GIT version control system

Version control system

A version control system is a software tool used for:

- tracking changes in the source code
- helping developers combine changes made to files by many people at different points in time

The version control system idea showcases that even if a mistake or partial data loss is made, it should be easy and inexpensive to repair and recover it.

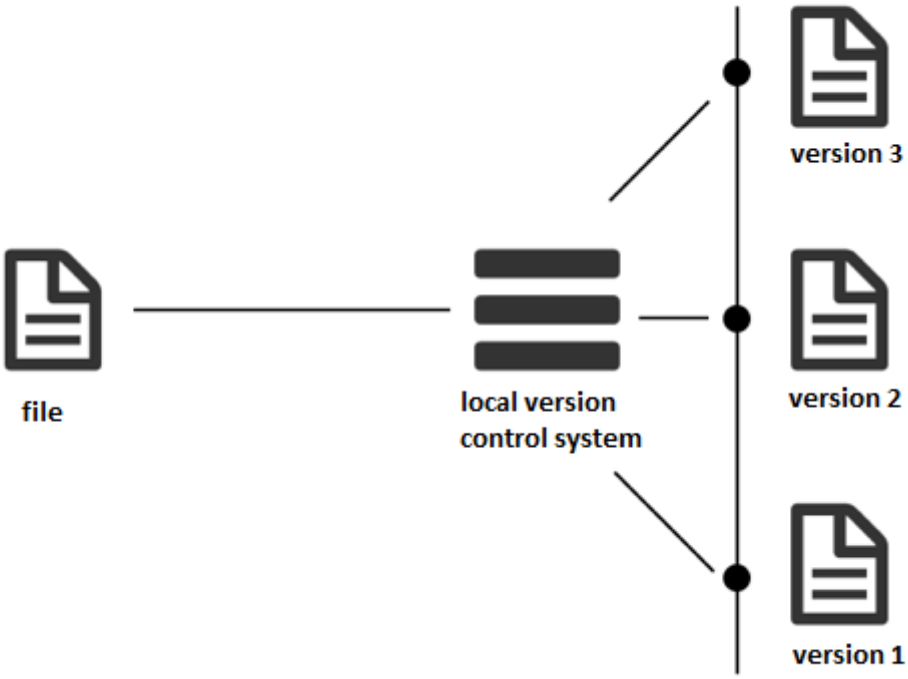
This software enables the user to:

- restore the file(s) to an earlier version
- recreate the state of the entire project
- compare introduced changes with previous modifications
- find out the most recent modification to the project that caused the problems, it's author and date of origin

There are three types of version control systems:

- local
- centralized
- scattered

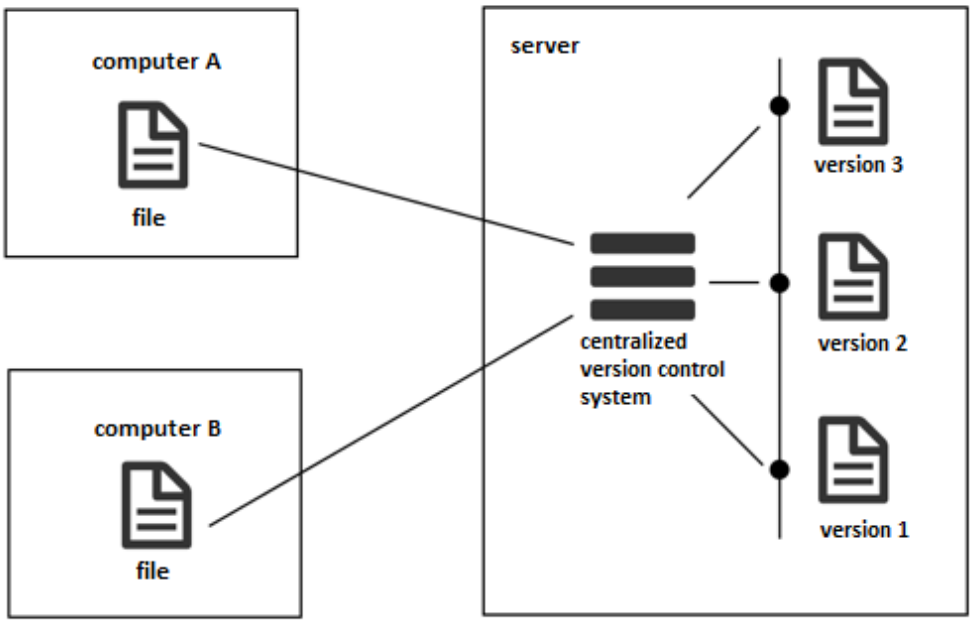
The local version control system



This software consists of a simple database in which all changes, that are made to the tracked files, are stored. The disadvantage of this system is that in the event of a database loss, it is not possible to restore the saved files. Additionally, data sharing and synchronization is impossible in an easy and effective way.

An example would be the `rcs` system. This program saves the differential data in a special format on the disk. With differential data, you can define data that contains only the differences between the files for each modification that was made. Using this data the `rcs` system is able to recall the state of a file from any point in time.

Centralized version control system



It has been the standard version control model for many years. The system consists of a single server that contains all version-controlled files, users (who can connect to it), and the latest versions of the files. This solution has many advantages, including:

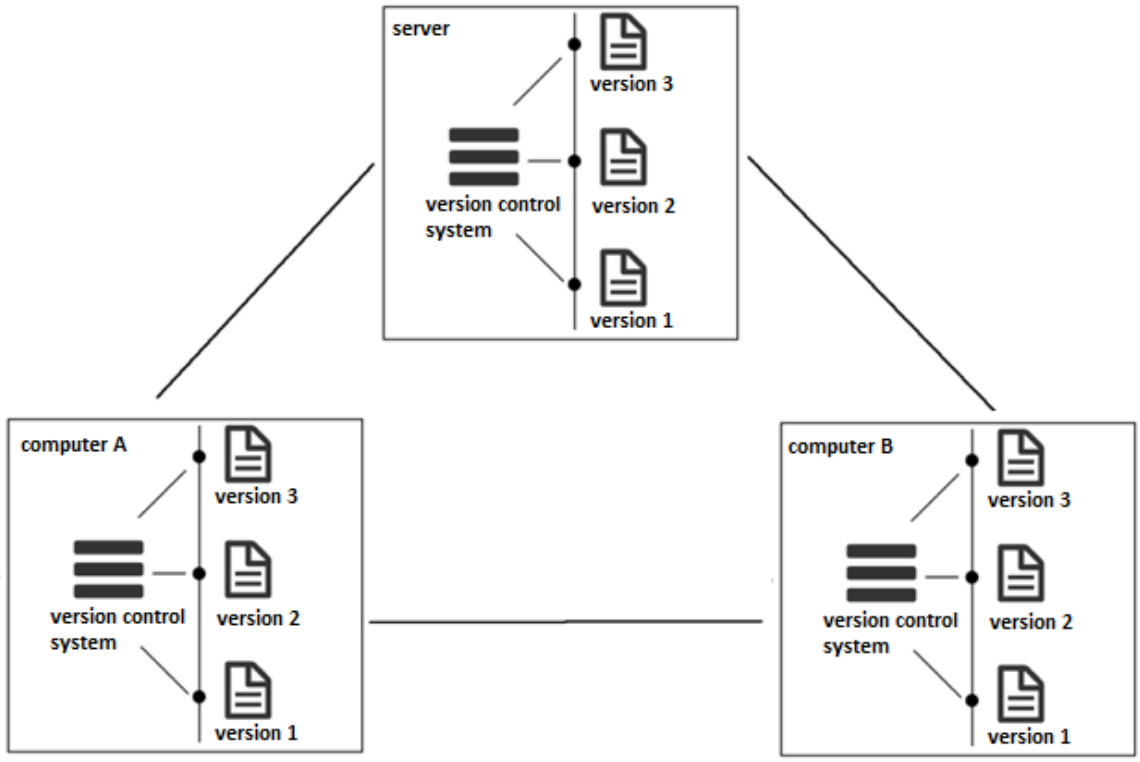
- easy insight into the activity of its users
- administrators have strict control over user permissions
- it is easier to manage than a local system

This system has one major disadvantage. In the event of a failure of the central server, there is a high probability of losing most of the data (in the absence of backups). Additionally, during failure, there is no possibility for the system users to take any action. This is the same problem as for local version control systems. If the entire history of a project is kept in only one place, users risk losing most of your data.

Examples of screntralized VCS are:

- CVS
- Subversion
- Perforce

Distributed version control system



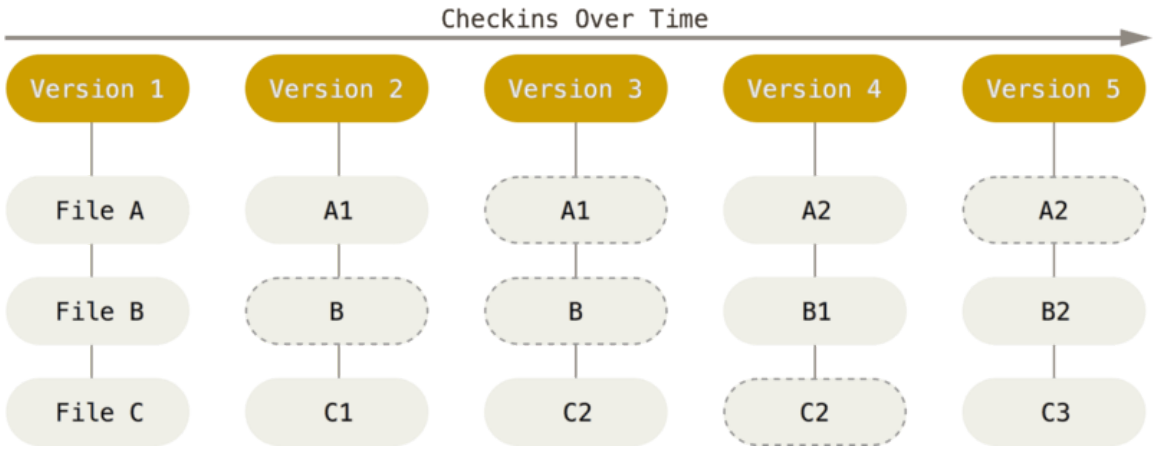
It is a system in where users get access to the latest versions of files and are able to copy the entire repository. In the event of a failure of one of the VCS systems, each client's repository can simply be copied to that server in order to restore it to work. Additionally, this system deals effectively with several remote repositories, so it is possible to simultaneously collaborate with different groups on the same project.

Examples of distributed VCS are:

- Git
- Mercurial
- Bazaar
- Darcs

Git

Distributed version control system developed by Linus Torvalds as a tool to aid the development of the Linux kernel. It is an open-source, highly scalable software, applicable to projects of all sizes.



