bugfix`** branch around now that the changes have been incorporated into master. With the **`branch -d <branchname>`** command the branch is deleted.

```
$ git branch
  bugfix
* master (1)
  testing
$ git branch -d  bugfix (2)
Deleted branch bugfix (was a27a927).
$ git log  --oneline -n 1
a27a927 (HEAD -> master) Bugfix for first file (3)
```

(1) The branch cannot be delete when checked out. Active branch is **`master`** which will work for deletion of **`bugfix`**.

(2) Branch is deleted and the output contains the short SHA1 hash.

❸ Checking with **`git log`** confirms **`master`** points to the same hash as **`bugfix`** was.

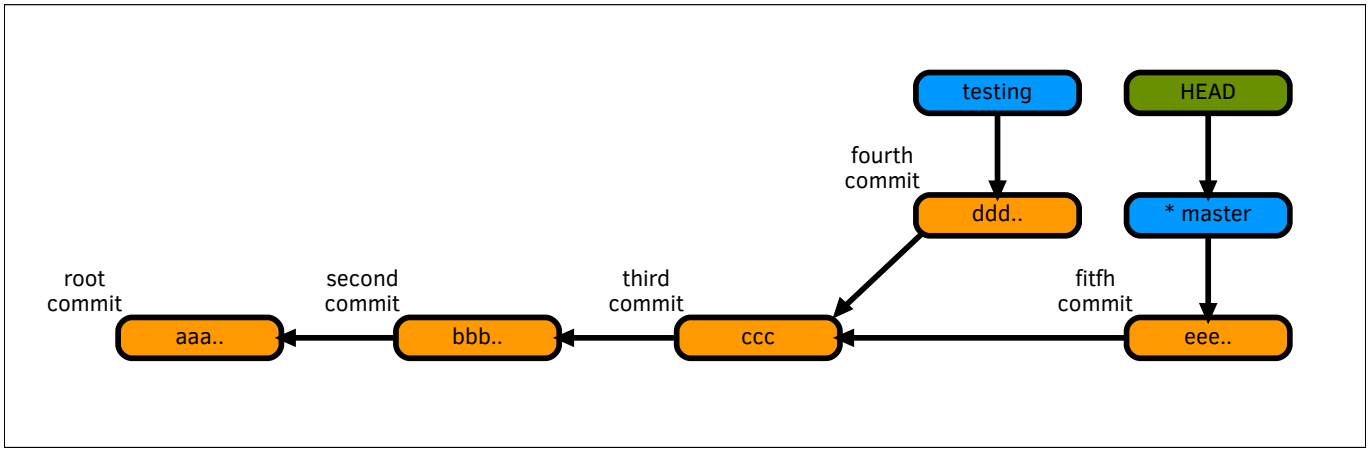

Figure 23. With the **`bugfix`** *branch deleted only* **`master`** *and* **`testing`** *remains.*

Rebasing branches

With the bug fix merged into branch **`master`** the next logical step is to fold the changes into the testing branch to ensure the next release does include the fixed version. When working with multiple branches this operation is required to not fall back to far with **`master`** and preventing lots of merge conflicts.

git rebase

Rebasing is shifting the parent commit of the first change in the branch and attach it to the current pointer of the branch or commit given on the command line as argument. In the below example we the first action is to change to the branch to rebase and then issue the **`rebase`** command against **`master`**.

```
$ git branch ❶
* master
  testing
$ git checkout testing ❷
Switched to branch 'testing'
$ git rebase master ❸
Successfully rebased and updated refs/heads/testing. ❹
```

❶ Currently on branch **`master`**, required to change to **`testing`** prior to the rebase.

❷ Switch to branch **`testing`** which will be rebased with **`master`**.

❸ Issue rebase command with argument **`master`** the branch or pointer used for the rebase.

❹ The message is terse and refers to the git internal file structure under the **`.git`** directory.

Conducting a rebase between two branches requires a common ancestor in the tree.

