Version control system

git

Andrii Pavelchak, 2018

‹epam›

When you work on a development team, you may be touching similar parts of the code throughout a project. As a result, changes made in one part of the source can be incompatible with those made by another developer working at the same time.

Version control helps solve these problems and provides:

- A complete history of every file, which enables you to go back to previous versions to analyze the source of bugs and fix problems in older versions.
- The ability to work on independent streams of changes, which allows you to merge that work back together and verify that your changes conflict.
- The ability to trace each change with a message describing the purpose and intent of the change and connect it to project management and bug tracking software.

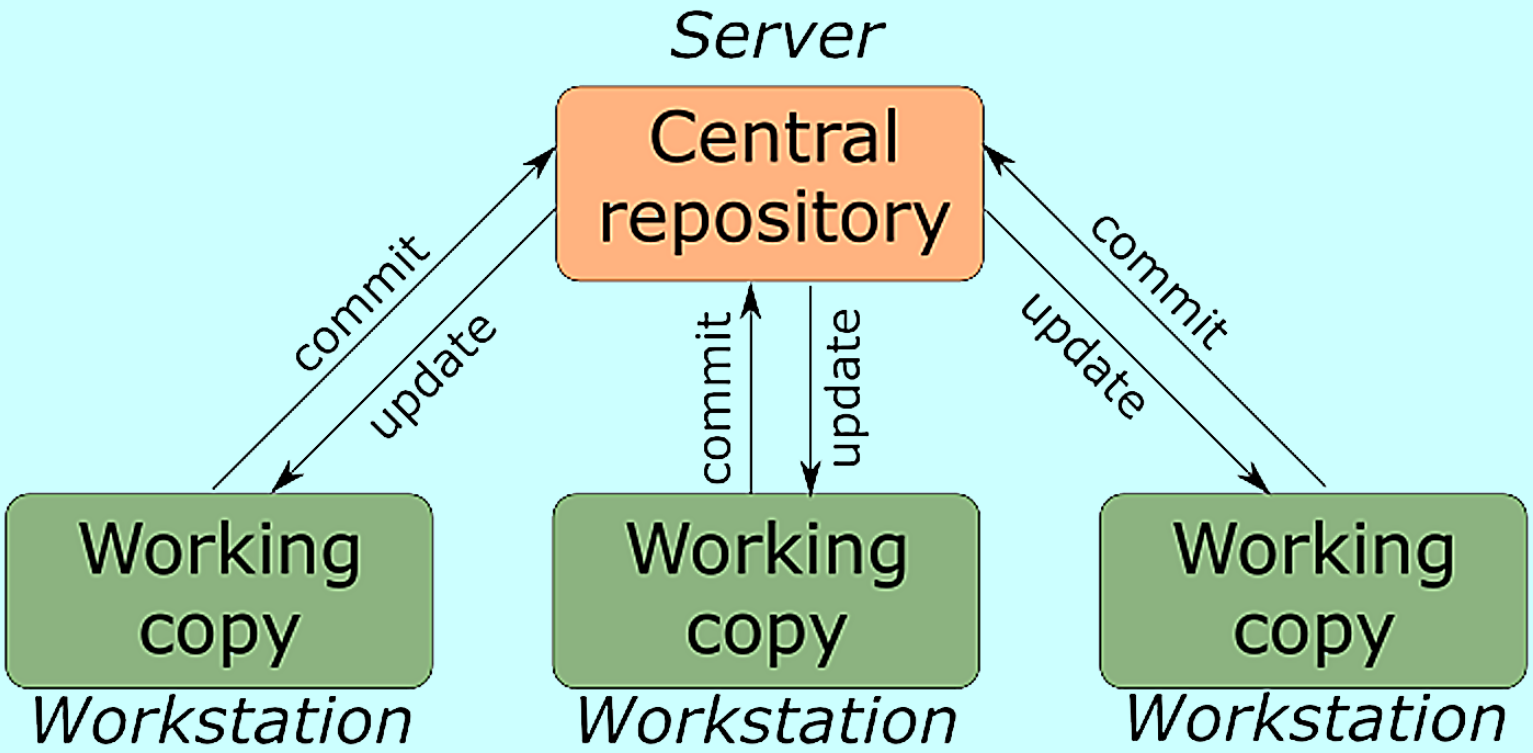
Centralized version control

With centralized version control systems, you have a single “central” copy of your project on a server and commit your changes to this central copy. You pull the files that you need, but you never have a full copy of your project locally.

Distributed version control

With distributed version control systems, you don't rely on a central server to store all the versions of a project’s files. Instead, you clone a copy of a repository locally so that you have the full history of the project.

Centralized version control



CVS (Concurrent Versions System)

It was first released in 1986. CVS is the de facto standard and is installed virtually everywhere. However, the code base isn't as fully featured as SVN or other solutions.

Pros:

- Has been in use for many years and is considered mature technology

Cons:

- Moving or renaming files does not include a version update
- Security risks from symbolic links to files
- No atomic operation support, leading to source corruption
- Branch operations are expensive as it is not designed for long-term branching

SVN (Apache Subversion) *from 2000 year*

SVN was created as an alternative to CVS that would fix some bugs in the CVS system while maintaining high compatibility with it. Because of Subversion's popularity, many different Subversion clients are available: Tortoise SVN (Windows), Versions (Mac).

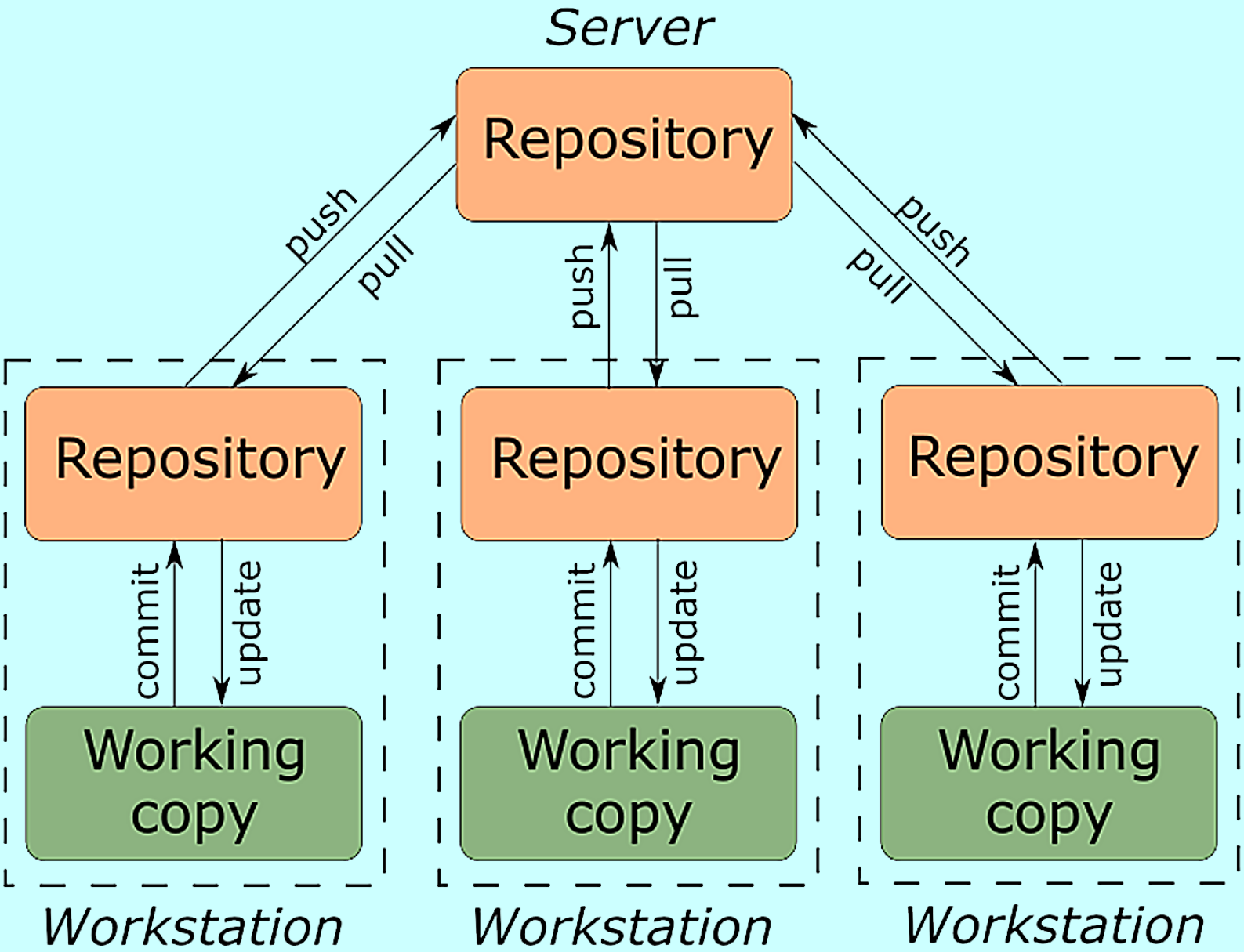
Pros:

- Newer system based on CVS
- Includes atomic operations
- Cheaper branch operations
- Wide variety of plug-ins for IDEs
- Does not use peer-to-peer model

Cons:

- Still contains bugs relating to renaming files and directories
- Insufficient repository management commands
- Slower comparative speed

Distributed version control



Git

from 2005 year

The original concepts for Git were to make a faster, distributed revision control system that would openly defy conventions and practices used in CVS. It is primarily developed for Linux and has the highest speeds on there.

Pros:

- Great for those who hate CVS/SVN
- Dramatic increase in operation speed
- Cheap branch operations
- Full history tree available offline
- Distributed, peer-to-peer model

Cons:

- Learning curve for those used to SVN
- Limited Windows support compared to Linux

Mercurial

from 2005 year

Mercurial was designed for larger projects, most likely outside the scope of designers and independent Web developers.

It's different from other revision control systems in that Mercurial is primarily implemented in Python as opposed to C, but there are some instances where C is used.