Git fixes the following two problems:

- version management over time
- change management in the team

This system has the following features:

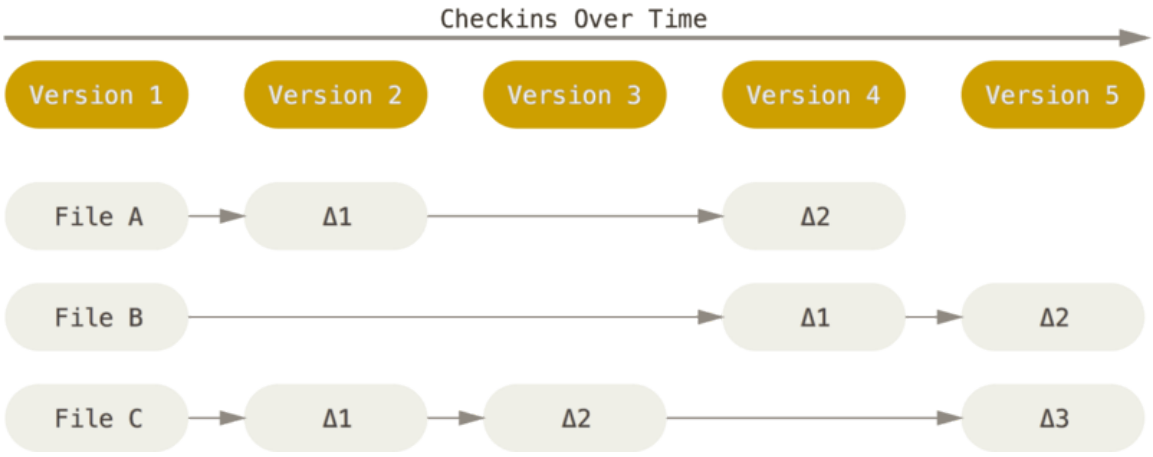
- it is free
- it is an open source solution
- it is platform independent
- it supports text and graphic mode
- it is fast and efficient
- it is file-based
- it is distributed

Its disadvantages include:

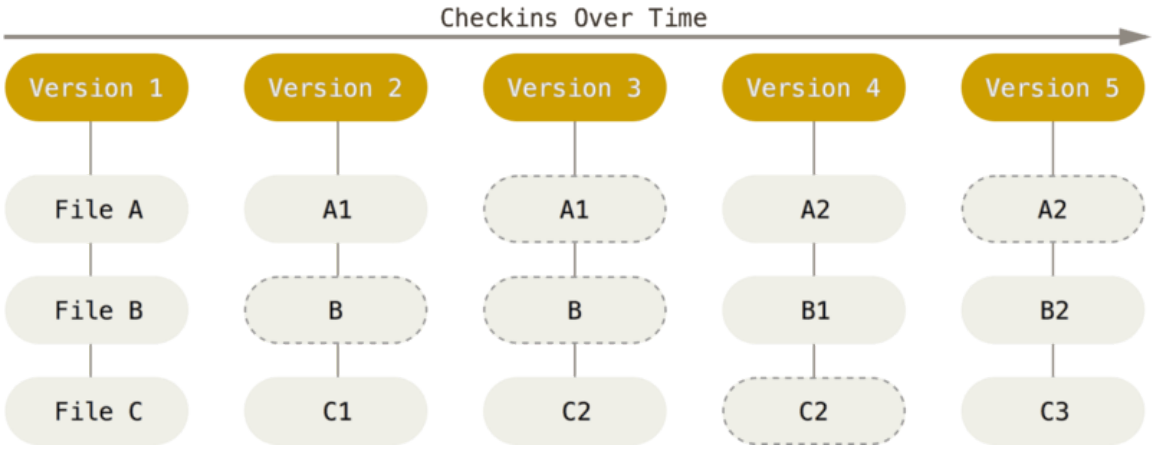
- no built-in user rights management mechanism
- no possibility to track empty folders

Approach to data storage

Most version control systems store information as a list of changes to files. Systems such as CVS, Subversion, Perforce, Bazaar treat information as a collection of files and changes that were made over a period of time.



Git treats data like a set of snapshots - aka. a small filesystem. Each time a commit is made or the state of a project is saved, Git creates an image of what all files look like at that point in time and stores a reference to that snapshot. For optimization purposes, if a given file has not changed, Git doesn't rewrite that file, only a reference to its previous, identical version that is already saved.



Data consistency mechanisms

Before a Git object is saved, a checksum is computed, which creates an identifier for the object. This makes it impossible to change the contents of any file or directory without Git's response.

The mechanism that Git uses to determine the checksum is the SHA-1 abbreviation. It is a 40-character string of hexadecimal numbers (0–9 and a – f). It is calculated based on the contents of the file or directory structure. The SHA-1 hash looks something like this:

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

Local nature of the operation

Git only requires local files and resources to run. It does not need to sync with other computers. This is due to the fact that the entire history of the project resides entirely on the local disk, which significantly affects the efficiency of the operation. It also means you can do almost anything you want when you are out of range of the network or corporate VPN. Changes can be committed locally and sent to a remote repository at a completely different time.

Initial Git configuration

The `git config` utility allows you to read and modify the variables that control all aspects of the Git version control system's operation and behavior. These variables can be stored in three different places:

- /etc/gitconfig file: it contains variable values visible to every user on the system and to each of their repositories. The configuration can be accessed using the `--system` option.
- file ~/.gitconfig: the location for this user. The configuration can be accessed with the `--global` option
- configuration file in the git directory (ie .git/config) of the current repository: contains configuration specific to this particular repository.

Each level has priority over the previous level, so the variable values in the .git / config file override the variable values in the / etc / gitconfig file.

Basic configuration

The basic configuration of each user should be based on the following commands:

```
$ git config --global user.name "SDA student"
$ git config --global user.email student@sda.pl
```

The command `git config --global user.name "SDA Student"` is responsible for setting up the username.

The command `git config --global user.email student@sda.pl` is responsible for setting up the email address.

This data is necessary because each commit you create uses this information.

Configuration verification

To view the current configuration, use the command `git config --list`:

```
$ git config --list
user.name=SDA student
```

```
user.email=student@sda.pl
color.status=auto
color.branch=auto
color.interactive=auto
color.diff=auto
...
```

Creating a repository

The possibility of creating a git project can be achived in two ways:

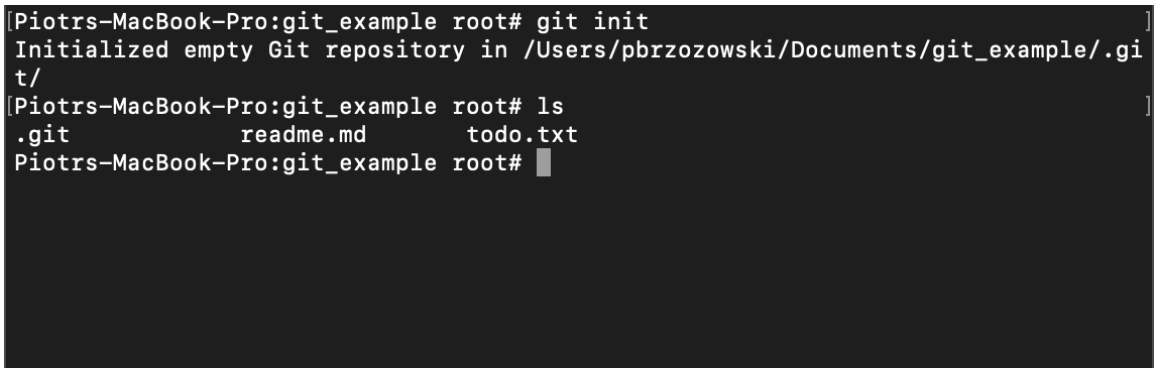
- by importing an existing repository into Git
- by cloning an existing repository from another server

Initializing Git in an existing directory

In order to start tracking changes in the files of an existing project, execute the command from the project directory level:

```
git init
```

This command will create a new subdirectory called `.git` containing all the necessary files - aka. the repository schema.



> ***NOTE:*** To create a default branch with other name, e.g. `main` use following command: `git init -b main`.

Clone an existing repository

In order to obtain a copy of an existing repository, use the following command:

```
git clone
```

After executing the `git clone` command, every revision of every file in the project's history will be downloaded.

In the event of a server becomeing damaged, it is possible to restore the server to the state it was in at the time of cloning.

The repository can be cloned with the command:

```
git clone [URL].
```

where `[URL]` is the url of the repository.

For example, if you want to clone the `GSON` library, execute the following command:

```
$ git clone https://github.com/google/gson.git
```

Log changes to the repository

In the Git repository, each time you can apply changes. Until the project reaches a state that requires saving, you must commit such changes at the repository level. Each file in the vault directory can be in one of two states:

- tracked
- untracked

Tracked files

The tracked files are the files that were included in the last snapshot. They can include:

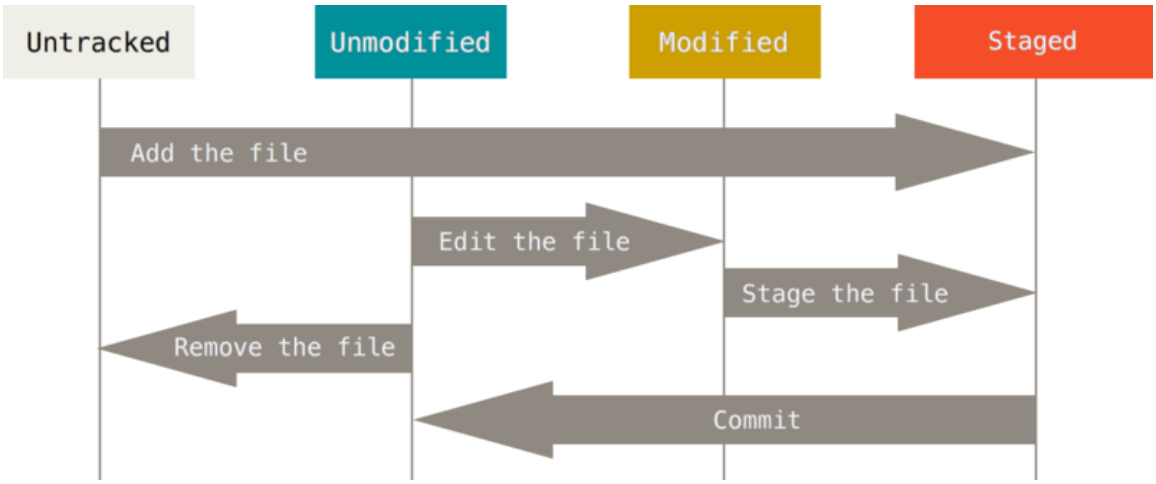
- unmodified
- modified
- those in the waiting list

Untracked files

These are any files in the vault that are not in the last snapshot and are not on the waiting list, they are ready to be committed.

After cloning the repository, all files in the root directory are tracked and unmodified.

File life cycle



When a file is modified, the Git version control system recognizes it as modified due to the difference between its current state and the last committed change. Modified or untracked files must be placed in the lobby before being approved.

Check the status of files in the repository

You can use the `git status` command to verify the status of files in the repository.

```
Piotrs-MacBook-Pro:git_example root# ls
.git            help.txt        readme.md       todo.txt
Piotrs-MacBook-Pro:git_example root# git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   readme.md

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   todo.txt

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

Piotrs-MacBook-Pro:git_example root#
```

Above is an example of calling this command. The following files have the following status:

- readme.md - the file is in the vault
- todo.txt - the file is modified
- help.txt - an untracked file

The command `git status -s` allows you to display an abbreviated form of the repository status.

```
Piotrs-MacBook-Pro:git_example root# git status -s
M  readme.md
 M todo.txt
?? help.txt
Piotrs-MacBook-Pro:git_example root#
```

In the abbreviated form of saving, the files have the following saving format:

```
XY PATH
XY ORIG_PATH -> PATH
```

The letters `X` and `Y` can contain the following statuses:

```
X          Y     Meaning
-------------------------------------------------
          [AMD]   not updated
M         [ MD]   updated in index
A         [ MD]   added to index
D                 deleted from index
R         [ MD]   renamed in index
C         [ MD]   copied in index
[MARC]            index and work tree matches
[ MARC]     M     work tree changed since index
[ MARC]     D     deleted in work tree
[ D]        R     renamed in work tree
[ D]        C     copied in work tree
-------------------------------------------------
D           D     unmerged, both deleted
A           U     unmerged, added by us
U           D     unmerged, deleted by them
U           A     unmerged, added by them
D           U     unmerged, deleted by us
A           A     unmerged, both added
U           U     unmerged, both modified
-------------------------------------------------
?           ?     untracked
!           !     ignored
-------------------------------------------------
```

Adding changes to the waiting list

To start tracing a new file, use the command `git add filename`. This instruction puts the file to the waiting list. Whenever `git add --all` or `git add .` is invoked, any file that is not ignored is added to the waiting list. The same command adds modified files to the lobby. Only files in the lobby can be approved.

```
[Piotrs-MacBook-Pro:git_example root# git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   readme.md
        modified:   todo.txt

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

no changes added to commit (use "git add" and/or "git commit -a")
[Piotrs-MacBook-Pro:git_example root# git add --all
[Piotrs-MacBook-Pro:git_example root# git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   help.txt
        modified:   readme.md
        modified:   todo.txt
```

In the above example, two files from the `changes not staged for commit` category, i.e. `readme.md`, `todo.txt`, and one non-tracked file: `help.txt` were added to the waiting room. After executing the command, all files went to the section: `changed to be commited`, which means they can be committed to the system.