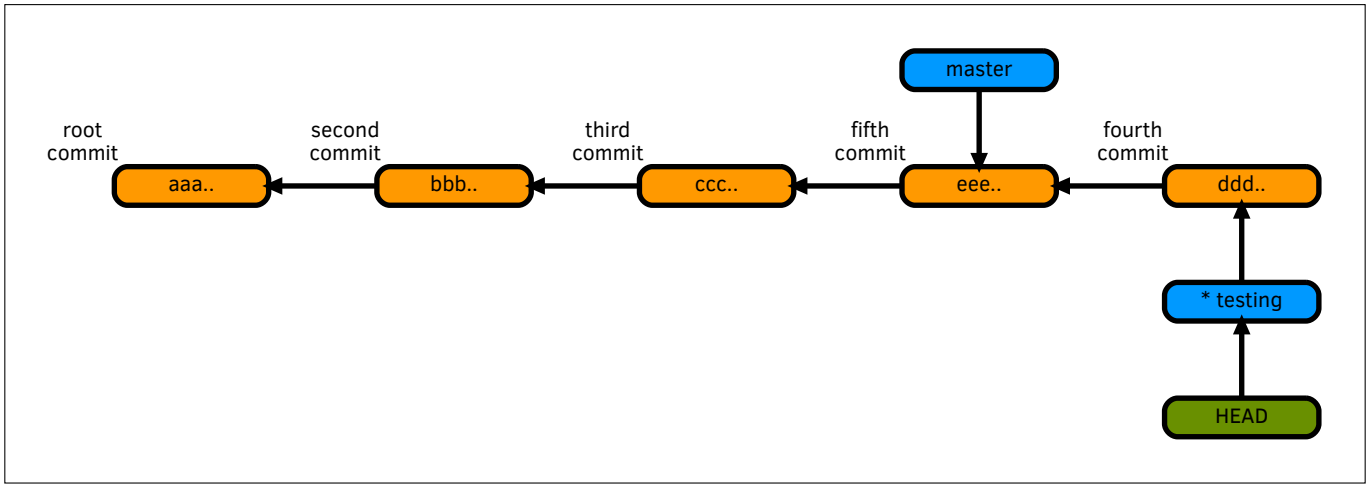

Figure 24. After the rebase; **`master`** *and* **`testing`** *are again in sync.*

Renaming branches

As learned earlier branches are just arbitrary names pointing to a revision within the git tree. As such renaming is a very painless and fast operation. No files have to be copied just the reference requires an update.

git branch -m

The **`-m`** option together with the **`branch`** command moves a branch or more appropriately renames the branch to the desired new name. Renaming can be from outside of the branch by first providing the current branch name followed by the desired branch name. In this example switching the branch first before issuing the **`branch -m`** command is used.

```
$ git branch ❶
* master
  testing
$ git checkout testing ❷
Switched to branch 'testing'
$ git branch -m production ❸
$ git branch
  master
* production ❹
```

❶ Currently on branch **`master`**, required to change to **`testing`** prior the rename.

❷ Switch to branch **`testing`** which will be renamed thereafter.

❸ Issue the **`-m`** move command with new branch name **`production`**. In true Unix fashion git does not output any message. Alternatively the command **`git branch -m testing production`** can be used instead of the 4 commands here.

❹ Branch **`testing`** is now called **`production`**.

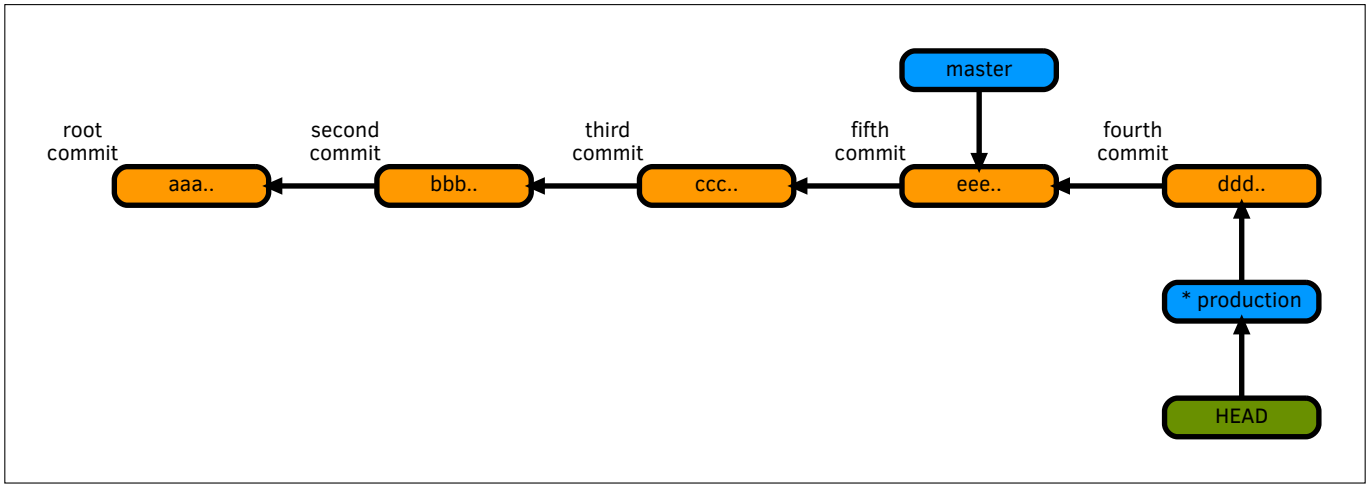

Figure 25. Structurally there is no change other then the branch name now being **`production`**.