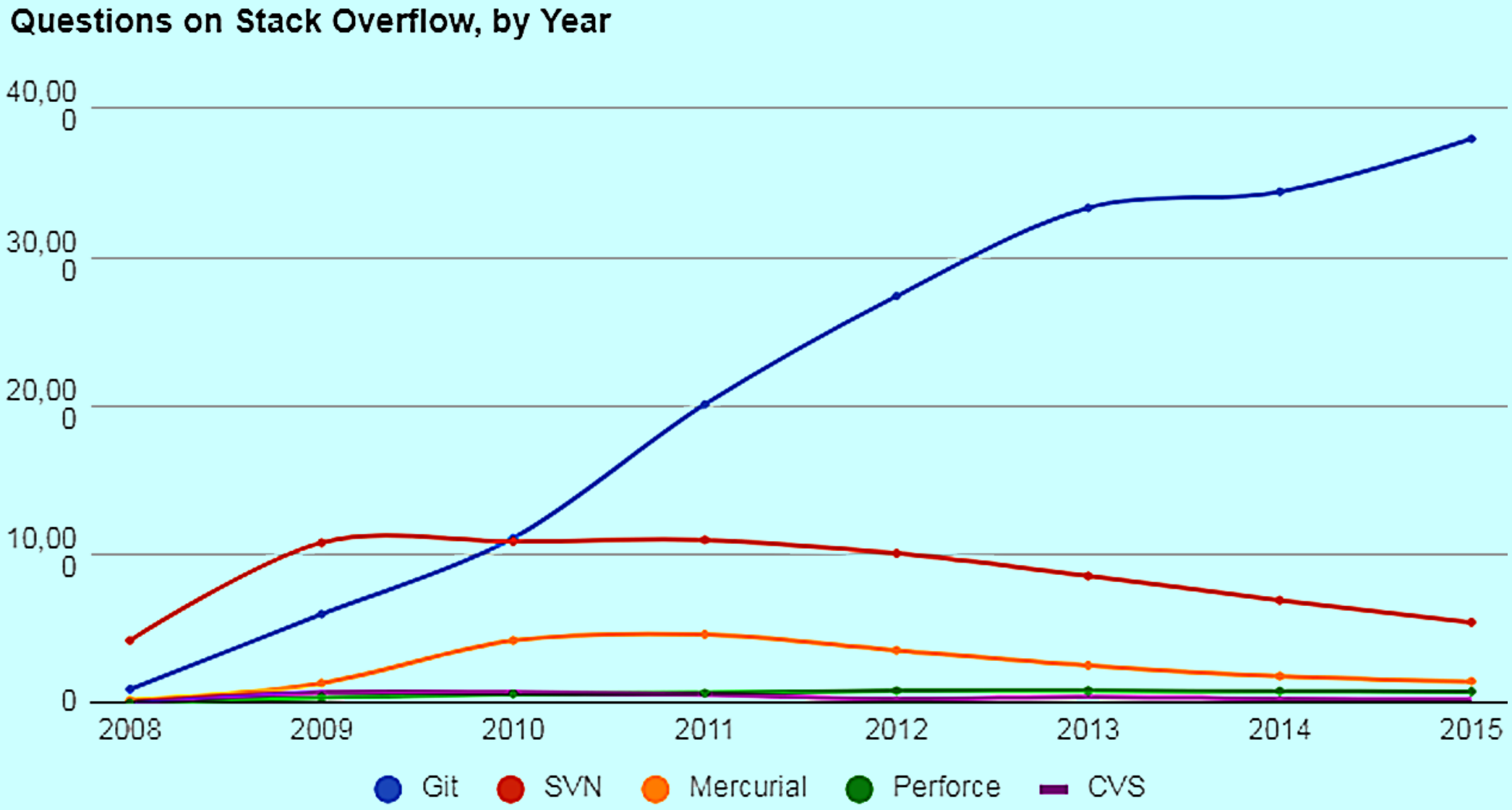
Pros:

- Easier to learn than Git
- Better documentation
- Distributed model

Cons:

- No merging of two parents
- Extension-based rather than scriptability
- Less out of the box power

Rating of version control systems



https://rhodecode.com/insights/version-control-systems-2016

Rating of version control systems

2018

https://hype.codes/version-control-systems-rating

Worldwide, Mar 2018 compared to a year ago:

1	Git	31.81	-2.56
2	GitHub	23.1	0.64
3	Fossil	20.92	-1.08
4	Bazaar	15.69	3.77
5	Mercurial	3.49	-0.18
6	Kiln	2.29	0.02
7	Beanstalk	1.28	-0.11
8	Subversion	0.57	-0.22

Git basics

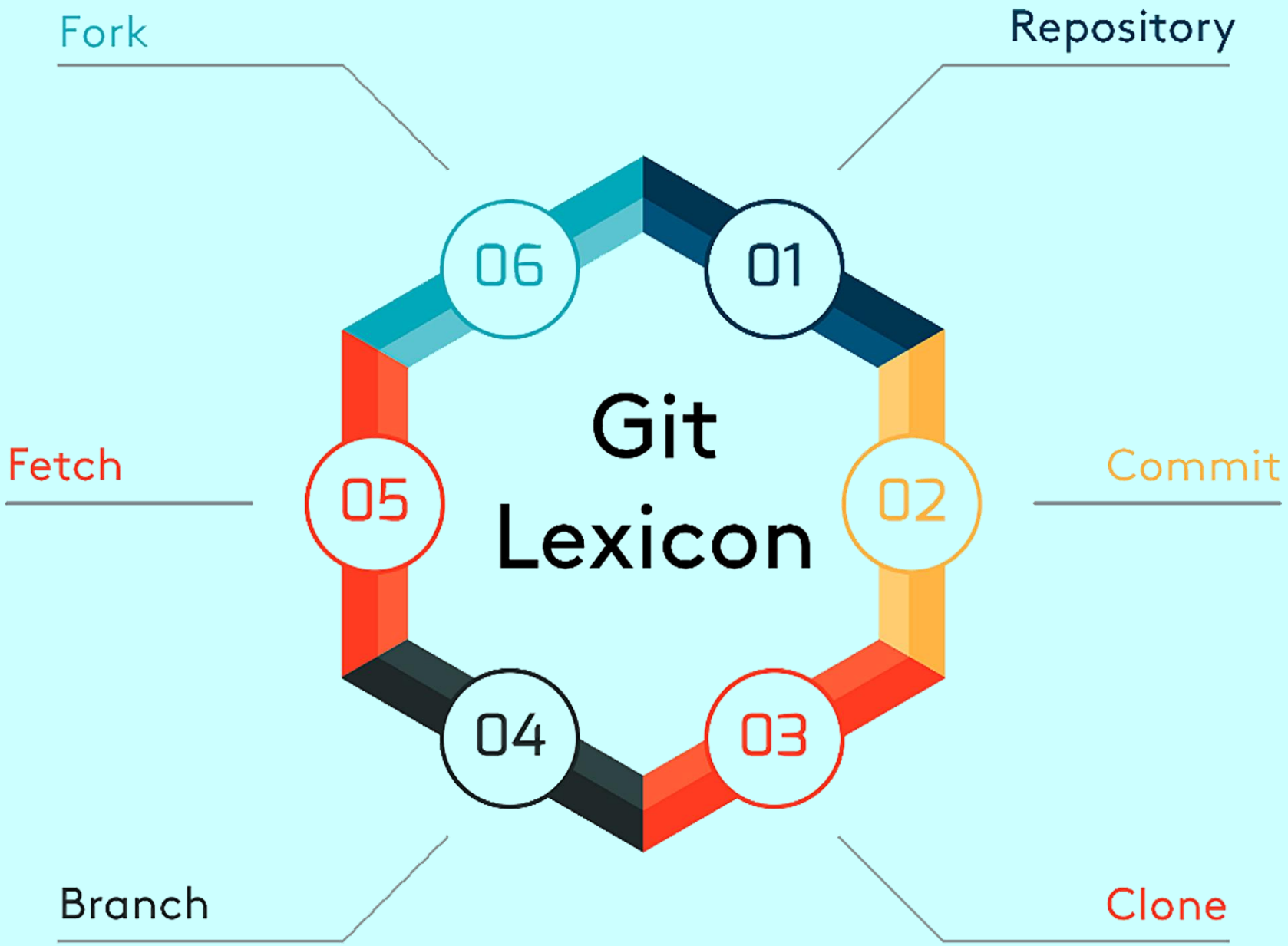
What are GitHub or Bitbucket?

They are like Wikipedia for programmers.
You can edit files, see who changed what,
view old versions of files, and access it
from anywhere in the world.

What is Git?

Git is an open source code management system. The basic idea of Git is to keep track of different versions of code or text, so you easily can compare what has changed.

Git terminology



Git terminology

The most fundamental element of GitHub, a repository is essentially a project's folder. Repositories store every single project file, its documentation and its revision history of every document.

Commits are easily one of the most frequented activities by a developer using GitHub. A commit is like 'saving' an updated file to its original folder.

Clones are literally clones (copies) of a repository that sit on the developer's computer instead of a server. Clones are great since you can download a code file to tinker around with offline.

A branch is a parallel version of a repository. It is contained within the repository, but does not affect the primary or master branch allowing you to work freely without disrupting the "live" version.

Git terminology

Fetch

Fetching refers to getting the latest changes from an online repository (like GitHub.com) without merging them in. Once these changes are fetched you can compare them to your local branches.

Fork

A ‘fork’ is a personal copy of another user's repository that lives on your GitHub account. Forks allow you to freely make changes to a project without affecting the original.

Push

Pushing refers to sending your committed changes to a remote repository. If you change something locally, you'd want to then push those changes so that others may access them.

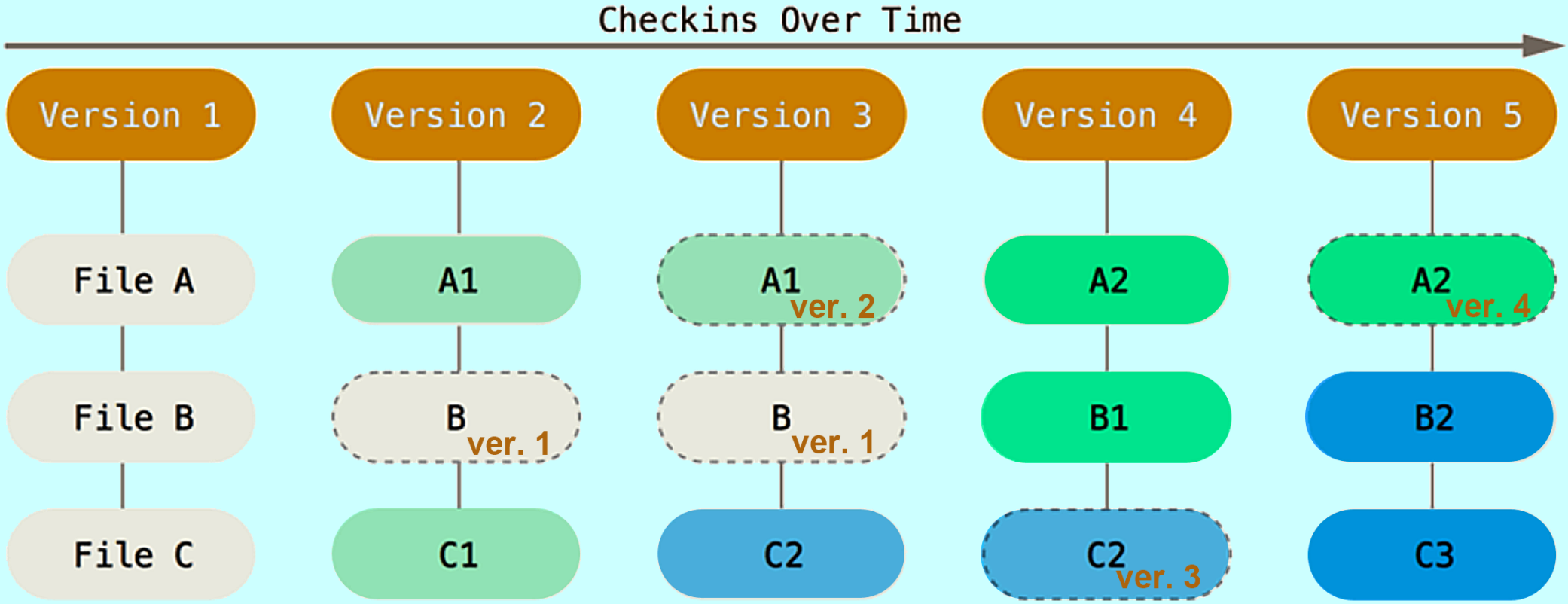
Pull

Retrieve remote changes. A Pull command is similar to a Fetch command, except that it both retrieves remote changes as well as merging them in to you own commit history.