Approval of changes

Any file that was either created or modified and did not have the `git add` command executed at a later date will not be included in the commit changes. They will only remain as modified files on your hard drive.

The changes are committed with the `git commit` command. Entering the command without specifying additional parameters launches the user-selected text editor (selected via the `$EDITOR` environment variable - usually it is `vim`, `emacs`, `nano`. Defining your own application is possible using the command `git config --global core.editor`).

The editor will be opened with the following message:

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch master
# Changes to be committed:
#       new file:   help.txt
#       modified:   readme.md
#       modified:   todo.txt
#
                                [ Read 10 lines ]
^G Get Help     ^O WriteOut     ^R Read File    ^Y Prev Page    ^K Cut Text     ^C Cur Pos
^X Exit         ^J Justify      ^W Where Is     ^V Next Page    ^U UnCut Text   ^T To Spell
```

By the editor, it is possible to edit the message, which will be visible at the time of approving the changes.

Alternatively, it is possible to define the content of the message at the time of approving the changes. This is possible with the command `git commit -m" message body "`:

```
Piotrs-MacBook-Pro:git_example root# git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   help.txt
        modified:   readme.md
        modified:   todo.txt

Piotrs-MacBook-Pro:git_example root# git commit -m "add base project files"
[master 8699714] add base project files
 3 files changed, 2 insertions(+)
 create mode 100644 help.txt
Piotrs-MacBook-Pro:git_example root#
```

The above example shows the commit of changes to the branch `master`, with the SHA-1 checksum (8699714).

It is also possible to approve changes and bypass the waiting room. To do this, use the `-a` attribute, e.g. `git commit -a -m "message_content"`.

```
Piotrs-MacBook-Pro:git_example root# git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   todo.txt

no changes added to commit (use "git add" and/or "git commit -a")
Piotrs-MacBook-Pro:git_example root# git commit -a -m "add item to todo list"
[master 8f976fb] add item to todo list
 1 file changed, 1 insertion(+)
Piotrs-MacBook-Pro:git_example root#
```

Deleting files

In order to remove a file from Git, first remove it from the group of tracked files and then confirm the changes. This can be done with the command `git rm`. This command also removes the file from the working directory, ie it will no longer be visible in the untracked files section.

```
Piotrs-MacBook-Pro:git_example root# ls
.git            help.txt        readme.md       todo.txt
Piotrs-MacBook-Pro:git_example root# git rm todo.txt
rm 'todo.txt'
Piotrs-MacBook-Pro:git_example root# git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        deleted:    todo.txt

Piotrs-MacBook-Pro:git_example root#
```

If a file is removed from the working directory, this file will be moved to the `Changes not staged for commit` section:

```
Piotrs-MacBook-Pro:git_example root# ls
.git            help.txt        readme.md       todo.txt
Piotrs-MacBook-Pro:git_example root# rm -f readme.md
Piotrs-MacBook-Pro:git_example root# git status
On branch master
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        deleted:    readme.md

no changes added to commit (use "git add" and/or "git commit -a")
Piotrs-MacBook-Pro:git_example root#
```

Rename files

Git doesn't track file movements directly, but does include the `git mv` command. It is used to rename a file in the repository, e.g.

```
Piotrs-MacBook-Pro:git_example root# ls
.git            help.txt        readme.md       todo.txt
Piotrs-MacBook-Pro:git_example root# git mv readme.md Readme.md
Piotrs-MacBook-Pro:git_example root# git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        renamed:    readme.md -> Readme.md

Piotrs-MacBook-Pro:git_example root#
```

This command allows you to remember the rename operation on the waiting list. This is equivalent to running the following commands:

```
mv readme.md Readme.md
git rm readme.md
git add Readme.md
```

Analysis of changes on the waiting list and beyond

Use the `git status` command to display general information about changes outside and in the lobby. However, this command cannot return more detailed information about specific changes made to specific files. The `git diff` command returns information about exactly which lines were added and which were removed:

```
Piotrs-MacBook-Pro:git_example root# git diff
diff --git a/todo.txt b/todo.txt
index c05c7f7..893652d 100644
--- a/todo.txt
+++ b/todo.txt
@@ -1,2 +1,2 @@
-Todo list
-- git commit -a -m "content"
+Todo list:
+- git diff
Piotrs-MacBook-Pro:git_example root#
```

The example below shows detailed information about the files that have entered the waiting list.

If you need to view detailed information about what has ended up on the waiting list, use the command `git diff --staged`:

```
Piotrs-MacBook-Pro:git_example root# git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   readme.md

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   todo.txt

Piotrs-MacBook-Pro:git_example root# git diff --staged
diff --git a/readme.md b/readme.md
index 712e7f4..d9c22b6 100644
--- a/readme.md
+++ b/readme.md
@@ -1 +1,3 @@
-readme.md
+
+Readme.md
+- Git version control system
Piotrs-MacBook-Pro:git_example root#
```

Excluding files from the repository

There is often a need to exclude certain types of files from the repository. These are usually automatically generated files, such as project build files, event logs. A list of such files / patterns corresponding to a group of files is placed in a special `.gitignore` file, which may have the following content:

```
# Covers JetBrains IDEs: IntelliJ, RubyMine, PhpStorm, AppCode, PyCharm, CLion, Android Studio, WebStorm and Rider
#           Reference:           https://intellij-support.jetbrains.com/hc/en-us/articles/206544839
```

```
# User-specific stuff
.idea/**/workspace.xml
.idea/**/tasks.xml
.idea/**/usage.statistics.xml
.idea/**/dictionaries
.idea/**/shelf

# Generated files
.idea/**/contentModel.xml

# Sensitive or high-churn files
.idea/**/dataSources/
.idea/**/dataSources.ids
.idea/**/dataSources.local.xml
.idea/**/sqlDataSources.xml
.idea/**/dynamic.xml
.idea/**/uiDesigner.xml
.idea/**/dbnavigator.xml

# Gradle
.idea/**/gradle.xml
.idea/**/libraries

# Gradle and Maven with auto-import
#  When  using  Gradle  or  Maven  with  auto-import,  you  should  exclude  module
files,
# since they will be recreated, and may cause churn.  Uncomment if using
# auto-import.
# .idea/artifacts
# .idea/compiler.xml
# .idea/jarRepositories.xml
# .idea/modules.xml
# .idea/*.iml
# .idea/modules
# *.iml
# *.ipr

# CMake
cmake-build-*/

# Mongo Explorer plugin
.idea/**/mongoSettings.xml

# File-based project format
*.iws

# IntelliJ
out/

# mpeltonen/sbt-idea plugin
.idea_modules/

# JIRA plugin
atlassian-ide-plugin.xml

# Cursive Clojure plugin
```

```
.idea/replstate.xml

# Crashlytics plugin (for Android Studio and IntelliJ)
com_crashlytics_export_strings.xml
crashlytics.properties
crashlytics-build.properties
fabric.properties

# Editor-based Rest Client
.idea/httpRequests

# Android studio 3.1+ serialized cache file
.idea/caches/build_file_checksums.ser
```

Expressions that meet the following rules can be placed in the `.gitignore` file:

- blank lines or lines starting with `#` are ignored
- standard regular expressions
- Expressions followed by a sign (`/`) define entire directories
- expressions starting with (`!`) allow negation

Revision history

You can use the `git log` command to verify the history of changes that were made. This command returns a list of the changes committed to this repository in reverse chronological order. It shows the most recent changes first. By default, information such as this is presented:

- all changes along with their SHA-1 checksum
- the name and e-mail of the author
- the date of recording
- a change note.

```
Piotrs-MacBook-Pro:ExoPlayer root# git log
commit 7d3f54a375fac04a746ca76a8f2e1ad32c8b45b2 (HEAD -> release-v2, tag: r2.11.4, origin/release-v
2, origin/HEAD)
Merge: 49910fe72 76374d782
Author: Oliver Woodman <olly@google.com>
Date:   Wed Apr 8 22:48:19 2020 +0100

    Merge pull request #7162 from google/dev-v2-r2.11.4

    r2.11.4

commit 76374d78222c43860d676f5d1aacafa915d984be (origin/dev-v2-r2.11.4)
Author: Oliver Woodman <olly@google.com>
Date:   Wed Apr 8 22:11:33 2020 +0100

    Clean up playWhenReady

commit 54f3e6a2034ac8006eba0f5430b88fabfb3106ec
Author: Oliver Woodman <olly@google.com>
Date:   Wed Apr 8 22:03:06 2020 +0100

    Revert release note indentation change

commit f95a0caec3a2b56300da435b5f9fd72426686a07
Author: olly <olly@google.com>
Date:   Wed Apr 8 18:21:39 2020 +0100

    Update misc dependencies

    PiperOrigin-RevId: 305503804
```

As part of the `git log` command, you can specify additional parameters, e.g.

- -p
- --stat
- --pretty

One of the most useful options is `-p`. It shows the differences made with each revision. Limiting the number of entries is possible using e.g. `-2`, where the `-` character is followed by the number to which the result returned from the command is to be limited.