Resolving merge conflicts

A merge conflict is a situation where the same file has been modified by one or more person at the same location in the file. Generally git does an excellent job working around merge conflicts. But occasionally during **`merge`** or **`rebase`** operations git interrupts and requires human intervention to solve a conflict between two revisions.

As a general rule many merge conflicts can be prevented or minimized by:

- Communicating changes between team members regularly.
- Regular rebases with the merge target branch.
- Creating small and atomic commits.

git commit (conflict)

To show a merge conflict the first thing to do to change the same file in two different branches and then rebase both branches.

Edit file **`first-file.txt`** *under branch* **`master`**.

```
$ git checkout master (1)
Switched to branch 'master'
$ vi first-file.txt (2)
$ cat first-file.txt
First file with bugfix from branch "master" (3)
$ git commit -m "first-file: Add from 'master'" first-file.txt  (4)
[master 64b6795] first-file: Add from 'master'
 1 file changed, 1 insertion(+), 1 deletion(-)
```

(1) Switch to branch **`master`**.

(2) Edit file **`first-file.txt`** and append **`from "master"`** to the first line.

(3) Confirm output with **`cat`** or **`git diff`**.

(4) Commit change.

Edit file **`first-file.txt`** *under branch* **`production`**.

```
$ git checkout production ❶
Switched to branch 'production'
$ vi first-file.txt ❷
$ cat first-file.txt ❸
First file with bugfix from branch "production"
$ git commit -m "first-file: Add from 'production'" first-file.txt ❹
[production bdfdc4a] first-file: Add from 'production'
 1 file changed, 1 insertion(+), 1 deletion(-)
```

❶ Switch to branch **`production`**.

❷ Edit file **`first-file.txt`** and append **`from "production"`** to the first line.

❸ Confirm output with **`cat`** or **`git diff`**.

❹ Commit change.