Git terminology

Pull Request	Collaborators without write access can send a pull request to the administrator with the changes they've made to the remote repo. The administrator can then approve and merge or reject the changes to the main repository.
Merge	Merging takes the changes from one branch, and applies them into another. A merge can be done automatically via a Pull Request or can be done via the command line.
Rebase	Another way to merge changes from one branch to another. Similar to merge, rebase allows you to include changes from one branch on to another. Unlike the merge command, rebase replays a branch’s commit history onto the branch it’s merging into.
Blame	The "blame" feature in Git describes the last modification to each line of a file, which generally displays the revision, author and time.

Snapshots

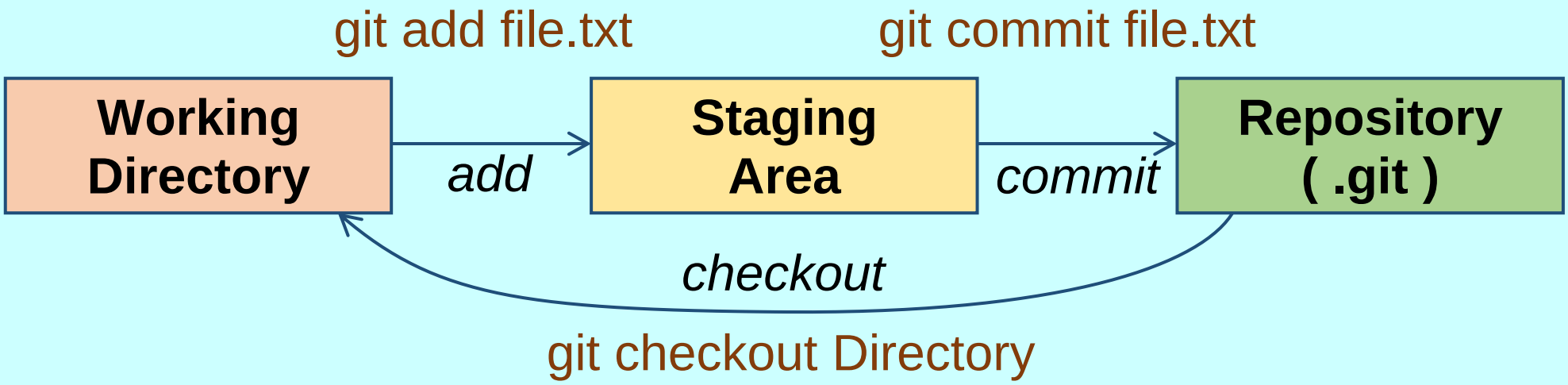


Snapshot is to a repository as screenshot is to a video.

Git repository consists of series of snapshots, which are a bunch of files and folders represented by a Git's tree object, stored in the repository's .git folder. When you commit, you store your current working directory as a new snapshot (commit).

If a file did not change, only a link to the file is stored.

Three-tree architecture



The basic Git workflow goes something like this:

1. You modify files in your working tree.
2. You selectively stage just those changes you want to be part of your next commit, which adds only those changes to the staging area.
3. You do a commit, which takes the files as they are in the staging area and stores that snapshot permanently to your Git directory.

What is a hash?

Every commit to a repository has a unique identifier called a SHA-1 hash. This is a 40-character hexadecimal string calculated based on the contents of a file or directory structure in Git. A SHA-1 hash looks like this:

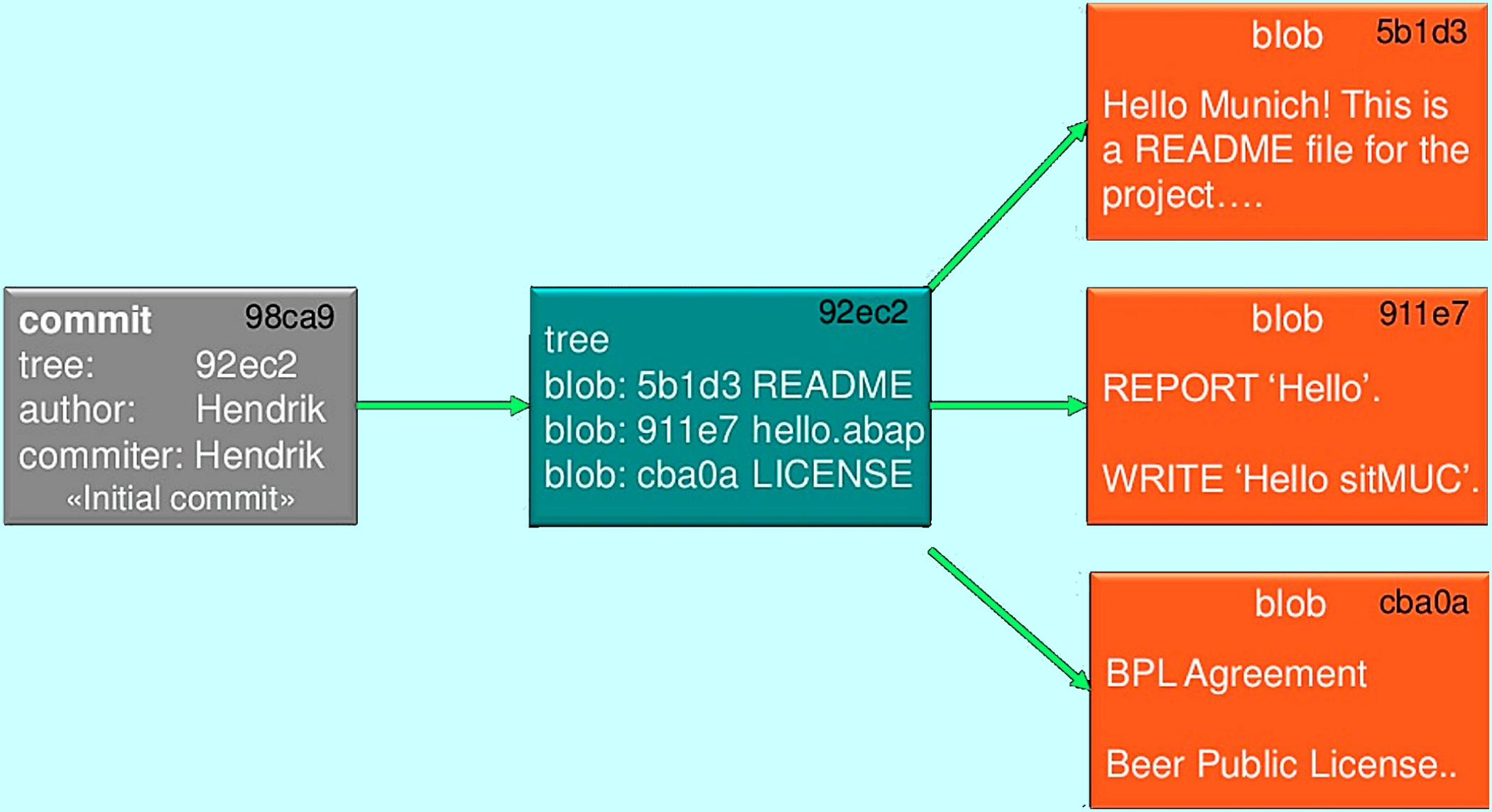
7c35a3ce607a14953f070f0f83b5d74c2296ef93

but most of the time, you only have to give Git the first 6 or 8 characters in order to identify the commit you mean.

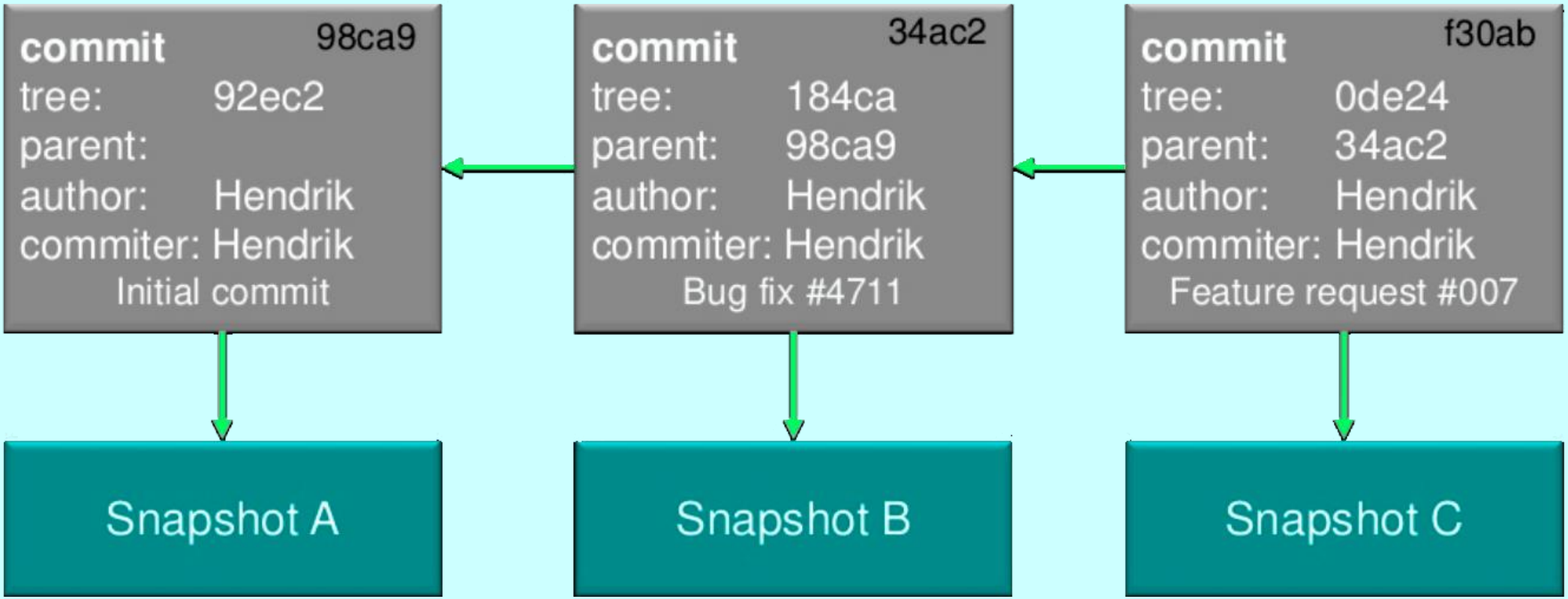
What is a hash?


```
commit e83c5163316f89bfbde7d9ab23ca2e25604af29
Author: Linus Torvalds <torvalds@ppc970.osdl.org>
Date: Thu Apr 7 15:13:13 2005 -0700

Initial revision of "git", the information manager
from hell
```