```
Piotrs-MacBook-Pro:ExoPlayer root# git log -p -2
commit 7d3f54a375fac04a746ca76a8f2e1ad32c8b45b2 (HEAD -> release-v2, tag: r2.11.4, origin/release-v
2, origin/HEAD)
Merge: 49910fe72 76374d782
Author: Oliver Woodman <olly@google.com>
Date:   Wed Apr 8 22:48:19 2020 +0100

    Merge pull request #7162 from google/dev-v2-r2.11.4

    r2.11.4

commit 76374d78222c43860d676f5d1aacafa915d984be (origin/dev-v2-r2.11.4)
Author: Oliver Woodman <olly@google.com>
Date:   Wed Apr 8 22:11:33 2020 +0100

    Clean up playWhenReady

diff --git a/library/core/src/main/java/com/google/android/exoplayer2/SimpleExoPlayer.java b/librar
y/core/src/main/java/com/google/android/exoplayer2/SimpleExoPlayer.java
index bb80d60f2..839c65b12 100644
--- a/library/core/src/main/java/com/google/android/exoplayer2/SimpleExoPlayer.java
+++ b/library/core/src/main/java/com/google/android/exoplayer2/SimpleExoPlayer.java
@@ -690,7 +690,7 @@ public class SimpleExoPlayer extends BasePlayer
     boolean playWhenReady = getPlayWhenReady();
     @AudioFocusManager.PlayerCommand
     int playerCommand = audioFocusManager.updateAudioFocus(playWhenReady, getPlaybackState());
-    updatePlayWhenReady(getPlayWhenReady(), playerCommand);
+    updatePlayWhenReady(playWhenReady, playerCommand);
   }
```

With the `--stat` option, it is possible to display each history entry details including:

- the list of modified files
- the number of files changed
- the number of lines added and removed

```
Piotrs-MacBook-Pro:ExoPlayer root# git log --stat -3
commit 7d3f54a375fac04a746ca76a8f2e1ad32c8b45b2 (HEAD -> release-v2, tag: r2.11.4, origin/release-v2, origin/HEAD)
Merge: 49910fe72 76374d782
Author: Oliver Woodman <olly@google.com>
Date:   Wed Apr 8 22:48:19 2020 +0100

    Merge pull request #7162 from google/dev-v2-r2.11.4

    r2.11.4

commit 76374d78222c43860d676f5d1aacafa915d984be (origin/dev-v2-r2.11.4)
Author: Oliver Woodman <olly@google.com>
Date:   Wed Apr 8 22:11:33 2020 +0100

    Clean up playWhenReady

 library/core/src/main/java/com/google/android/exoplayer2/SimpleExoPlayer.java | 7 ++++---
 1 file changed, 4 insertions(+), 3 deletions(-)

commit 54f3e6a2034ac8006eba0f5430b88fabfb3106ec
Author: Oliver Woodman <olly@google.com>
Date:   Wed Apr 8 22:03:06 2020 +0100

    Revert release note indentation change

 RELEASENOTES.md | 114 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++--------------------------------------------------
 1 file changed, 57 insertions(+), 57 deletions(-)
Piotrs-MacBook-Pro:ExoPlayer root#
```

The `--pretty` parameter allows you to format the output of the `git log` command in a new, non-default format. It is possible to use several pre-defined variants:

- the `oneline` option displays each committed change on a single line

```
Piotrs-MacBook-Pro:ExoPlayer root# git log --pretty=oneline
7d3f54a375fac04a746ca76a8f2e1ad32c8b45b2 (HEAD -> release-v2, tag: r2.11.4, origin/release-v2, origin/HEAD) Merge pull request #7162 from google/dev-v2-r2.11.4
76374d78222c43860d676f5d1aacafa915d984be (origin/dev-v2-r2.11.4) Clean up playWhenReady
54f3e6a2034ac8006eba0f5430b88fabfb3106ec Revert release note indentation change
f95a0caec3a2b56300da435b5f9fd72426686a07 Update misc dependencies
1f4d37431ec7d04f0f9c64514676752857d7e0de Upgrade cast dependency
55e33e36995dbcaea6f370f331f0e72d63441f2e Fix missing subtitle addition
f696a56b56e924a16a494e8e9788ed9178f203f1 Audio focus: Restore full volume if focus is abandoned when ducked
49858b82f24ea67373024c2ac71213ee44e2552b Merge pull request #7184 from TiVo:p-subtitle-format-from-codecs
cc29798d9b7dec65785c1a93a2a186352c92ff6f Audio focus: Re-request audio focus if in a transient loss state
3c0e617837d84cf7647db37afdbc0c2b9d47af65 Merge AudioFocusManager methods to simplify control flow.
07cbfb65e576e09805cec1b2dc0051ba6c0d0bc2 Fix stuck ad playbacks with DRM-protected content
d4d7907193385efae1f48e980b1f14c38c217283 Revert "Remove duplicate SCTE-35 format and add sample to TsExtractorTest"
29118f45869d1f9e61122943c4e06c9a0a8d7cb1 Tweak release note
f14c028078d8b2aa62cf8d5b3a45d398869193b5 Fix dump files for release
b8e6e98430917cd6c10b44810262bd7311aab958 Simplify `WakeLockManager` and `WifiLockManager` logic.
83c2ca59025604d7940806a3d26bf4614a4a2041 Add `WifiLock` management to `SimpleExoPlayer`.
e4e56fac094c4924d702b3e0b75cca293135bdb0 Clean `WakeLockManager.updateWakeLock` logic.
a5420a0b65169521e8f0491b1ac28edc236d7e10 Fix release notes
d921491e7d8c8d0772324a60938e345517251479 Remove thread checks in player constructor
38f282800c42591dfc5390491347a9a88a90fd38 Fix ADTS extraction with mid-stream ID3
d466b8c5d59893b13a3ddc50310d3fb3fcaac84e Make javadoc links point to Android docs for java.* classes
a9c4d2f9cff42d8a231ea12ff3379533ff0f75d1 Ensure javadoc fix applies to Android links with anchors
c75f3f77ffc9d5cf8c2973fa02d1c43071ede76e Merge pull request #7099 from matamegger:feature/fix_pssh_
```

- the `format` option allows you to specify your own appearance and format for the information displayed by the `git log` command

```
[Piotrs-MacBook-Pro:ExoPlayer root# git log --pretty=format:"%h - %an, %ar : %s"
7d3f54a37 - Oliver Woodman, 2 weeks ago : Merge pull request #7162 from google/dev-v2-r2.11.4
76374d782 - Oliver Woodman, 2 weeks ago : Clean up playWhenReady
54f3e6a20 - Oliver Woodman, 2 weeks ago : Revert release note indentation change
f95a0caec - olly, 2 weeks ago : Update misc dependencies
1f4d37431 - olly, 2 weeks ago : Upgrade cast dependency
55e33e369 - Oliver Woodman, 3 weeks ago : Fix missing subtitle addition
f696a56b5 - olly, 3 weeks ago : Audio focus: Restore full volume if focus is abandoned when ducked
49858b82f - Oliver Woodman, 3 weeks ago : Merge pull request #7184 from TiVo:p-subtitle-format-from
-codecs
cc29798d9 - olly, 3 weeks ago : Audio focus: Re-request audio focus if in a transient loss state
3c0e61783 - tonihei, 7 weeks ago : Merge AudioFocusManager methods to simplify control flow.
07cbfb65e - andrewlewis, 3 weeks ago : Fix stuck ad playbacks with DRM-protected content
d4d790719 - Oliver Woodman, 3 weeks ago : Revert "Remove duplicate SCTE-35 format and add sample to
 TsExtractorTest"
29118f458 - Oliver Woodman, 3 weeks ago : Tweak release note
f14c02807 - Oliver Woodman, 3 weeks ago : Fix dump files for release
b8e6e9843 - samrobinson, 7 weeks ago : Simplify `WakeLockManager` and `WifiLockManager` logic.
83c2ca590 - samrobinson, 8 weeks ago : Add `WifiLock` management to `SimpleExoPlayer`.
e4e56fac0 - samrobinson, 5 months ago : Clean `WakeLockManager.updateWakeLock` logic.
a5420a0b6 - olly, 3 weeks ago : Fix release notes
d921491e7 - kimvde, 3 weeks ago : Remove thread checks in player constructor
38f282800 - olly, 3 weeks ago : Fix ADTS extraction with mid-stream ID3
d466b8c5d - ibaker, 4 weeks ago : Make javadoc links point to Android docs for java.* classes
a9c4d2f9c - ibaker, 4 weeks ago : Ensure javadoc fix applies to Android links with anchors
c75f3f77f - Oliver Woodman, 4 weeks ago : Merge pull request #7099 from matamegger:feature/fix_pssh
_v1_on_firetv
1f44a4db4 - olly, 4 weeks ago : Bump version to 2.11.4
fea0acd41 - andrewlewis, 4 weeks ago : Fix PlaybackStatsListener behavior when not keeping history
94ca84ff2 - jaewan, 4 weeks ago : Allow developers to specify CharSequence for Notification strings
```

Below are a set of useful options for formatting logs:

Option	Description
%H	commit's hash
%h	abbreviated commit's hash
%T	tree hash
%t	abbreviated tree hash
%P	parent commit's hash
%p	abbreviated parent commit's hash
%an	author's name
%ae	author's email
%ad	author's date (refers to --date=option
%ar	author's date, relative
%cn	approver's name
%ce	approver's email
%cd	approver's date
%cr	approver's date, relative
%s	messages

Reverting changes

In case of erroneous or too hasty execution of the revision (other scenarios can include: omitting of some files, or an error in the message being committed) it is possible to execute the `git commit` command with the `--amend` option, which corrects the commit.

```
Piotrs-MacBook-Pro:git_example root# git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   readme.md
        modified:   todo.txt

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   help.txt

Piotrs-MacBook-Pro:git_example root# git commit -m "modify todo list"
[master 46f6a5a] modify todo list
 2 files changed, 5 insertions(+), 3 deletions(-)
Piotrs-MacBook-Pro:git_example root# git add help.txt
Piotrs-MacBook-Pro:git_example root# git commit --amend
[master 9d9bbab] modify todo list
 Date: Sun Apr 26 20:16:50 2020 +0200
 3 files changed, 6 insertions(+), 3 deletions(-)
Piotrs-MacBook-Pro:git_example root# git status
On branch master
nothing to commit, working tree clean
Piotrs-MacBook-Pro:git_example root#
```

In the example above, only two files were validated: `todo.txt` and `readme.md`. The file `help.txt` was not included by mistake. By running the command `git commit --amend`, you can modify the committed changes. Before calling the command, the `help.txt` file was added to the waiting list. The `git commit --amend` command opens a text editor where you can modify the message and the commit itself:

```
  GNU nano 2.0.6   File: /Users/pbrzozowski/Documents/git_example/.git/COMMIT_EDITMSG

modify todo list

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# Date:      Sun Apr 26 20:16:50 2020 +0200
#
# On branch master
# Changes to be committed:
#       modified:   help.txt
#       modified:   readme.md
#       modified:   todo.txt
#

                              [ Read 13 lines ]
^G Get Help    ^O WriteOut    ^R Read File   ^Y Prev Page   ^K Cut Text    ^C Cur Pos
^X Exit        ^J Justify     ^W Where Is    ^V Next Page   ^U UnCut Text  ^T To Spell
```

After approving the fixes, the last commit has been modified with the file `help.txt`.

Removing files from the lobby

In order to remove a file from the waiting room, you can use the command:

```
git reset HEAD <file> ..
```

where `<file>` is the file to be removed from the waiting area. This command does not undo changes to a modified file, it only removes files from the waiting room.

```
Piotrs-MacBook-Pro:git_example root# git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   readme.md
        modified:   todo.txt

no changes added to commit (use "git add" and/or "git commit -a")
Piotrs-MacBook-Pro:git_example root# git add --all
Piotrs-MacBook-Pro:git_example root# git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   readme.md
        modified:   todo.txt

Piotrs-MacBook-Pro:git_example root# git reset HEAD readme.md
Unstaged changes after reset:
M       readme.md
Piotrs-MacBook-Pro:git_example root# git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   todo.txt

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   readme.md

Piotrs-MacBook-Pro:git_example root#
```

In the example above, all files were added to the waiting list by mistake. They are also waiitng for approval. The file `readme.md` should not be committed. To remove it from the standby, the command `git reset HEAD readme.md` was executed.

Undoing changes to the modified file

In the case of modified files, it is possible to restore them to the state from the last commit. This is possible by executing:

```
git checkout -- <file> ..
```

where `<file>` is the name of the file we want to restore to the state it was before modification.