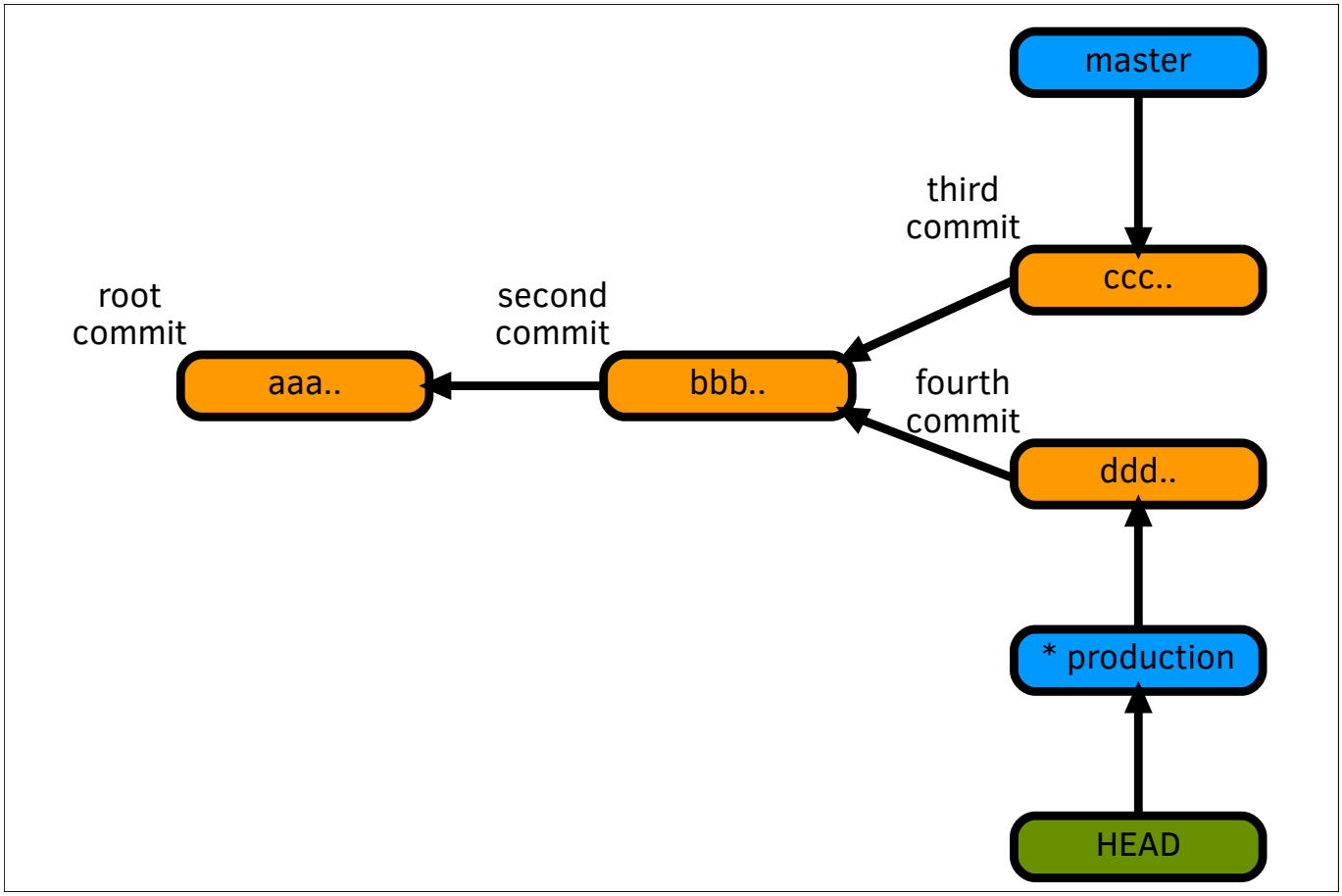

Figure 26. There are now new commits in both **`master`** *and* **`production`**.

git rebase (conflict)

With the conflicting changes in place a rebase of branch **`production`** with branch **`master`** is initiated. Unlike in the rebase example before **`git`** is not able to resolve the differences in the file and interrupts the rebase process.

```
$ git rebase master
Auto-merging first-file.txt (1)
CONFLICT (content): Merge conflict in first-file.txt (2)
error: could not apply bdfdc4a... first-file: Add from 'production'
Resolve all conflicts manually, mark them as resolved with (3)
"git add/rm <conflicted_files>", then run "git rebase --continue".
You can instead skip this commit: run "git rebase --skip".
To abort and get back to the state before "git rebase", run "git rebase --abort".
Could not apply bdfdc4a... first-file: Add from 'production'
```

(1) Notice that **`first-file.txt`** will be automatically merged as both branches have changes in this file. If the changes were in different parts of the file auto merge would merge and continue.

(2) A conflict was detected in file **`first-file.txt`**. If more than one file runs into merge conflicts they will be listed as well.

(3) Git instructions how to proceed. Either resolve the conflict and then the git commands to use to continue with the rebase. Or simply skip the change. At this point it is up to the person behind the keyboard to make a decision how to proceed.

Inspecting the current environment

```
$ git branch (1)
* (no branch, rebasing production) (2)
  master
  production
```

(1) Issue git branch to see the current state of git.

(2) During the rebase an unamed branch is created for conducting the merge operation this branch is aptly named **`(no branch, rebasing production)`**. The branch will disappear after the rebase has concluded.

Inspecting the file with the merge conflict.

```
$ cat first-file.txt (1)
<<<<<<< HEAD (2)
First file with bug fix from branch "master" (3)
======= (4)
First file with bug fix from branch "production" (5)
>>>>>>> bdfdc4a... first-file: Add from 'production' (6)
```

(1) The contents of the file **`first-file.txt`** was modified by the git merge operation and has to be modified before continuing.

(2) The line starting with **`<<<<<<< HEAD`** refers to the version found in the revision from branch **`master`**

(3) Lists one or more lines of the conflicting changes with content from **`master`**.

(4) Delimiter, below the changes from branch **`production`** start.

(5) List one or more lines of the conflicting changes with content from **`production`**.

❻ The SHA1 and commit message of the change causing the conflict.

Manually edit file (conflict)

A conflict can be resolved by editing the file in question and a decision is made for each line with a conflict which one to pick. This strategy is recommended when neither the version from **`HEAD`** in this case **`master`** nor the one from the to be merged revision in this case **`production`** can be used.

```
$ vi first-file.txt ❶
$ cat first-file.txt
First file with bugfix from branch "master" and from "production" ❷
$ git add first-file.txt ❸
$ git rebase --continue ❹
❺
[detached HEAD 7c857af] first-file: Add from 'master' and 'production'
 1 file changed, 1 insertion(+), 1 deletion(-)
Successfully rebased and updated refs/heads/production.
$ git branch
  master
* production ❻
```

❶ Edit the file to be merged.

❷ All the metadata about the merge conflict has been removed and the conflicting line has been modified to contain both changes.

❸ As mentioned in the instruction when the conflict occurred the next after resolving the conflict is to add the modified file to staging.

❹ And then continue with the rebase.

❺ Due to the manual changes made an interactive commit session pops up. One can change the commit message if desired.

❻ The temporary branch has is gone an the branch **`production`** is the **`HEAD`** again.