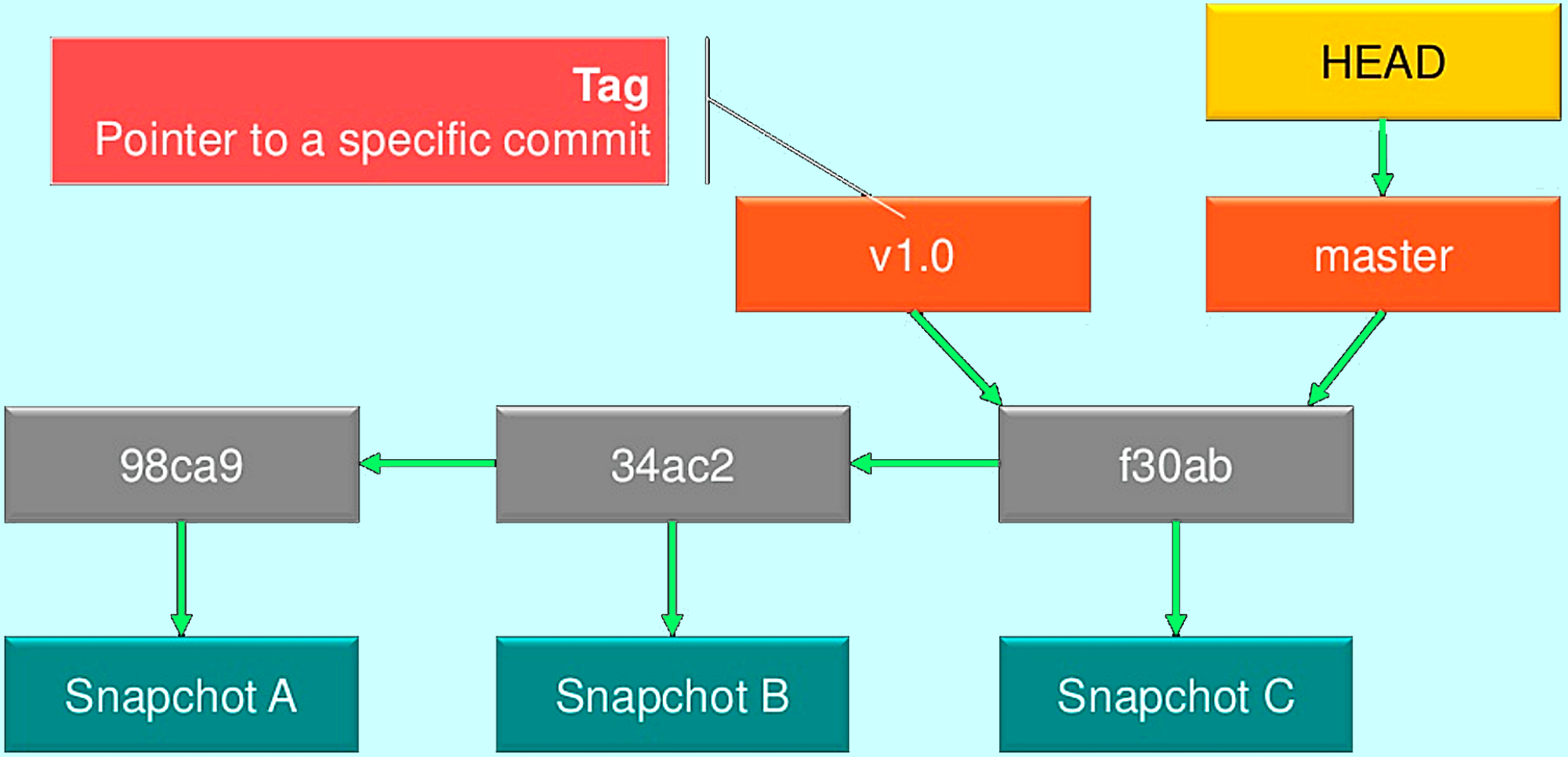
A commit and its tree



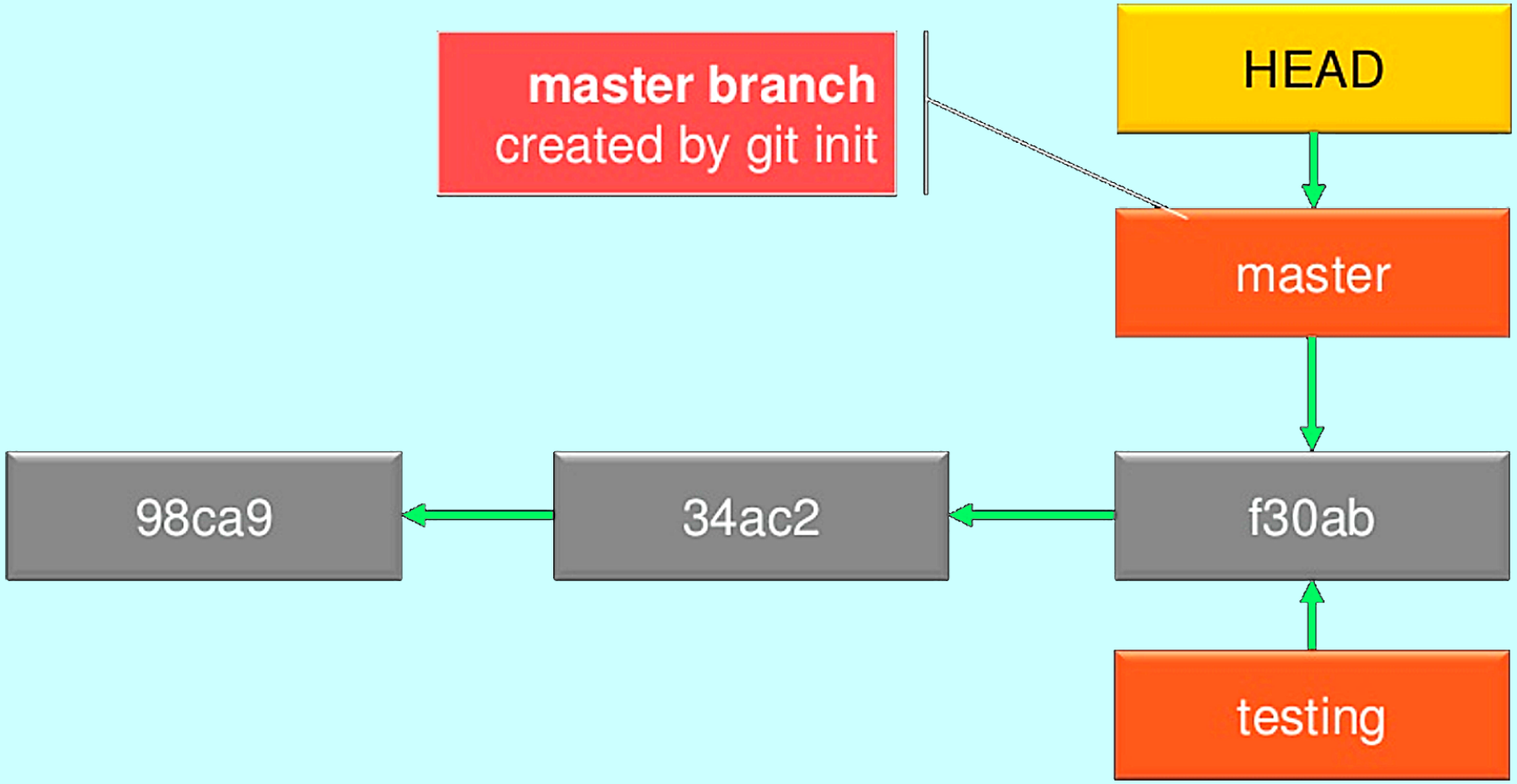
Commits and parents



A branch and its commit history

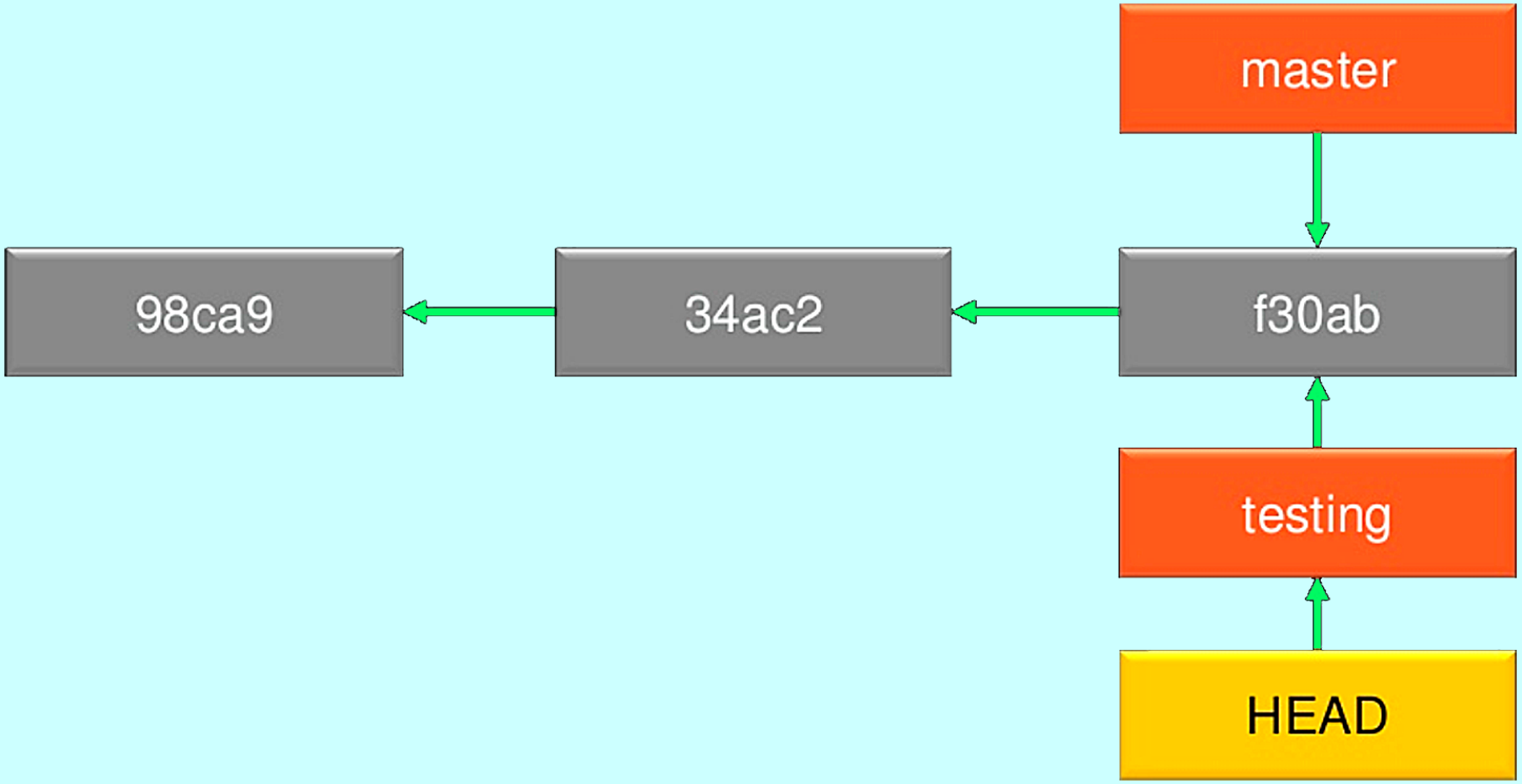


git branch testing