```
[Piotrs-MacBook-Pro:git_example root# git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   todo.txt

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   readme.md

[Piotrs-MacBook-Pro:git_example root# git checkout -- readme.md
[Piotrs-MacBook-Pro:git_example root# git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   todo.txt

Piotrs-MacBook-Pro:git_example root#
```

In the example above, the `readme.md` file has been set back to the state it was in before any changes with the command `git checkout - readme.md` were made. This command is quite dangerous: any changes made to the file are lost after the file has been executed and cannot be restored using the git version control system.

Tags

Git implements the possibility of labeling specific, important places, e.g. releases, or production versions of the application.

Displaying tags

In order to display all available tags, it is possible to use the command `git tag`:

```
[Piotrs-MacBook-Pro:ExoPlayer root# git tag
r1.0.10
r1.0.10_docs
r1.0.11
r1.0.11_docs
r1.0.12
r1.0.12_docs
r1.0.13
r1.0.13_docs
r1.2.1
r1.2.1_docs
r1.2.2
r1.2.2_docs
r1.2.3
r1.2.3_docs
r1.2.4
r1.2.4_docs
r1.3.1
r1.3.1_docs
r1.3.2
r1.3.2_docs
r1.3.3
r1.3.3_docs
r1.4.0
r1.4.0_docs
r1.4.1
r1.4.1_docs
r1.4.2
r1.4.2_docs
r1.5.0
r1.5.0_docs
r1.5.1
```

The `-l` attribute allows you to define a pattern according to which you can search for a specific group of tags:

```
Piotrs-MacBook-Pro:ExoPlayer root# git tag -l r2.*
r2.0.0
r2.0.0_docs
r2.0.1
r2.0.1_docs
r2.0.2
r2.0.2_docs
r2.0.3
r2.0.3_docs
r2.0.4
r2.0.4_docs
r2.1.0
r2.1.0_docs
r2.1.1
r2.1.1_docs
r2.10.0
r2.10.0_docs
r2.10.1
r2.10.1_docs
r2.10.2
r2.10.2_docs
r2.10.3
r2.10.3_docs
r2.10.4
r2.10.4_docs
r2.10.5
r2.10.5_docs
r2.10.6
r2.10.6_docs
r2.10.7
r2.10.7_docs
r2.10.8
```

Creating tags

In order to create labels, you can use the command:

```
git tag [label-name]
```

```
Piotrs-MacBook-Pro:ExoPlayer root# git tag v3.0.1
Piotrs-MacBook-Pro:ExoPlayer root# git tag -l v3.*
v3.0.1
Piotrs-MacBook-Pro:ExoPlayer root#
```

Tag sharing

By default, the `git push` command does not push labels to the remote repository. In order to send them to the server, follow the same process as for transferring the branches:

```
git push origin [label-name].
```

The result of the call will be as below:

```
$ git push origin v3.0.1
Counting objects: 25, done.
```

```
Delta compression using up to 8 threads.
Compressing objects: 100% (12/12), done.
Writing objects: 100% (25/25), 4.34 KiB | 0 bytes/s, done.
Total 25 (delta 3), reused 0 (delta 0)
To https://github.com/google/ExoPlayer.git
 * [new tag]         v3.0.1 -> v3.0.1
```

Aliases

Git also allows you to set aliases. They are shortcuts that will correspond to the original version of the commands. Aliases can be set in the configuration file using the format `alias.*`, Where `*` is any abbreviated version of the command:

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

Thanks to aliases the following abbreviated commands can be used:

```
Piotrs-MacBook-Pro:git_example root# git st
On branch master
nothing to commit, working tree clean
Piotrs-MacBook-Pro:git_example root# git br
  conflict_branch
* master
Piotrs-MacBook-Pro:git_example root#
```

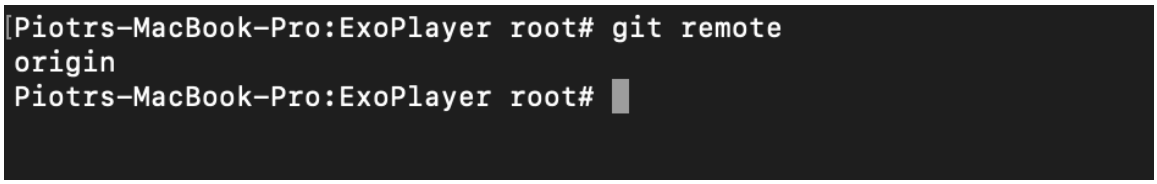
Remote repository

A remote repository is a version of the project maintained on a server which is accessible over the network. Collaboration in the project is based on managing remote repositories, sending changes, and downloading them for work or code sharing. Managing remote repositories includes:

- adding remote repositories
- removing remote repositories
- management of remote branches
- defining remote branches as those that are tracked or those that are not

Display remote repositories

The command `git remote` allows you to list all configured servers. In the case of a cloned repository, at least the `origin` server should be visible. This name is the default name given to the server by Git.



The optional `-v` parameter allows to additionally display the full, VCS-visible URL assigned to the shortcut:

Adding remote repositories

Adding remote repositories is possible with the command `git remote add [shortcut] [url]`:

```
$ git remote
origin
$ git remote add sda https://github.com/sda/vcs_git
$ git remote -v
origin  https://github.com/sda/vcs_git (fetch)
origin  https://github.com/sda/vcs_git (push)
sda https://github.com/sda/vcs_git (fetch)
sda https://github.com/sda/vcs_git (push)
```

In the example above, it is possible to use the sda name instead of the entire URL. For example, to update the local version of the repository instead of pointing to a specific url, you can use the command `git fetch pb`.

Download and merge changes from remote repositories

The function of retrieving data from a remote repository is possible with the command:

```
$ git fetch [remote-name]
```

Execution of the above command allows you to retrieve information on all remote branches. In addition, the `fetch` command downloads data to the local repository. However, it does not automatically merge changes with any of the working files, nor does it modify these files in any other way.

If you need to download changes as well as automatically merge them, you can use the `git pull` command. This command does the following:

- data download (`fetch`)
- data merge with local files (`merge`)

The `git clone` command allows the local master branch to track changes to the remote master branch on the server from which the repository was cloned (assuming the remote repository has a master branch). Running the `git pull` command allows you to retrieve data from the server that was the source. In the next step an attempt is made to automatically merge the changes with the working code where work is currently being performed locally.

Pushing changes to a remote repository

The command `git push [remote-repo-name] [branch-name]` can be used to send committed changes to the server. For example, to send the master branch to the original `origin` source server, you could run the following command:

```
git push origin master
```

If two people work on the same branch and one person sends changes to the server, the next person who wants to share their changes will be rejected. You will need to download and merge the changes from the remote repository first.

> ***NOTE:*** If you work with repository on GitHub, after executing `git push` command for the first time, you will be asked for password. Instead of providing your GitHub account password, you should paste your `Personal Access Token`. Here is a detailed instruction how to generate one.

Rename and remove changes from remote repository

In order to rename a shortcut assigned to the repository, use the command `git remote rename`, e.g.

```
git remote rename sda sda_course
```

The above command allows you to change the shortcut of the `sda` repository to `sda_course`.

If you need to remove a link, you can use the command `git remote rm`, e.g.

```
$ git remote -v
origin  https://github.com/sda/vcs_git (fetch)
origin  https://github.com/sda/vcs_git (push)
sda_course  https://github.com/sda/vcs_git (fetch)
sda_course  https://github.com/sda/vcs_git (push)

$ git remote rm sda_course
$ git remote -v
```

```
origin  https://github.com/sda/vcs_git (fetch)
origin  https://github.com/sda/vcs_git (push)
```

The above example allows you to remove all references to the remote `sda_course` repository.

Overview of remote repositories

For more information about the remote repository, use the command `git remote show [remote-repo-name]`. By running it with a specific shortcut, such as `origin`, you will see the following result:

```
[Piotrs-MacBook-Pro:ExoPlayer root# git remote show origin
* remote origin
  Fetch URL: https://github.com/google/ExoPlayer.git
  Push  URL: https://github.com/google/ExoPlayer.git
  HEAD branch: release-v2
  Remote branches:
    dev-v1                                tracked
    dev-v2                                tracked
    dev-v2-cea                            tracked
    experiment-dash-lowlatency            tracked
    experiment-dummysurface-toggle        tracked
    experiment-subtitle-webview           tracked
    gh-pages                              tracked
    io18                                  tracked
    release-v1                            tracked
    release-v2                            tracked
    release-v2-surfacetexture-experiment tracked
  Local branch configured for 'git pull':
    release-v2 merges with remote release-v2
  Local ref configured for 'git push':
    release-v2 pushes to release-v2 (up to date)
Piotrs-MacBook-Pro:ExoPlayer root#
```

The above information includes:

- the URL of the remote repository
- some information about the tracked branch
- a list of all downloaded remote links
- the command also says that if the user is in the release_v2 branch and the `git pull` command is run, changes from the remote repository will be automatically merged with the release_v2 branch in the local repository after download

Branches in Git

Almost every version control system is based on branch support. Branching can be understood from the perspective of software development. It allows the user to continue making changes without making a mess. It is a very costly process for most version control systems. For the Git version control system it is one of its most important features. The way Git handles branches is very light and efficient. The user is able to makie new branches almost instantaneously however switching between branches takes a little longer - this is unlike the solutions presented for many other systems.

Branches in Git are simple files, containing the 40 SHA-1 checksum characters from the changelog they point to. They are very cheap to create and remove. Creating a new branch takes exactly the amount of time equivalent to writing 41 bytes to a file (40 characters + newline).

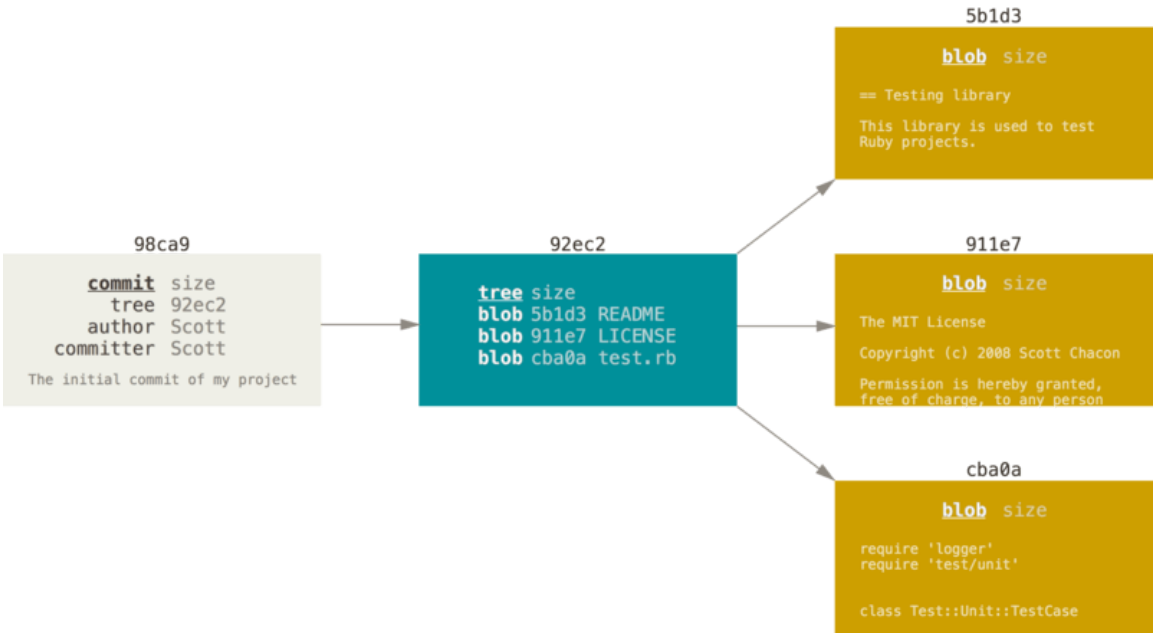
Approval of changes

When the files are transferred to the waiting list, the following occurs:

- calculating the checksum (SHA-1 shortcut) for each file
- saving file versions in the repository (so-called blobs)
- adding a checksum to the waiting list

When you commit your changes by running `git commit`, Git checks the sum of each subdirectory and writes these objects to the repository. Next a commit object is created containing:

- metadata
- a pointer to the main project tree, which will allow you to recreate the entire snapshot if necessary



The above schema presents the situation created due to the following changes:

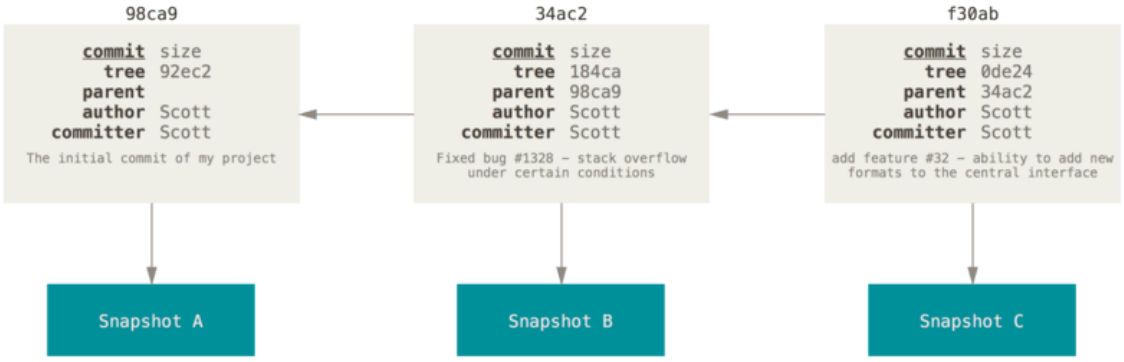
```
$ git add README test.rb LICENSE
$ git commit -m 'initial commit of my project'
```

A sample Git repository after performing this operation should include:

- one `blob` for the contents of each of the three files
- one tree that describes the contents of the directory and the files that are stored in which blobs
- one set of changes with a pointer to the tree and all metadata

Branch

A branch in Git can be described as a light, sliding indicator on one of the sets of changes. The default Git branch name is usually `master` or `main` (but this name is not reserved, in can have any name we want). When the first changes are committed, a master branch is obtained which points to the last committed set of changes. The branch automatically moves forward with each commit.



> ***Note:*** In this section in all examples we use `master` for default branch name. Nowadays, `main` is often used as default branch name, but remember that is can have any name.

Creating a new branch

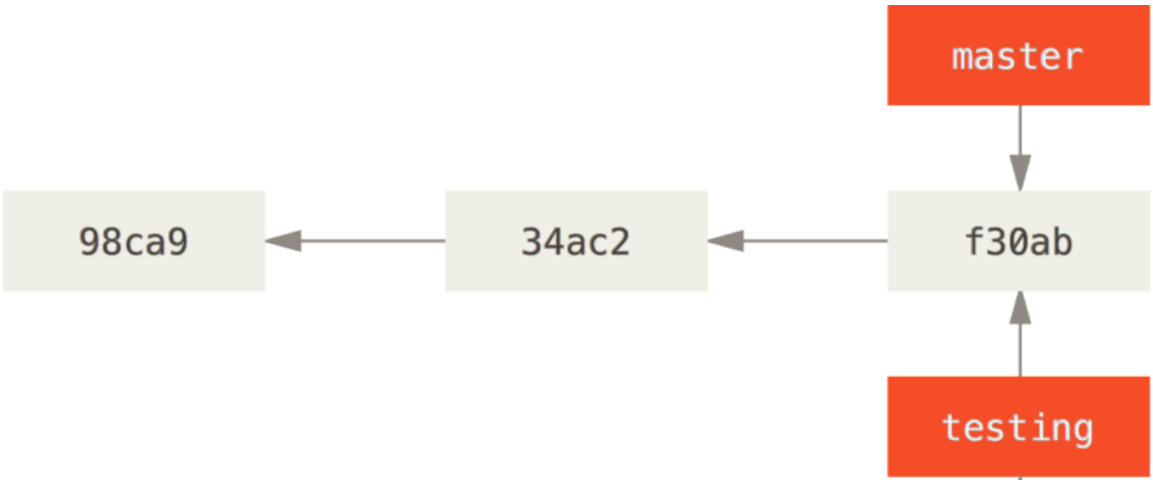
Creating a new branch is performed using the command:

```
git branch branch_name
```

When invoking the command:

```
git branch testing
```

a new pointer to the current set of changes is created:



Alternatively, a new branch can be created with the command:

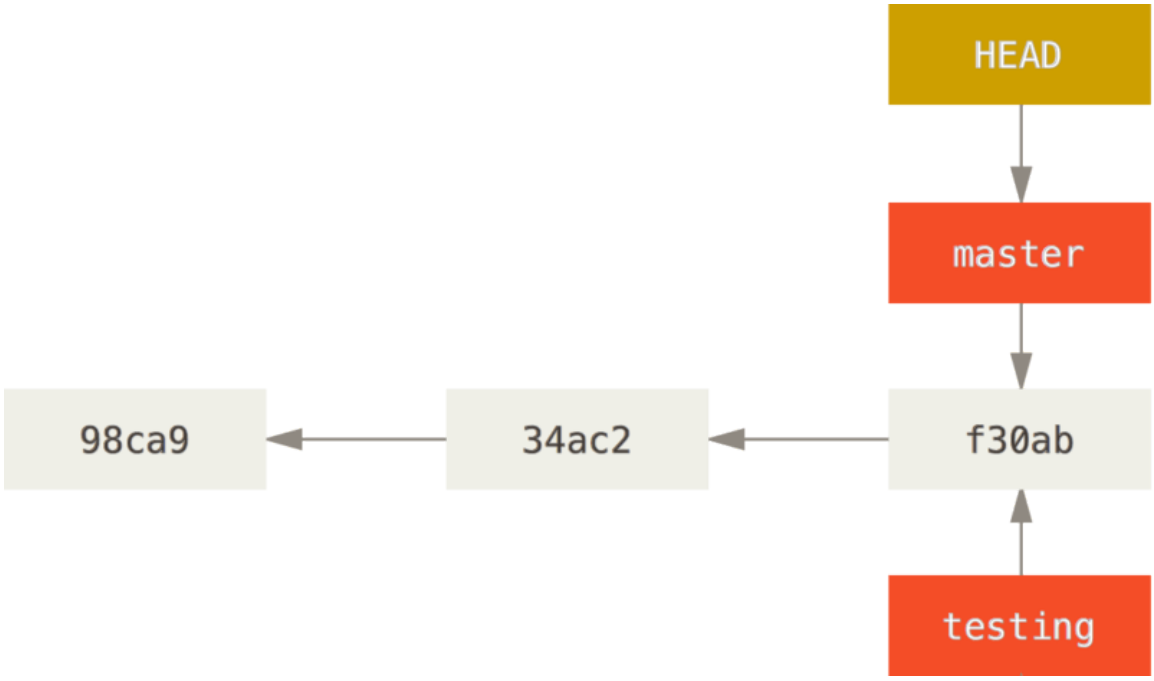
```
git checkout -b branch_name
```

```
Piotrs-MacBook-Pro:ExoPlayer root# git checkout -b testing
Switched to a new branch 'testing'
Piotrs-MacBook-Pro:ExoPlayer root# git branch
  release-v2
* testing
Piotrs-MacBook-Pro:ExoPlayer root#
```

The above example shows how the `git checkout -b` command works, which creates a new change pointer, and moves the `HEAD` pointer to the branch you created.

HEAD pointer

Git keeps track of the branch a user is currently residing on using the `HEAD` pointer. It is a pointer to the local branch the user is currently working on.



In the above case, after calling the command:

```
git branch testing
```

The user is still on the master branch. The command `git branch` creates a new branch, there is no switching invloved.

The location of the `HEAD` pointer can be easily checked with the `git log` command with the `--decorate` option, e.g.

```
git log --oneline --decorate
```

```
Piotrs-MacBook-Pro:ExoPlayer root# git log --oneline --decorate
7d3f54a37 (HEAD -> release-v2, tag: r2.11.4, origin/release-v2, origin/HEAD) Merge pull request #7162 fr
om google/dev-v2-r2.11.4
76374d782 Clean up playWhenReady
54f3e6a20 Revert release note indentation change
f95a0caec Update misc dependencies
1f4d37431 Upgrade cast dependency
55e33e369 Fix missing subtitle addition
f696a56b5 Audio focus: Restore full volume if focus is abandoned when ducked
49858b82f Merge pull request #7184 from TiVo:p-subtitle-format-from-codecs
cc29798d9 Audio focus: Re-request audio focus if in a transient loss state
3c0e61783 Merge AudioFocusManager methods to simplify control flow.
07cbfb65e Fix stuck ad playbacks with DRM-protected content
d4d790719 Revert "Remove duplicate SCTE-35 format and add sample to TsExtractorTest"
29118f458 Tweak release note
f14c02807 Fix dump files for release
b8e6e9843 Simplify `WakeLockManager` and `WifiLockManager` logic.
83c2ca590 Add `WifiLock` management to `SimpleExoPlayer`.
e4e56fac0 Clean `WakeLockManager.updateWakeLock` logic.
a5420a0b6 Fix release notes
d921491e7 Remove thread checks in player constructor
38f282800 Fix ADTS extraction with mid-stream ID3
d466b8c5d Make javadoc links point to Android docs for java.* classes
a9c4d2f9c Ensure javadoc fix applies to Android links with anchors
c75f3f77f Merge pull request #7099 from matamegger:feature/fix_pssh_v1_on_firetv
1f44a4db4 Bump version to 2.11.4
fea0acd41 Fix PlaybackStatsListener behavior when not keeping history
94ca84ff2 Allow developers to specify CharSequence for Notification strings
f0e734d33 Workaround C2 AAC decoder flush problem on Android 10
69ca53493 Parse opus gain correctly as a signed value
a9cbbf91c Skip aliases of other codecs
0a5a3cbf4 Handle orientation changes in GL demo
```

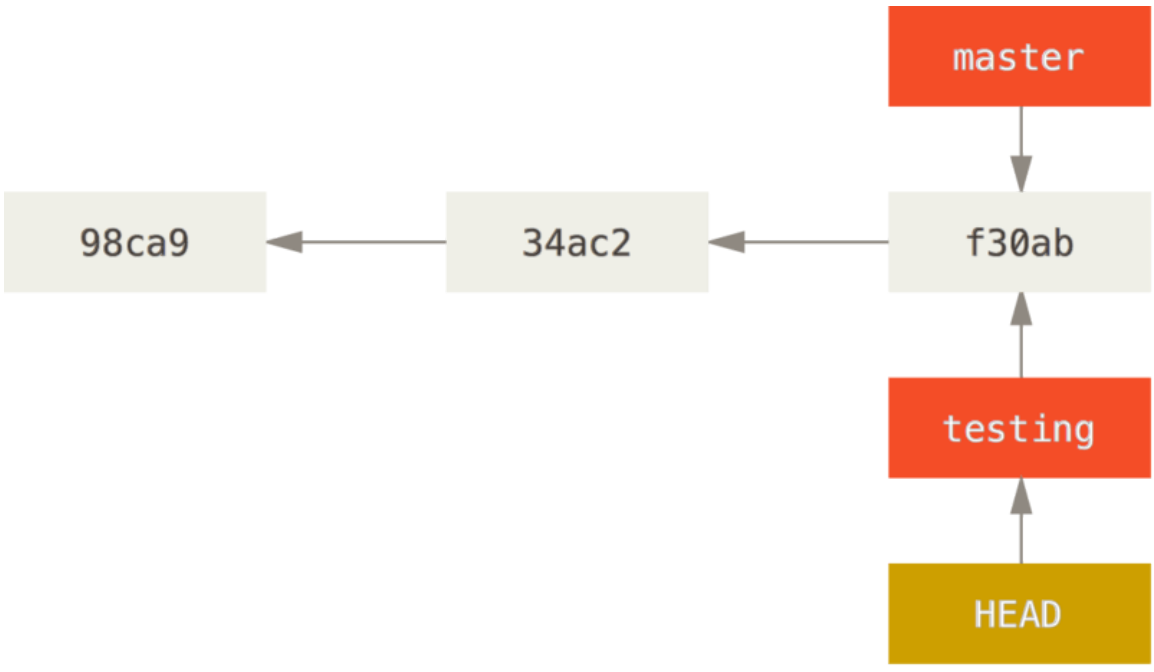
Switching branches

In order to switch to an existing branch, use the command:

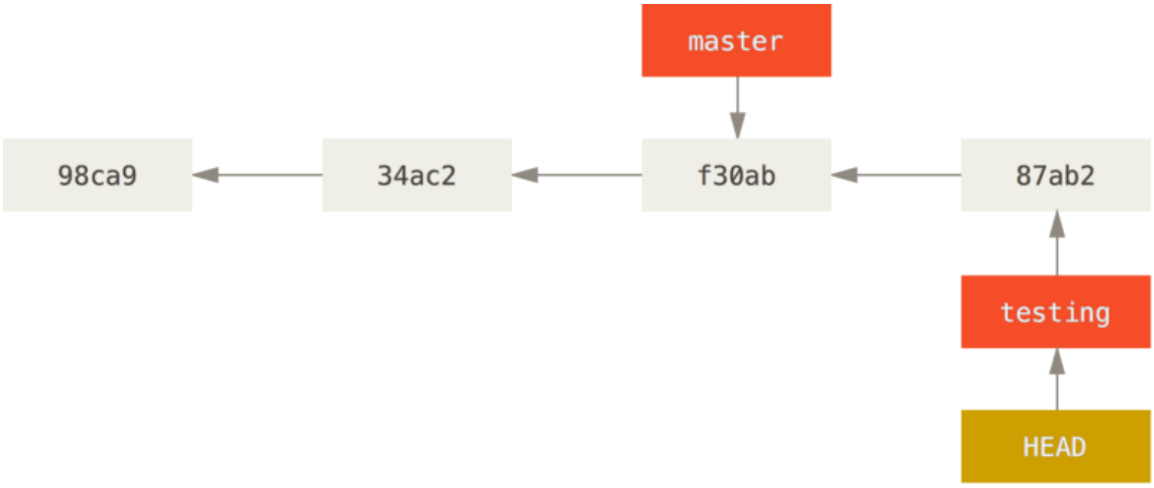
```
git checkout branch_id
```

In the case above, the HEAD pointer is moved to point to the `testing` branch:

```
git checkout testing
```



When you commit any changes to the `testing` branch, the `HEAD` indicator moves forward and points to the committed changes in sequence, as shown in the diagram below:

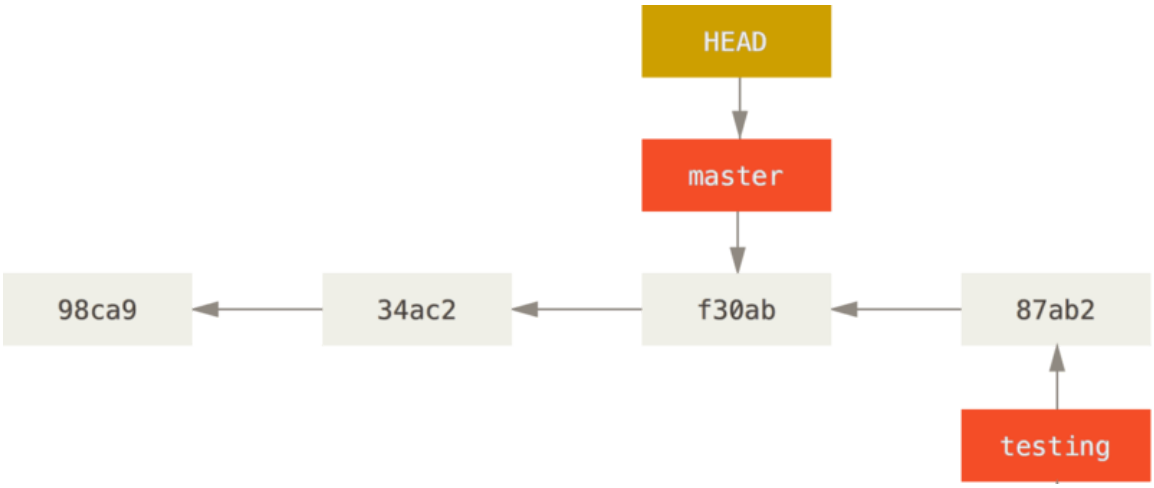


Based on the above flowchart, it can be concluded that the `testing` branch comes before the `master` branch. When invoking the command:

```
git checkout master
```

The above command performs two actions:

- it moves the `HEAD` pointer back to the master branch
- it restores the files in the working directory to the state in the snapshot as pointed to by master



Branch management

local branches

The command `git branch` (with no additional arguments) allows you to show all local branches that are part of the version control system:

```
Piotrs-MacBook-Pro:ExoPlayer root# git branch
* release-v2
Piotrs-MacBook-Pro:ExoPlayer root#
```

The symbol `*` defines the branch that the pointer `HEAD` refers to.

The above command coupled with an additional `-v` attribute allows you to extend the scope of information provided for the current revisions on individual branches:

```
Piotrs-MacBook-Pro:ExoPlayer root# git branch -v
* release-v2 7d3f54a37 Merge pull request #7162 from google/dev-v2-r2.11.4
Piotrs-MacBook-Pro:ExoPlayer root#
```

remote branches

The optional `-r` attribute allows you to display all remote branches that are in the Git version control system, as shown below:

```
Piotrs-MacBook-Pro:ExoPlayer root# git branch -r
  origin/HEAD -> origin/release-v2
  origin/dev-v1
  origin/dev-v2
  origin/dev-v2-cea
  origin/experiment-dash-lowlatency
  origin/experiment-dummysurface-toggle
  origin/experiment-subtitle-webview
  origin/gh-pages
  origin/io18
  origin/release-v1
  origin/release-v2
  origin/release-v2-surfacetexture-experiment
Piotrs-MacBook-Pro:ExoPlayer root#
```

--merged and --no-merged

The `--merged` and `--no-merged` options allow you to filter for the branches that have already been or have not yet been merged into the active branch.

- `--merged`

```
Piotrs-MacBook-Pro:ExoPlayer root# git branch -r --merged
  origin/HEAD -> origin/release-v2
  origin/dev-v2-cea
  origin/release-v2
Piotrs-MacBook-Pro:ExoPlayer root#
```

- `--no-merged`

```
Piotrs-MacBook-Pro:ExoPlayer root# git branch -r --no-merged
  origin/dev-v1
  origin/dev-v2
  origin/experiment-dash-lowlatency
  origin/experiment-dummysurface-toggle
  origin/experiment-subtitle-webview
  origin/gh-pages
  origin/io18
  origin/release-v1
  origin/release-v2-surfacetexture-experiment
Piotrs-MacBook-Pro:ExoPlayer root#
```

Removal of a branch

Removing a branch is possible with the command:

```
git branch -d branch_name
```

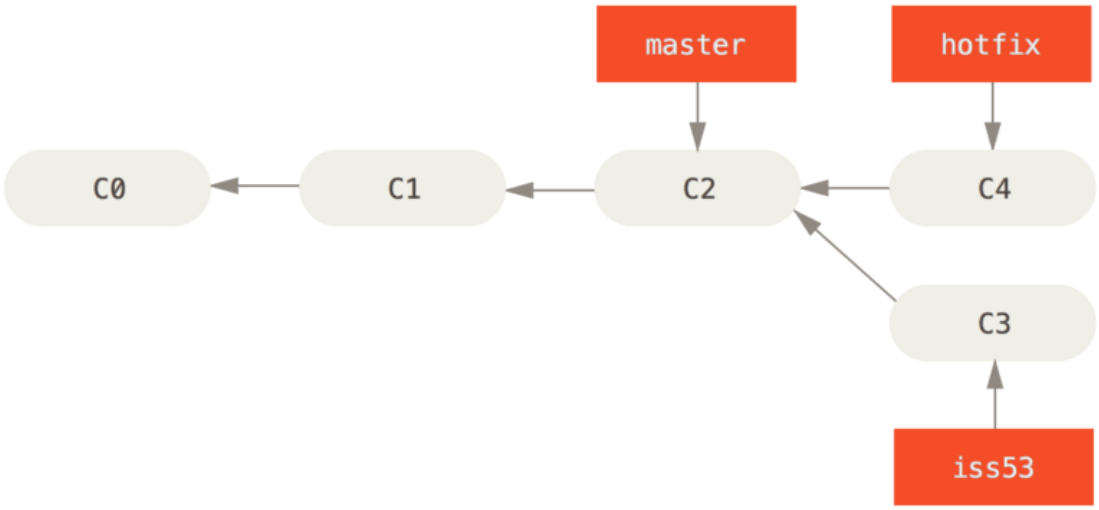
```
Piotrs-MacBook-Pro:ExoPlayer root# git branch
* release-v2
  testing
Piotrs-MacBook-Pro:ExoPlayer root# git branch -d testing
Deleted branch testing (was 7d3f54a37).
Piotrs-MacBook-Pro:ExoPlayer root# git branch
* release-v2
Piotrs-MacBook-Pro:ExoPlayer root#
```

Merging branches

Branching is done with the `git merge` command. This command allows you to integrate changes from one branch into the target branch. This command should be run from the branch to which we want to merge the changes.