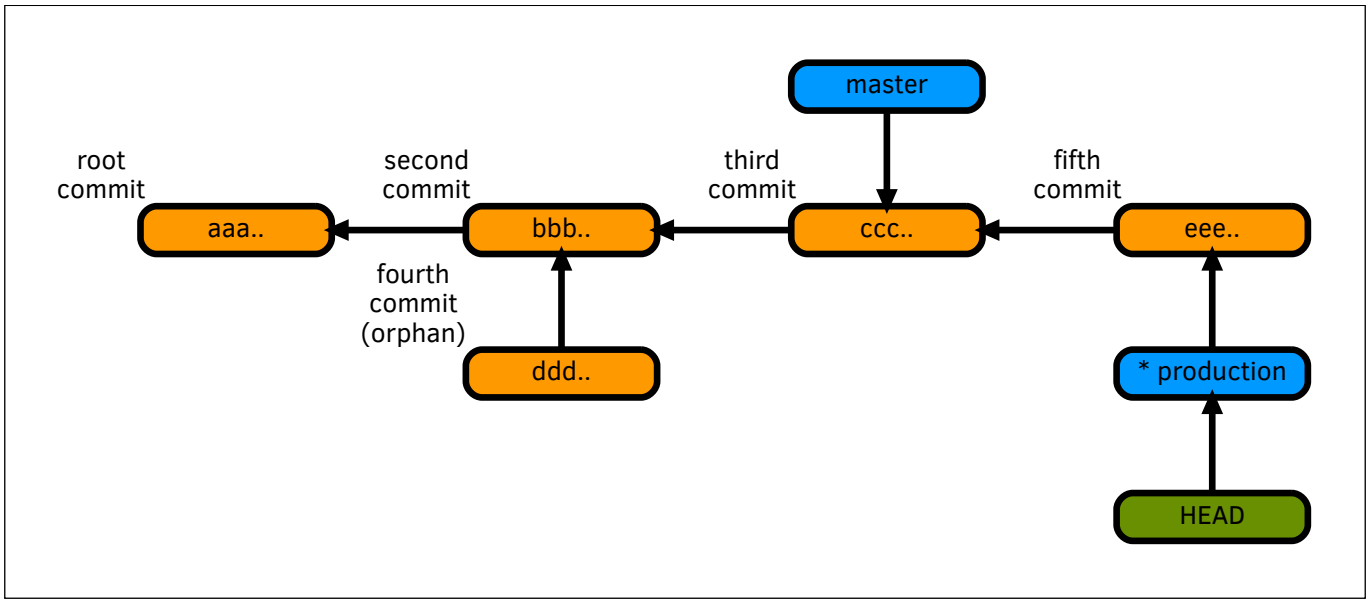

Figure 27. There is an orphan commit **`ddd…`** *and a new one* **`eee…`** *with parent* **`ccc…`**

git checkout --theirs (conflict)

A different merge strategy is to use the file from **`production`** and discard the changes from the **`master`**. In the end the process and how the git tree ends up looks eventually the same as for the manual process but the content of the file is different.

To use the version from branch **`production`** checkout is issued with switch **`--theirs`**.

```
$ git checkout --theirs -- first-file.txt (1)
$ cat first-file.txt
First file with bugfix from branch "production" (2)
$ git add first-file.txt (3)
$ git rebase --continue (4)
(5)
[detached HEAD 45a6088] first-file: Add from 'production'
 1 file changed, 1 insertion(+), 1 deletion(-)
Successfully rebased and updated refs/heads/production.
$ git branch
  master
* production (6)
```

(1) Use the checkout function with switch **`--ours`** and the file name to be checked out from **`HEAD`**.

(2) The file's content is now the same as the one from branch **`master`**.

(3) As mentioned in the instruction when the conflict occurred the next after resolving the conflict is to add the modified file to staging.

(4) And then continue with the rebase.

(5) With the **`--theirs`** option the file's content compared to the version of **`master`** is being modified. A new commit is required and the user is prompted to modify the commit message if so desired.

(6) The temporary branch has is gone and the branch **`production`** is the **`HEAD`** again.