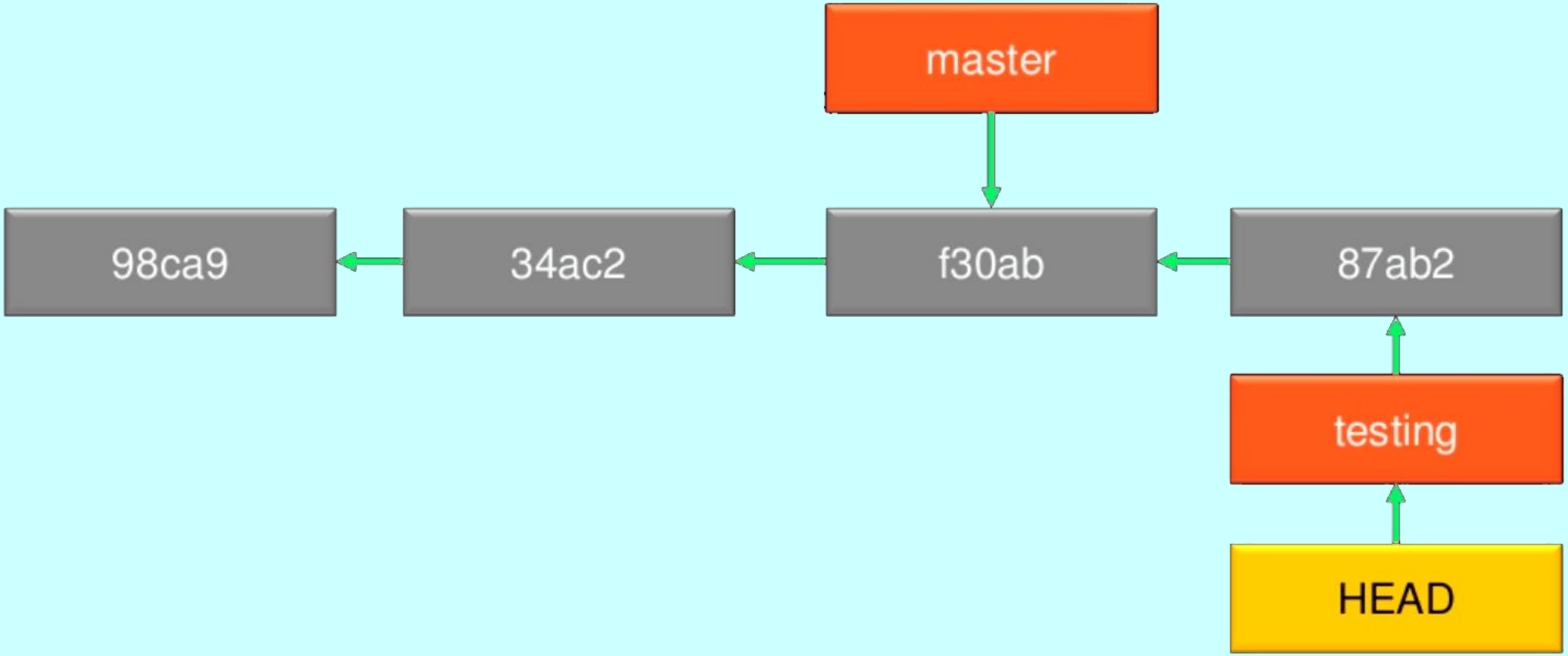
Switch to another branch

git checkout testing



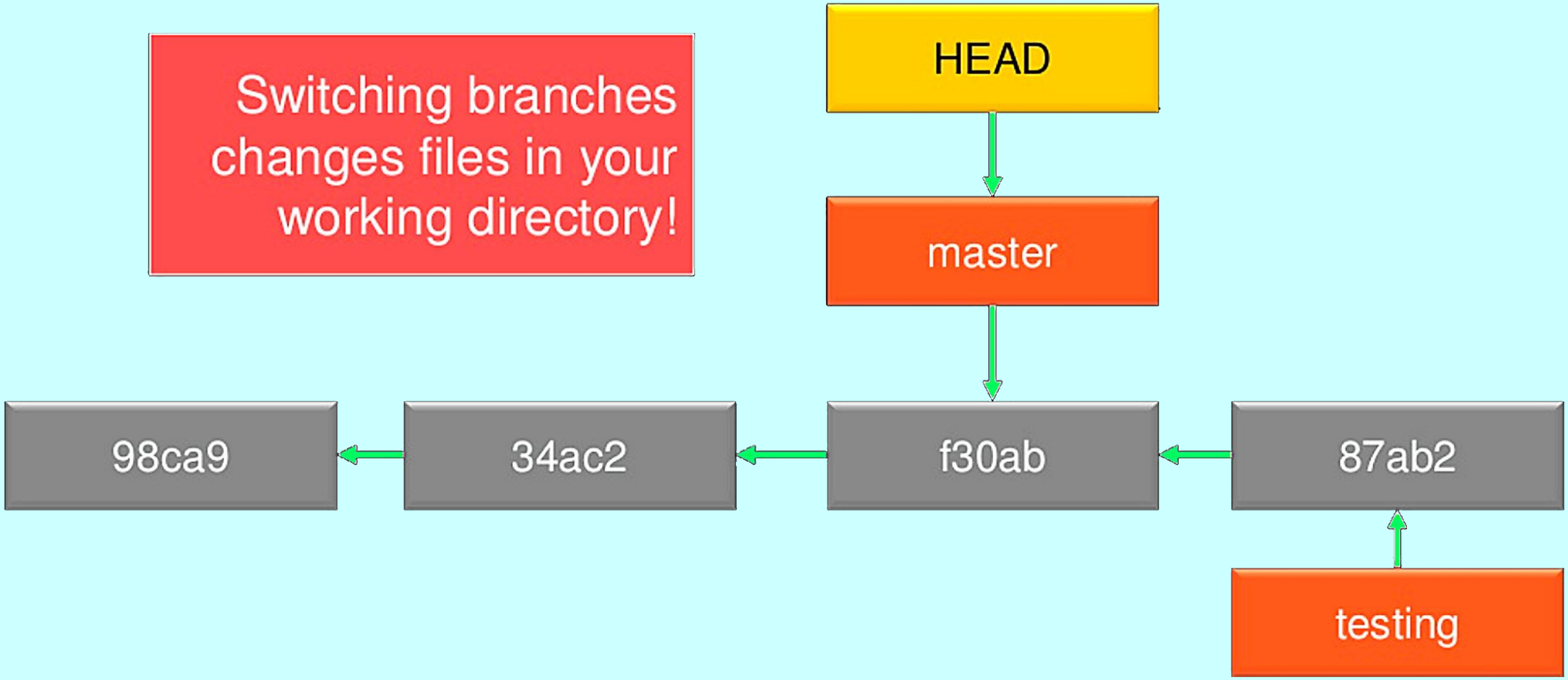
New commit

git commit –a –m "made a change"



Switch to branch "master"