```
git merge testing
```

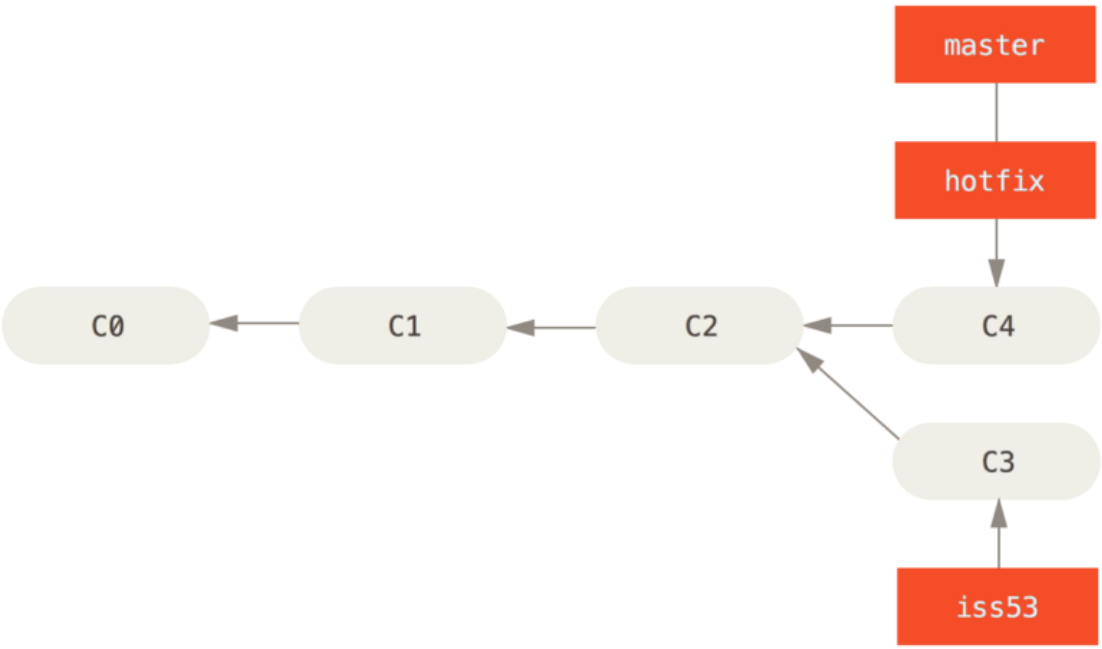
Fast Forward merge



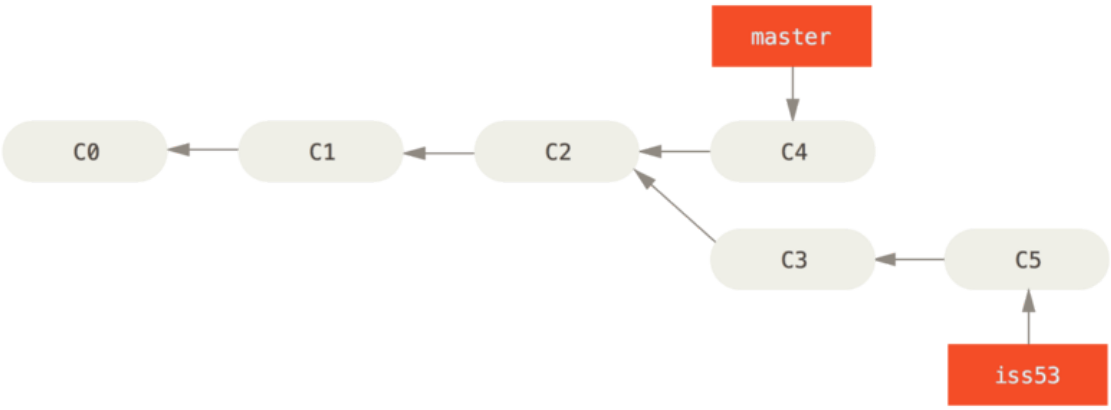
In the scenario above while working on the `iss53` functionality, there was a need to patch changes to the `master` branch. For this purpose, a new `hotfix` branch has been created, on it the `master` corrections have been added. At the time of calling the command:

```
git checkout master
git merge hotfix
```

The above merge is a fast forward merge. The changes indicated by the branch to be merged became the direct parent of the current set of changes. Git moves the pointer forward in this scenario. At the moment of trying to merge a set of changes with othe revisions that can be indicated by following the history of the first pointer. Git simplifies the process by moving the pointer forward as there are no forks to merge along the way - hence the name `fast forward`.



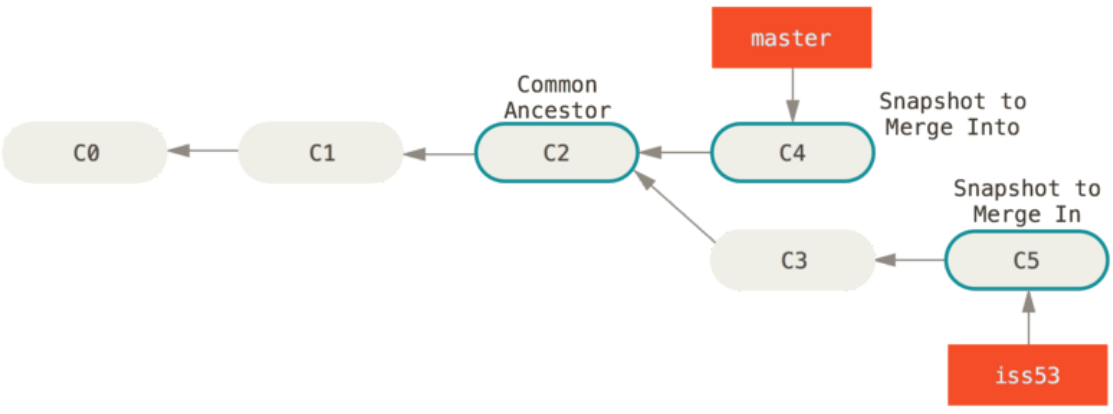
Three-way merge



In the scenario above while working on the `iss53` functionality, some modifications were made to the `master` branch in parallel. This resulted in the splitting of these branches.

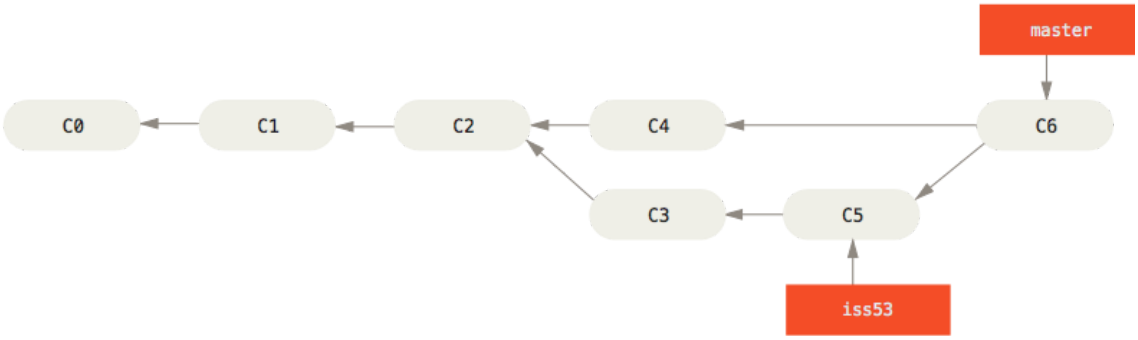
As a result of executing the following command, the merge will be performed differently than in the `fast forward` strategy:

```
git checkout master
git merge iss53
```



In the case showcased, the history of development was split at an earlier stage. Since the `master` changelog is not a direct descendant of the `iss53` merge branch, Git performs a three-way merge using the two snapshots pointed to by the branch endings and their common ancestor.

Git creates a new snapshot that is the result of a tripartite concatenation, and automatically creates a new changelog pointing to the snapshot. This is known as a merge commit that has more than one parent.



Resolving the conflict

If the same file has differentchangs in both branches being merged, Git won't be able to merge it on its own.

```
[Piotrs-MacBook-Pro:git_example root# git branch
  conflict_branch
* master
[Piotrs-MacBook-Pro:git_example root# git merge conflict_branch
Auto-merging todo.txt
CONFLICT (content): Merge conflict in todo.txt
Automatic merge failed; fix conflicts and then commit the result.
Piotrs-MacBook-Pro:git_example root#
```

In the example above, the user tried to merge the branch `conflict_branch` into the branch `master`. Here the `todo.txt` file was modified on both branches, hence a conflict message appeared.

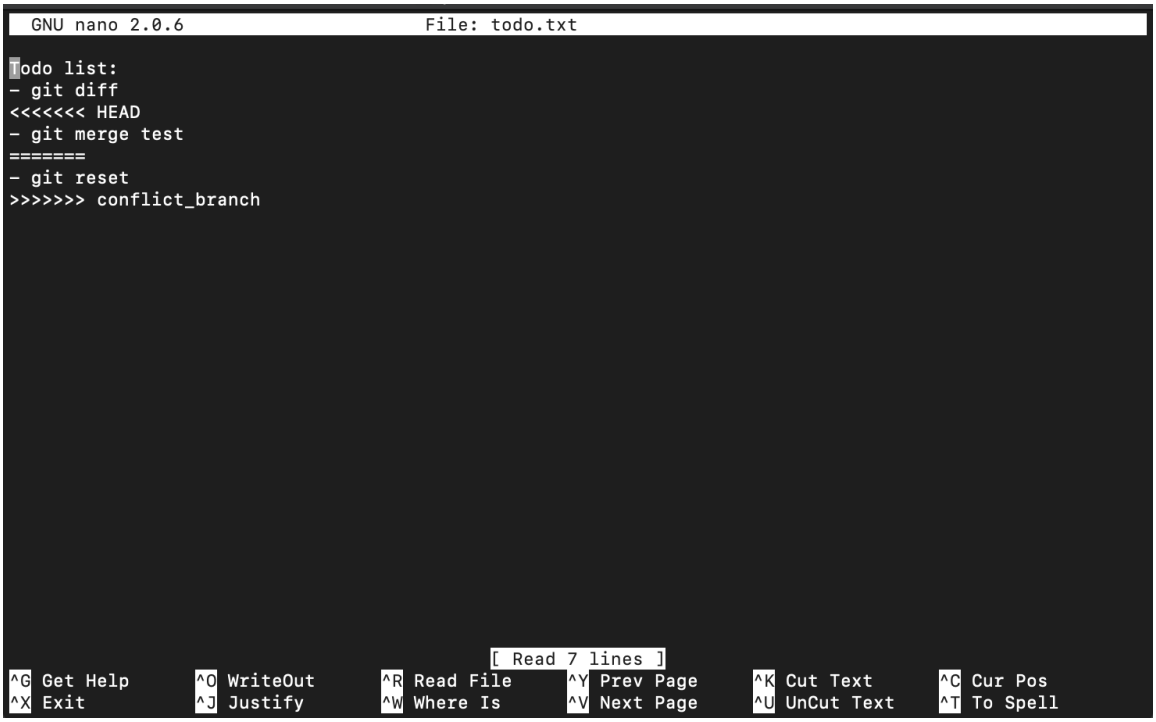
Git didn't automatically commit the change to be merged. It suspended the entire process until the conflict can be resolved. The state of the files can be verified with the command `git status`.

```
Piotrs-MacBook-Pro:git_example root# git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)
        both modified:   todo.txt

no changes added to commit (use "git add" and/or "git commit -a")
Piotrs-MacBook-Pro:git_example root#
```

Any conflicting files that are not automatically resolved are listed as "unmerged". For such changes, Git adds standard conflict resolution markup to problematic files. This allows the user to open these files without any problems and then resolve the conflicts manually. As a result of opening the file the following tag will be added:



- everything below the `<<<<<<< HEAD` specifies all changes that were in the file at the time of merging the branches
- everything below the `=======` section is all the changes that were made to the branch that was merged

If the user wants to resolve conflicts with a graphical tool, he/she can use the command `git mergetool`:

```
[Piotrs-MacBook-Pro:git_example root# git mergetool

This message is displayed because 'merge.tool' is not configured.
See 'git mergetool --tool-help' or 'git help config' for more details.
'git mergetool' will now attempt to use one of the following tools:
opendiff tortoisemerge emerge vimdiff
Merging:
todo.txt

Normal merge conflict for 'todo.txt':
  {local}: modified file
  {remote}: modified file
Hit return to start merge resolution tool (opendiff):
```

The above command allows for the selection of the conflict utility, in this case `opendiff`.

After making all the necessary corrections and adding changes to the waiting list, the user can finally approve the changes by using the `git commit` command, which allows for the creation of `merge commit`:

```
  GNU nano 2.0.6     File: /Users/pbrzozowski/Documents/git_example/.git/COMMIT_EDITMSG

Merge branch 'conflict_branch'

# Conflicts:
#       todo.txt
#
# It looks like you may be committing a merge.
# If this is not correct, please remove the file
#       .git/MERGE_HEAD
# and try again.


# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch master
# All conflicts fixed but you are still merging.
#
# Changes to be committed:
#       modified:   todo.txt
#

                                [ Read 20 lines ]
^G Get Help      ^O WriteOut      ^R Read File     ^Y Prev Page     ^K Cut Text      ^C Cur Pos
^X Exit          ^J Justify       ^W Where Is      ^V Next Page     ^U UnCut Text    ^T To Spell
```