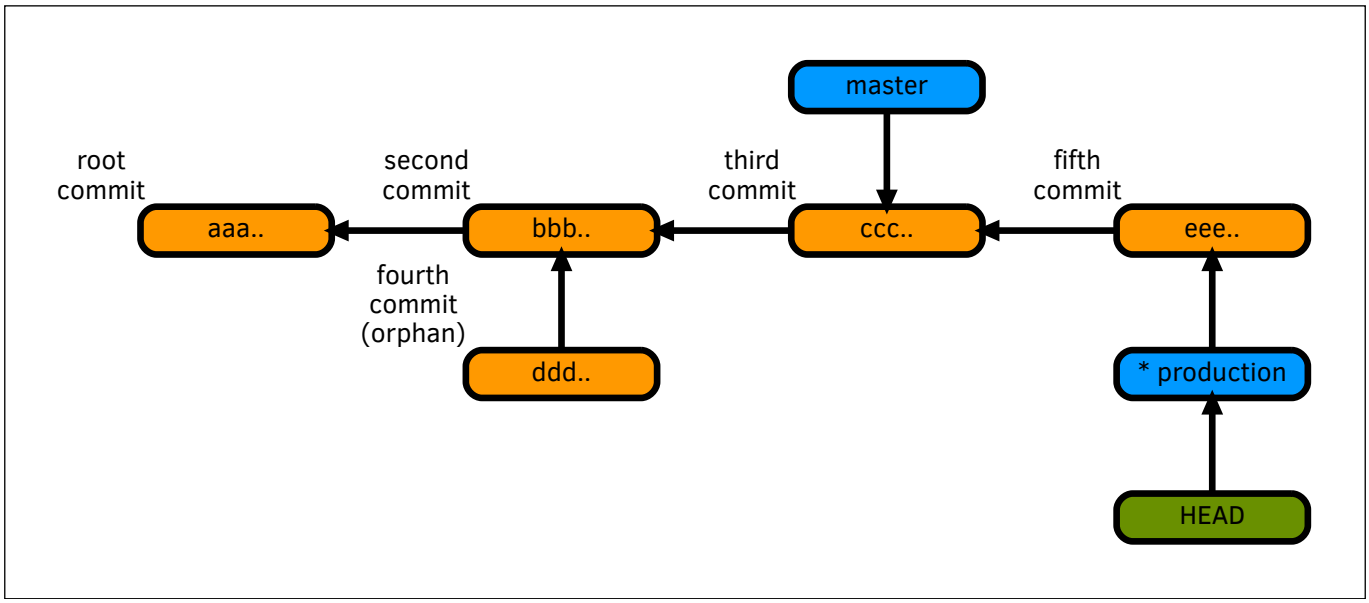

Figure 28. There is an orphan commit **`ddd…`** *and a new one* **`eee…`** *with parent* **`ccc…`**.

git checkout --ours (conflict)

A different merge strategy is to use the file from **`master`** and discard the changes from the **`production`** commit. This could be done by editing the file manually as shown before which in case of the task at hand with only one line certainly is an option. However if the merge conflict is hundreds of lines scattered throughout the file it is a daunting and error prone task.

The git command **`checkout`** has an option called **`--ours`** to use use the version currently present in **`HEAD`**.

```
$ git checkout --ours -- first-file.txt ❶
$ cat first-file.txt
First file with bugfix from branch "master" ❷
$ git add first-file.txt ❸
$ git rebase --continue ❹
Successfully rebased and updated refs/heads/production.
$ git branch
  master
* production ❺
```

❶ Use the checkout function with switch **`--ours`** and the file name to be checked out from **`HEAD`**.

❷ The file's content is now the same as the one from branch **`master`**.

❸ As mentioned in the instruction when the conflict occurred the next after resolving the conflict is to add the modified file to staging.

❹ And then continue with the rebase.

❺ The temporary branch has is gone and the branch **`production`** is the **`HEAD`** again.