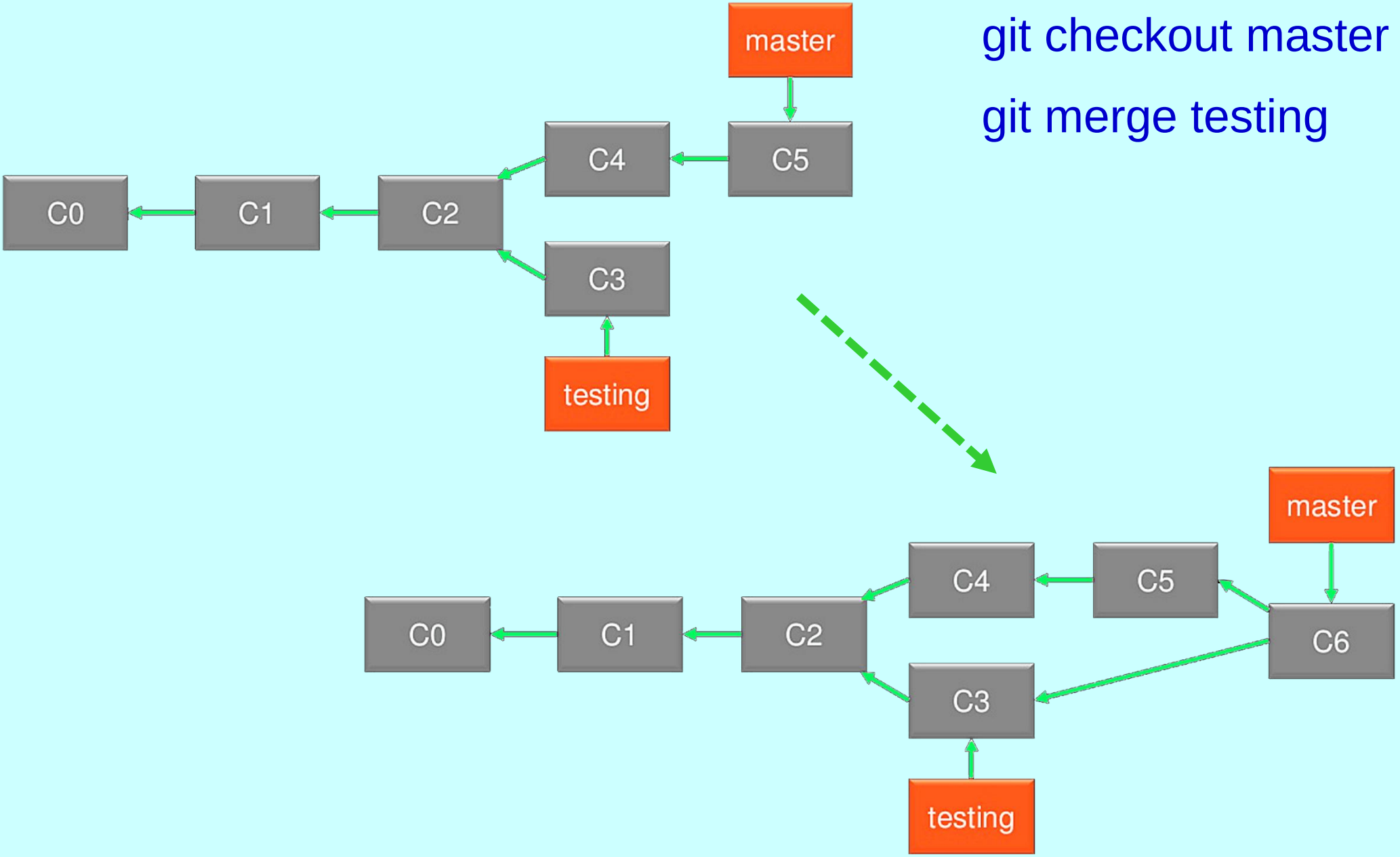
git checkout master



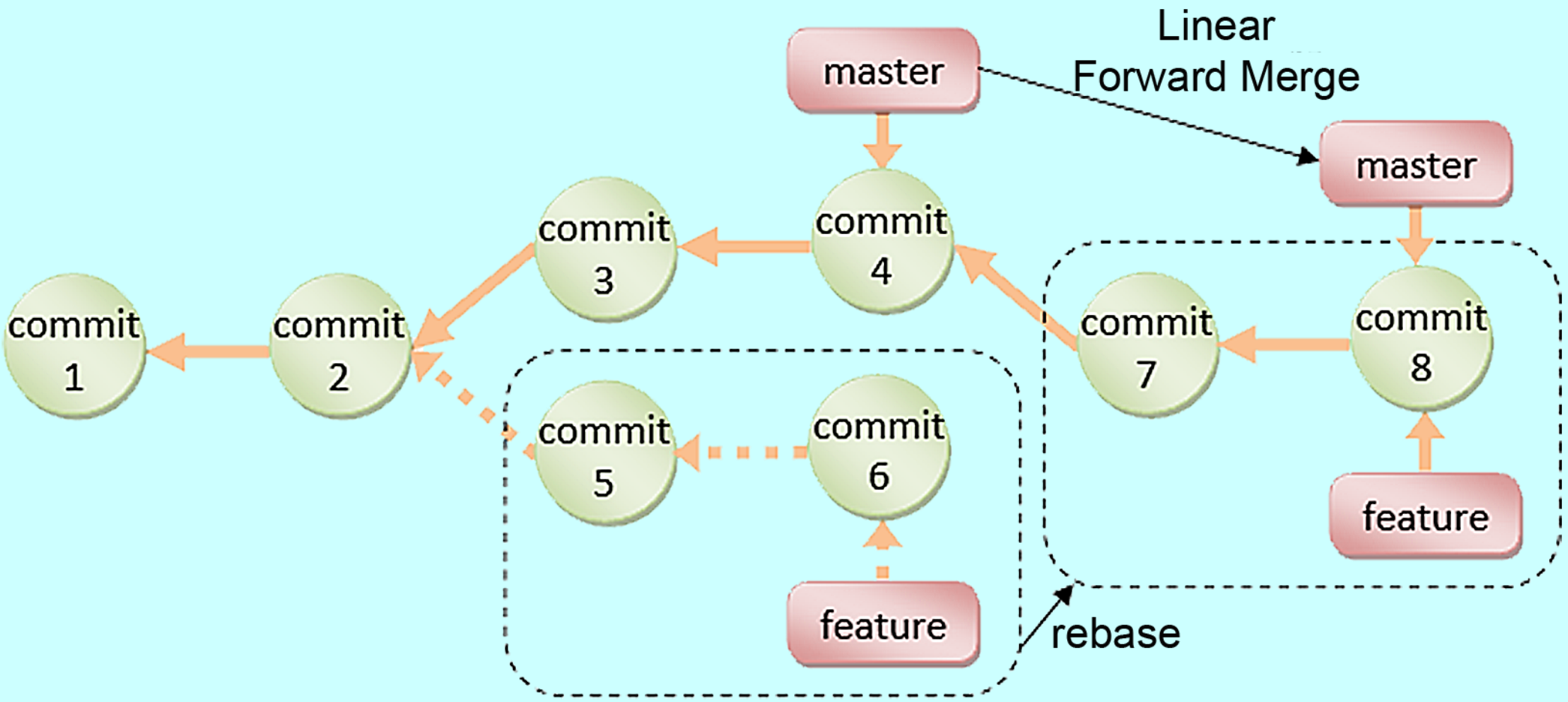
Git merge

git checkout master

git merge testing



Rebasing branch

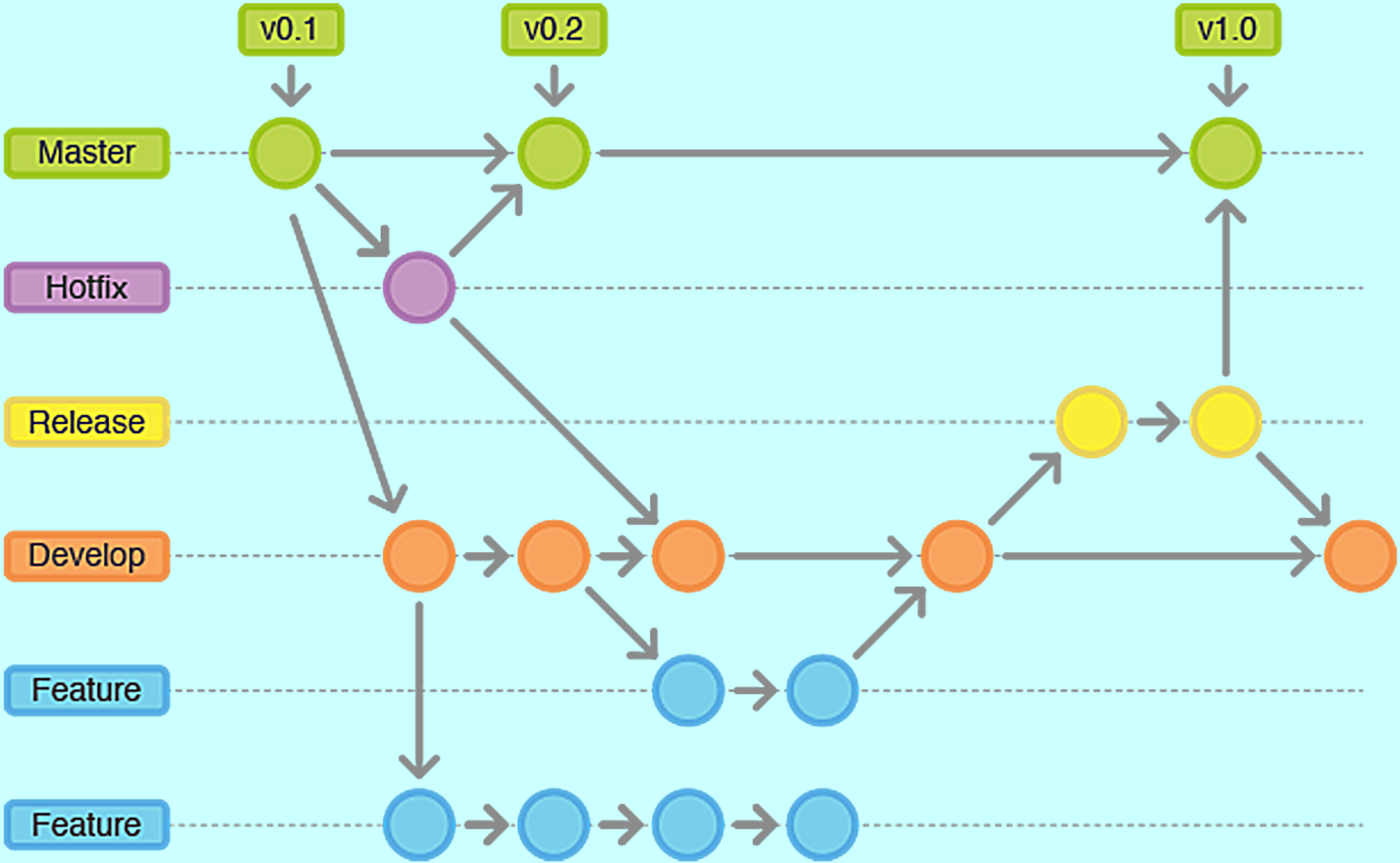


The primary purpose for rebasing is to maintain a linear project history.

Rebasing branch

- ONLY rebase on a local repository, never rewrite shared history.
- If you have pushed your changes, live with it ☺.
- You can use “git revert” to develop compensating commits.

Git Flow Works



Naming Git branches

Stable:	master
Hotfix:	hotfix/XXX
Release:	release/XXX
	develop
Feature:	feature/XXX
Bugfix:	bugfix/XXX

Git using

Git installation

https://git-scm.com

git --everything-is-local

Search entire site...

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.

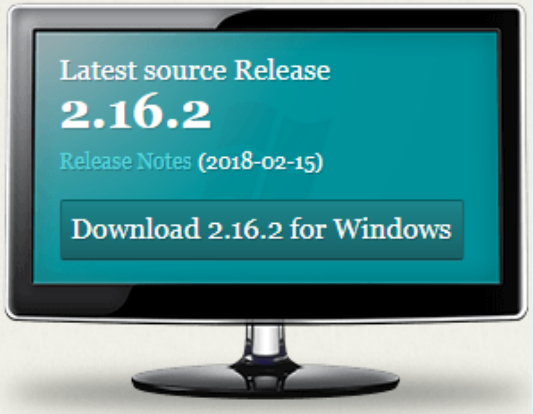
Learn Git in your browser for free with Try Git.

About
The advantages of Git compared to other source control systems.

Documentation
Command reference pages, Pro Git book content, videos and other material.

Downloads
GUI clients and binary releases for all major platforms.