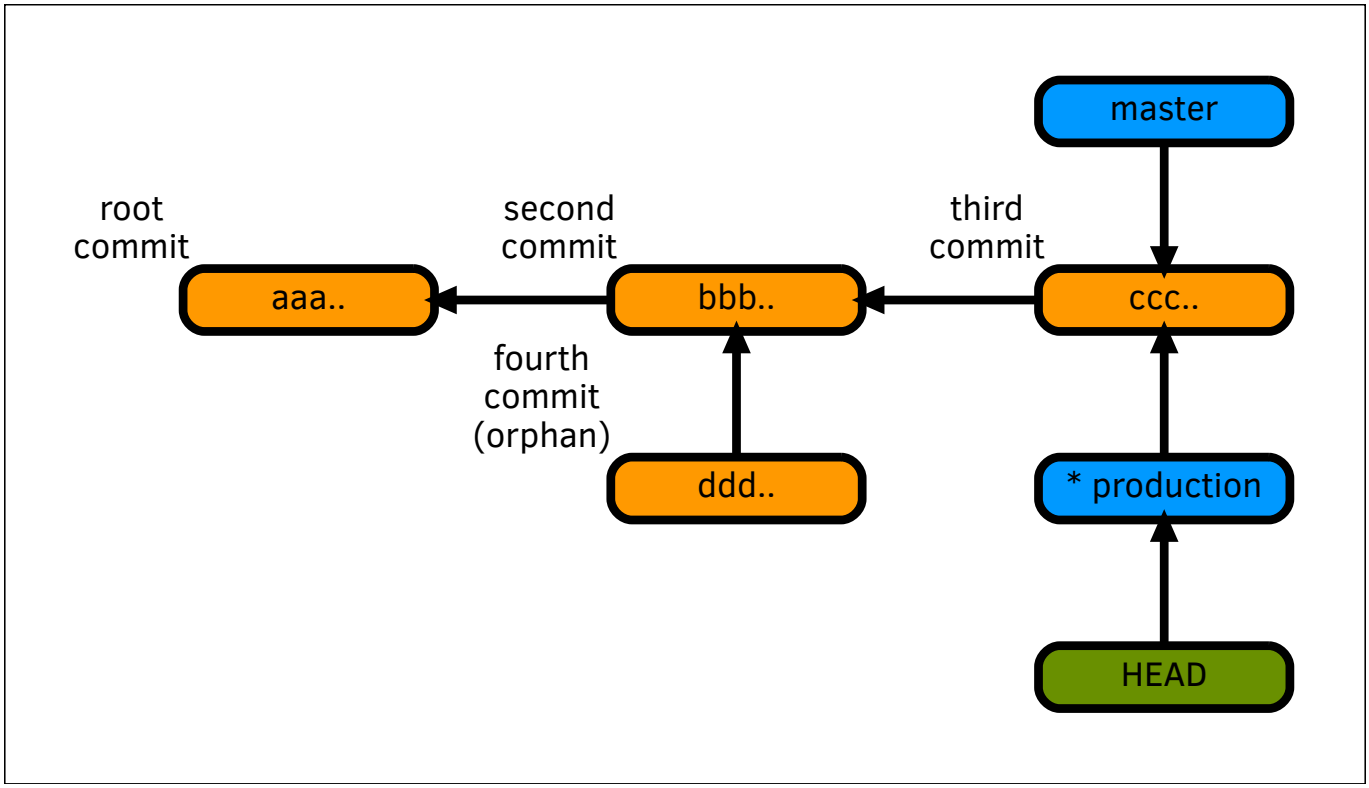

Figure 29. Both branches **`master`** *and* **`production`** *point to* **`ccc…`** *and* **`ddd…`** *became an orphan.*

Summary

Branching is a important concept in git for further reading With all the basic research tools under our belt here the summary of the commands.

Item	Description	Resource
`git branch`	Create and manage branches.	man git-branch
`git checkout`	Switch and create new branches.	man git-checkout
`git rebase`	Keep branches up to date with external changes.	man git-blame

Module 7 - Customizing git commands

Goals

In true Unix fashion git can be customized to suit one's need. There is the option of creating aliases for often used commands or creating custom git commands in one's favorite language.

- Create command short cuts.
- Deleting aliases.
- Configure aliases for often used commands.
- Solve complex recurring workflows with custom git scripts.

Command shortcuts

For some the git commands such as **`checkout`** are too cumbersome to type each time what if one could shorten that to say **`co`** like on subversion.

git config alias

Aliases are put in place with the **`config`** command and can be local, global or system wide. For this exercise the global scope is used as it makes the most sense. Convenience settings are best shared among more than just one repository.

```
$ git config --global alias.co checkout ❶
$ git config --global alias.ci commit
$ git config --global alias.br branch
$ git config --global alias.st status

$ git st --short ❷
A  second-file.txt
```

❶ The alias is defined as a single command.

❷ The alias for **`status`** can be invoked with options.

While simple alias shortcuts are certainly useful one can also create aliases containing some of the rather long options.

```
$ git config --global alias.lol  'log --oneline --no-decorate' ❶
$ git config --global alias.top 'log -n 3 HEAD' ❷

$ git lol
49b7a9c The rest ❸
e514995 Third line
2e97d38 Second line
8070030 First commit
```

❶ When using a command plus options enclose in quotes.

❷ Besides options keywords such as **`HEAD`** can also be used.

❸ Executes previously defined alias **`git log --oneline --no-decorate`**.

To modify an alias one can simply use the same command as when creating it.

```
$ git config --global alias.top 'log -n 1 HEAD' ❶

$ git top
commit 49b7a9cf2a8f1ae6ba94141268716ba0b07949d6 (HEAD -> master)
Author: Urs Roesch <github@bun.ch>
Date:   Tue Nov 3 05:23:49 2020 +0000

    The rest
```

❶ Change from **`-n 3`** to **`-n 1`** to only show the most recent revision.

To remove an alias the switch **`--unset`** is used.

```
$ git config --global --unset alias.top ❶
$ git config --global --get-regex 'alias.*' ❷
alias.st=status
alias.lol=log --oneline --no-decorate
alias.co=checkout
alias.ci=commit
alias.br=branch
```

❶ Removing alias **`top`** with the **`--unset`** switch.

❷ Verifying the deletion of the alias **`top`**.

Complex aliases

Aliases in git can be more than just shortcuts for overly lengthy commands. One can cram multiple commands into a single alias. This can be done by prefixing by starting the command with an exclamation mark **`!`**.

git config alias

The first example does chain two git commands together for syncing a forked repository on GitHub with the upstream repository.

```
$ git config --global alias.sync-upstream \
  '!sh -x -c "git fetch upstream && git rebase upstream/master master"' ❶

$ git sync-upstream
+ git fetch upstream ❷
From https://github.com/sample/repository ❸
   41cc734..1bd94bb  master     -> upstream/master
 * [new tag]         v1.60      -> v1.60
+ git rebase upstream/master master ❹
Current branch master is up to date. ❺
```

❶ Execution a shell command that contains two **`git`** commands the first fetches that new changes from the previously configured remote name **`upstream`** and the second runs a rebase between **`upstream/master`** and the local **`master`** branch.

❷ When executing the alias. First the **`fetch`** triggers.

❸ In this instance the fetch finds a new tag.

❹ The second command is invoked rebasing with master.

❺ Command output from the rebase command.

While the above example is fairly sophisticate there is no way to pass parameters to the alias. With the next sample a list of files is passed to the **`vi`** editor and when finished editing the files are added to staging area.

```
$ git config --global alias.vi '!sh -x -c "vi \"$@\" && git add \"$@\""' ❶

$ git vi second-file.txt
+ vi second-file.txt ❷
+ git add second-file.txt
```

❶ Enclosing the command in single quotes and adding the optional switch **`-x`** to visualize the commands being executed.

❷ Visualizing the commands executed due to the **`-x`**.

The sky is the limit! If a difficult operation can be put into a simple alias go for it.

Custom commands

For some task even a complex alias is not cutting it! For such instances there is the option of creating a custom command. Any programming language available on the system can be used to do so. To integrate the newly minted command into **`git`** the script must be named **`git-<command>`** and be placed in a directrory included in **`$PATH`**.

git opush (bash script)

The script sampled below implements the command **`opush`** a shortcut for **`push origin <branch>`** and **`push --tags`** While this can be implemented easily with an alias there are a few security precaution like not pushing the **`master`** branch as the switches **`--force`** and

`--remove` could wipe out the main branch of the remote repository.

```
#!/usr/bin/env bash

# ------------------------------------------------------------------------------
# Small script to push upstream without a fuss
# ------------------------------------------------------------------------------

# ------------------------------------------------------------------------------
# Setup
# ------------------------------------------------------------------------------
set -o errexit
set -o nounset
set -o pipefail

# check bash version compatiblity requires 4.2 or better
shopt -u compat41 2>/dev/null || {
  echo -n "\nBash Version 4.2 or higher is required!\n";
  exit 127;
}


# ------------------------------------------------------------------------------
# Globals
# ------------------------------------------------------------------------------
declare -r SCRIPT=${0##*/}
declare -r VERSION=0.3.1
declare -r AUTHOR="Urs Roesch <github@bun.ch>"
declare -r LICENSE="GPLv2"
declare -g FORCE=""
declare -g REMOVE=""

# ------------------------------------------------------------------------------
# Functions
# ------------------------------------------------------------------------------
function usage() {
  local exit_code=${1:-1}
  cat <<USAGE

  Usage:
    ${SCRIPT//-/ } [options]

  Opttions:
    -h | --help    This message
    -f | --force   Force a push to upstream
    -r | --remove  Remove the repository from upstream
    -V | --version Display version and exit

  Description:
    Origin push to upstream without a fuss. Exludes pushes to master.

USAGE

  exit ${exit_code}
}
```

```
# ------------------------------------------------------------------------------

function parse_options() {
  while [[ ${#} -gt 0 ]]; do
    case ${1} in
    -h|--help)    usage 0;;
    -f|--force)   FORCE="true";;
    -r|--remove)  REMOVE="true";;
    -V|--version) version;;
    -*)           usage 1;;
    esac
    shift
  done
}

# ------------------------------------------------------------------------------

function version() {
  printf "%s v%s\nCopyright (c) %s\nLicense - %s\n" \
    "${SCRIPT}" "${VERSION}" "${AUTHOR}" "${LICENSE}"
  exit 0
}

# ------------------------------------------------------------------------------

function current_branch() {
  git rev-parse --abbrev-ref HEAD
}

# ------------------------------------------------------------------------------

function push_origin() {
  local branch=$(current_branch)
  if [[ ${branch} == master ]]; then
    echo "Not pushing master!"
    exit 1
  fi
  git push ${FORCE:+-f} origin ${REMOVE:+:}${branch}
}

# ------------------------------------------------------------------------------

function push_tags() {
  git push --tags
}

# ------------------------------------------------------------------------------
# Main
# ------------------------------------------------------------------------------
parse_options "${@}"
push_origin
push_tags
```

Placing the script **`git-opush`** under **`${HOME}/bin`** wich is in the path one can execute with **`git opush`**. In below case the command is invoked while under branch **`master`**.

```
$ git opush
Not pushing master!
```

Summary

Tags are a big help in locating important milestones and releases and help with navigating a repository quickly.

Item	Description	Resource
`git config`	Manage aliases and other configuration items.	man git-config