Community
Get involved! Bug reporting, mailing list, chat, development and more.



Pro Git by Scott Chacon and Ben Straub is available to read online for free. Dead tree versions are available on Amazon.com.

Windows GUIs **Tarballs**

Mac Build **Source Code**

Git installation for Linux

https://git-scm.com/download/linux

Debian/Ubuntu

For the latest stable version for your release of Debian/Ubuntu

```
# apt-get install git
```

For Ubuntu, this PPA provides the latest stable upstream Git version

```
# add-apt-repository ppa:git-core/ppa
```

```
# apt update; apt install git
```

Fedora

```
# yum install git
```
(up to Fedora 21)

```
# dnf install git
```
(Fedora 22 and later)

Gentoo

```
# emerge --ask --verbose dev-vcs/git
```

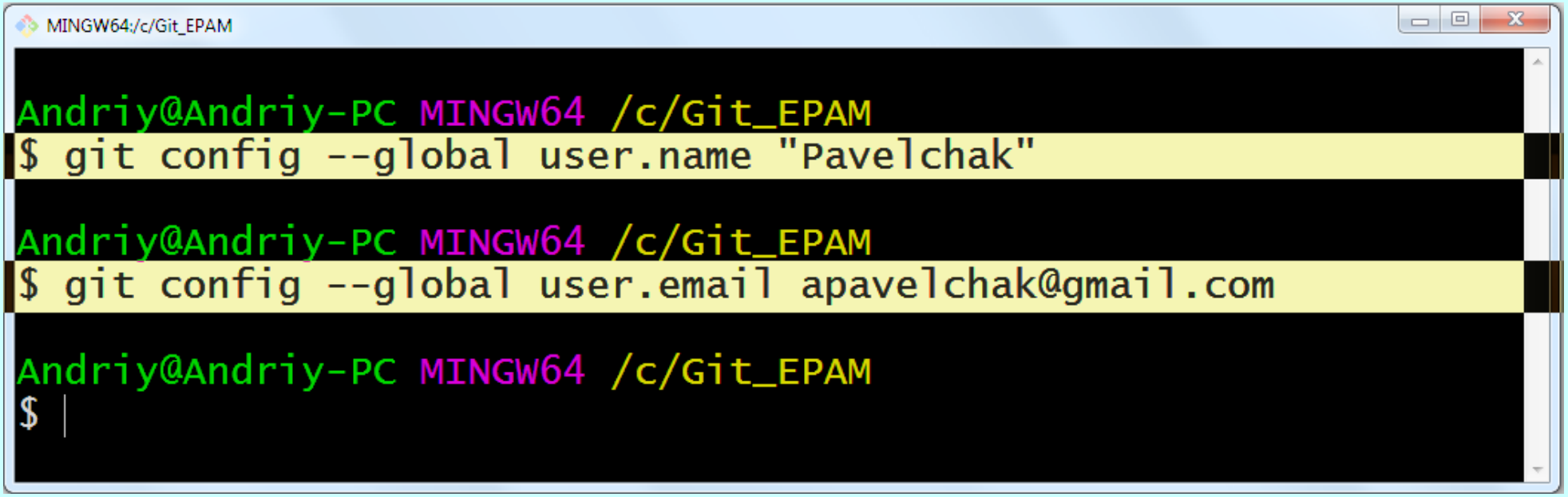
Arch Linux

```
# pacman -S git
```

openSUSE

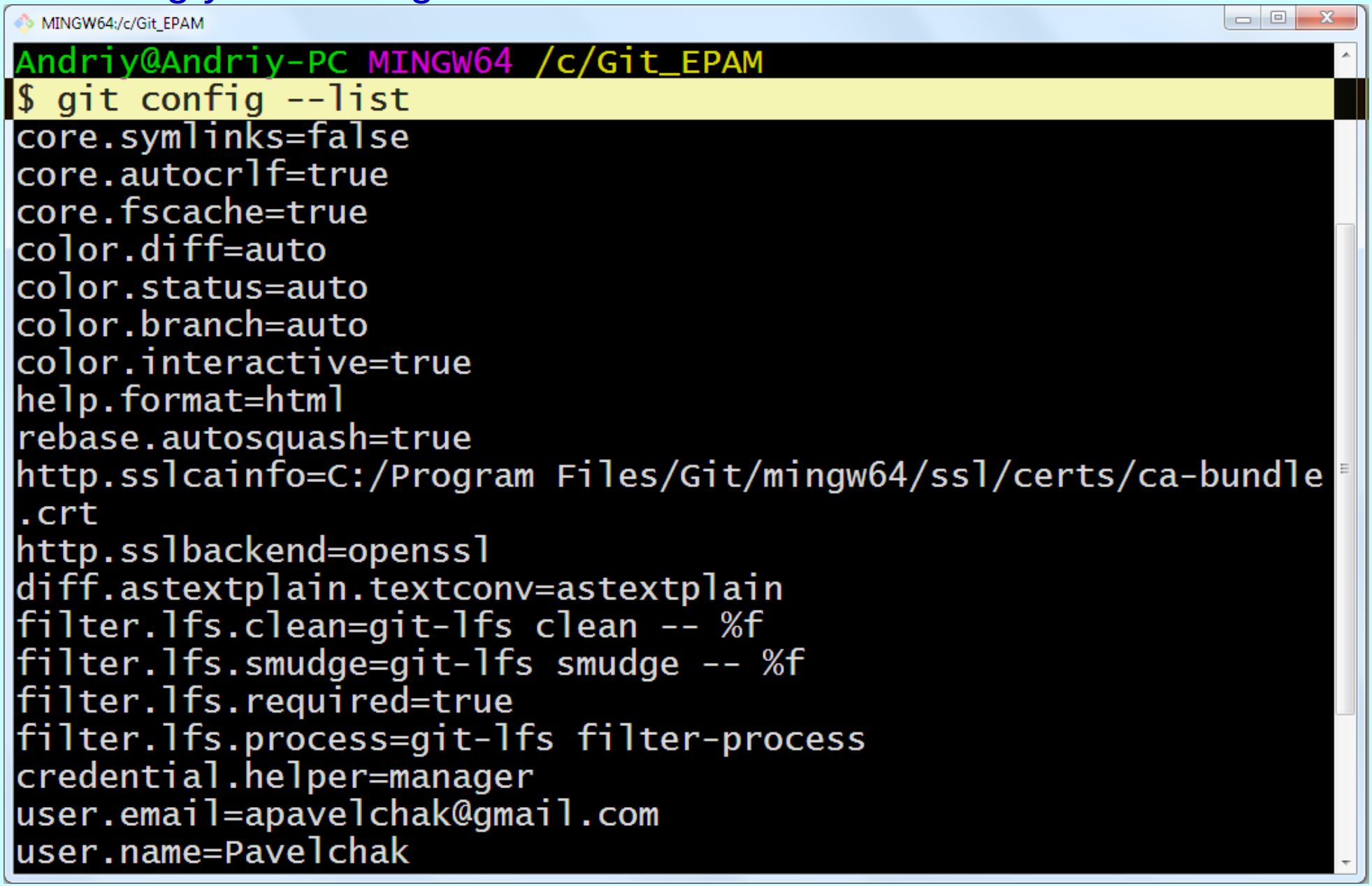
```
# zypper install git
```

Your identity:



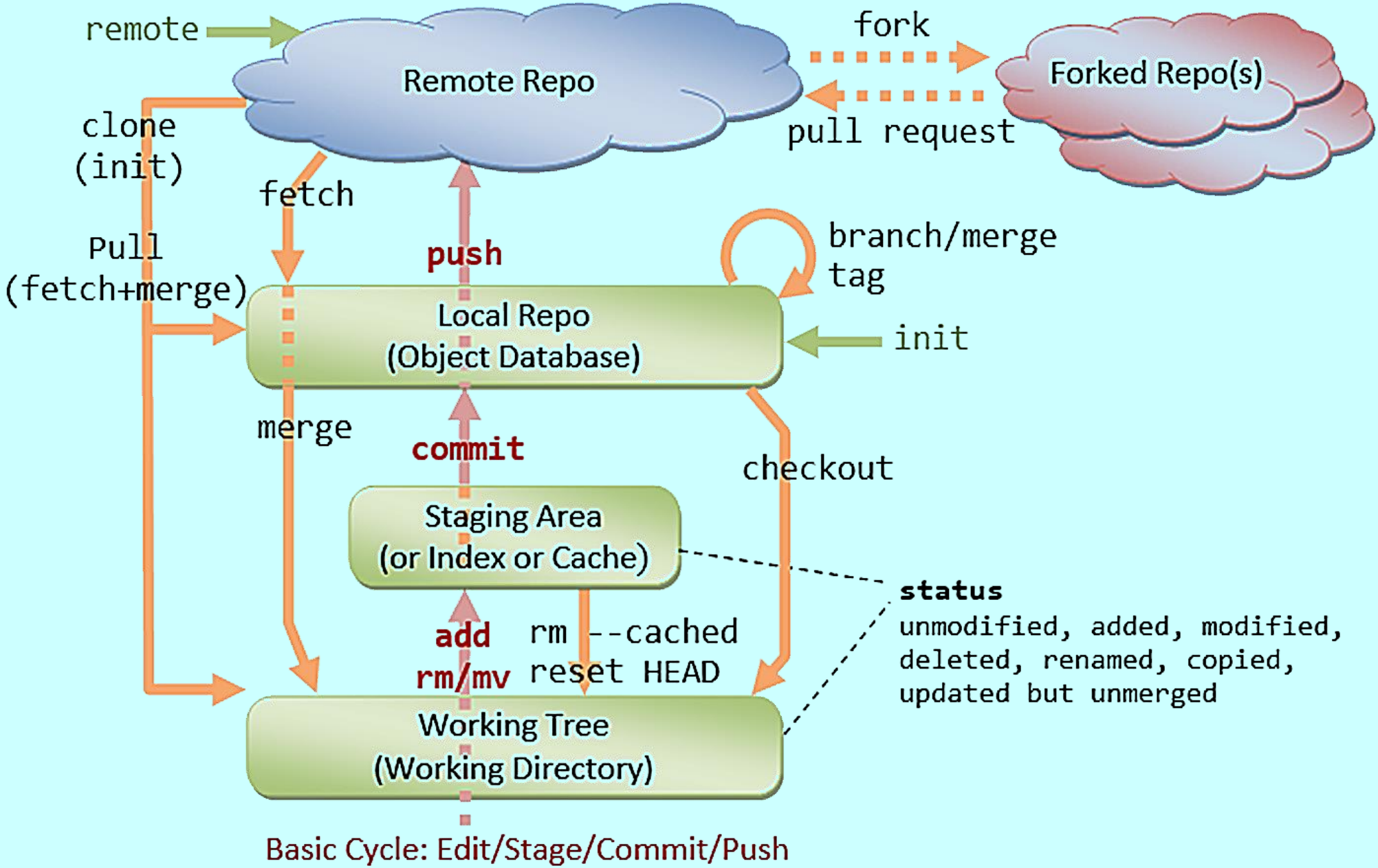
First-time Git setup

Checking your settings:



Git Storage Model

https://www.ntu.edu.sg



Cloning an existing project from a GIT host

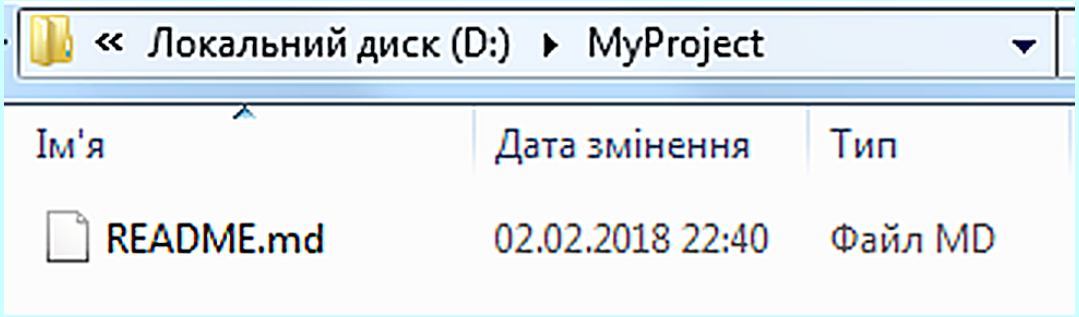
MINGW64:/d/CloneSM

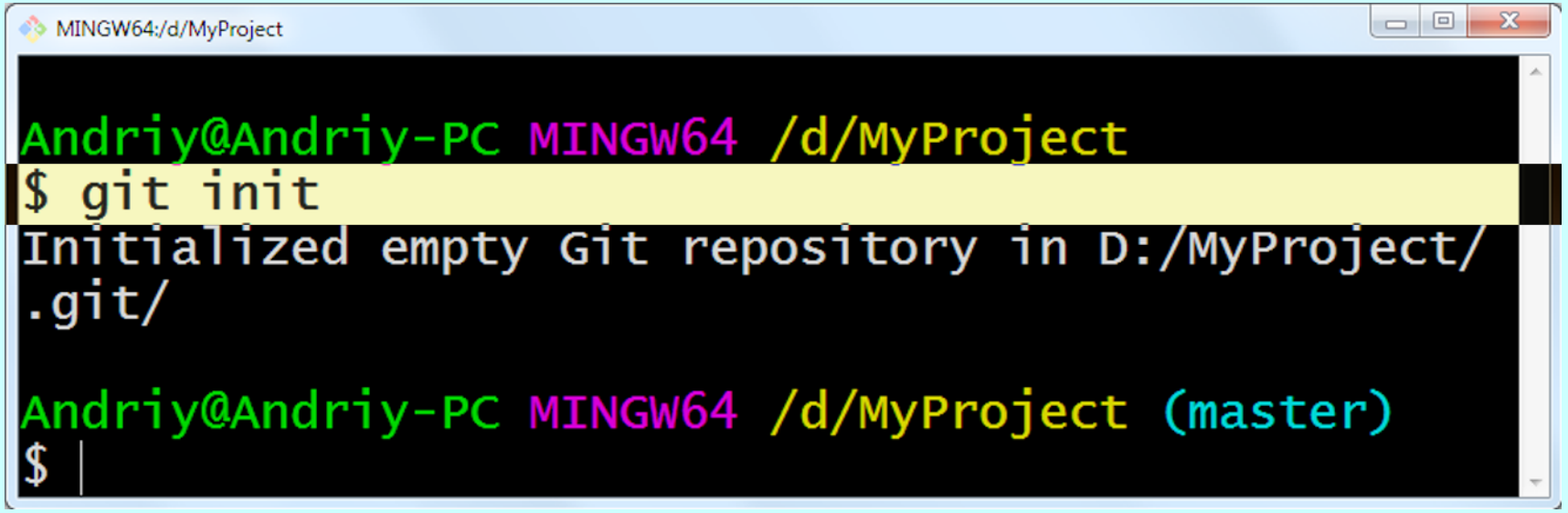
```
Andriy@Andriy-PC MINGW64 /d/CloneSM
$ git clone https://bitbucket.org/pavelchak/junit
Cloning into 'junit'...
remote: Counting objects: 78, done.
remote: Compressing objects: 100% (51/51), done.
remote: Total 78 (delta 16), reused 0 (delta 0)
Unpacking objects: 100% (78/78), done.

Andriy@Andriy-PC MINGW64 /d/CloneSM
$
```

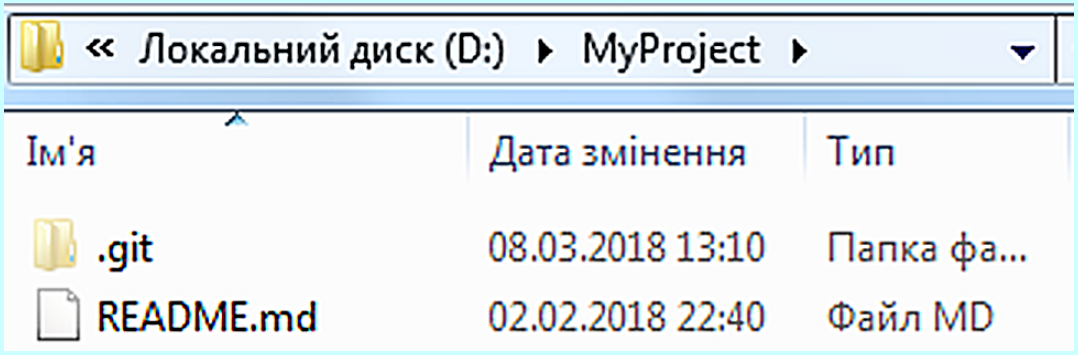
« Локальний диск (D:) ▸ CloneSM ▸ junit ▸

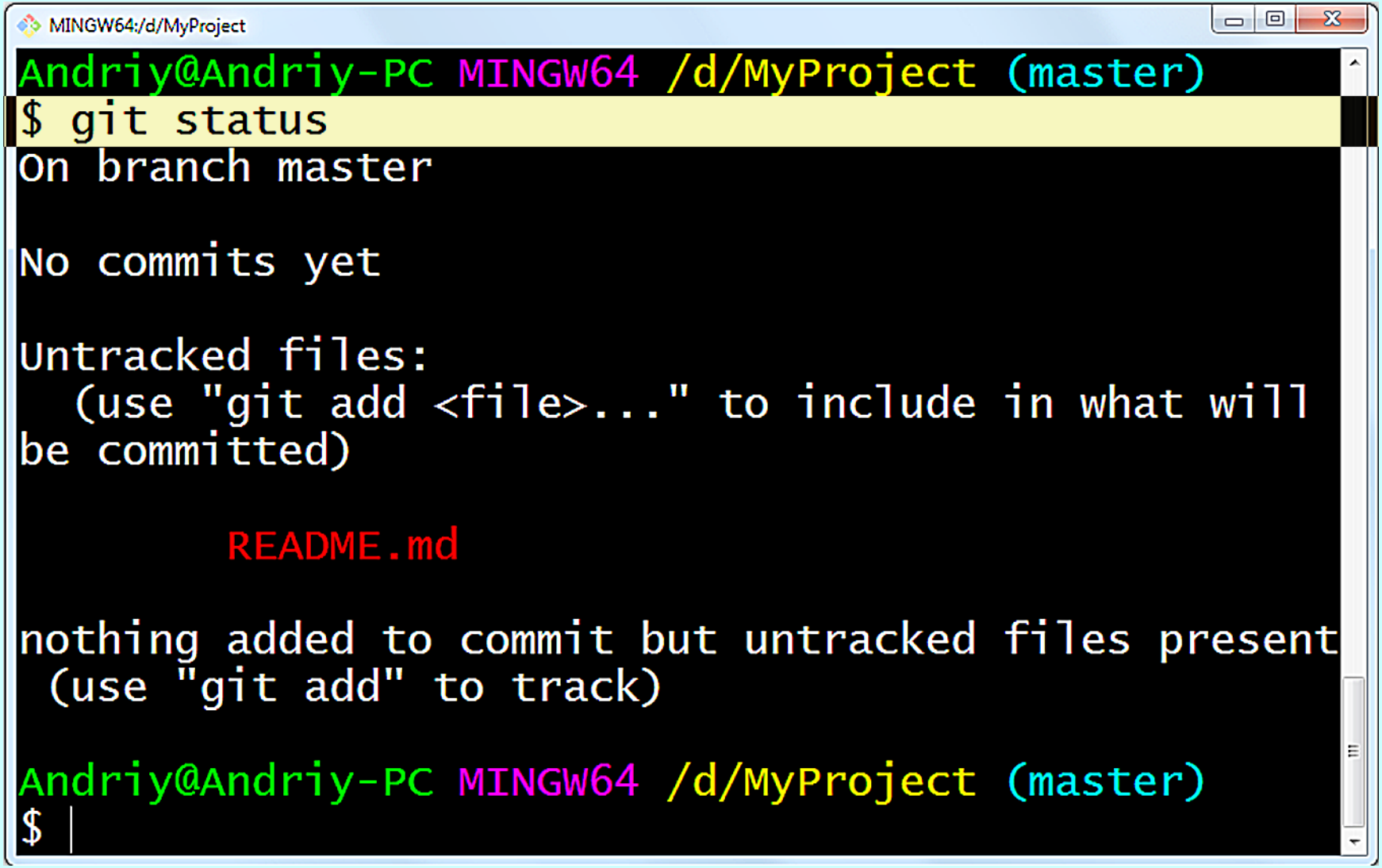
Ім'я	Дата змінення	Тип	Розмір
.git	08.03.2018 0:39	Папка файлів	
src	08.03.2018 0:39	Папка файлів	
.gitignore	08.03.2018 0:39	Текстовий докум...	1 КБ
info.txt	08.03.2018 0:39	Текстовий докум...	1 КБ
JUnit.iml	08.03.2018 0:39	Файл IML	2 КБ
pom.xml	08.03.2018 0:39	Документ XML	2 КБ

Starting own project








MINGW64:/d/MyProject
Andriy@Andriy-PC MINGW64 /d/MyProject (master)
$ git status
On branch master
No commits yet
Untracked files:
(use "git add <file>..." to include in what will be committed)
README.md
nothing added to commit but untracked files present (use "git add" to track)
Andriy@Andriy-PC MINGW64 /d/MyProject (master)
$

Starting own project (**Add**)

```
git add <path to file>
```

Add the individual file.

```
git add <directory name>
```

Will add all files in the directory.

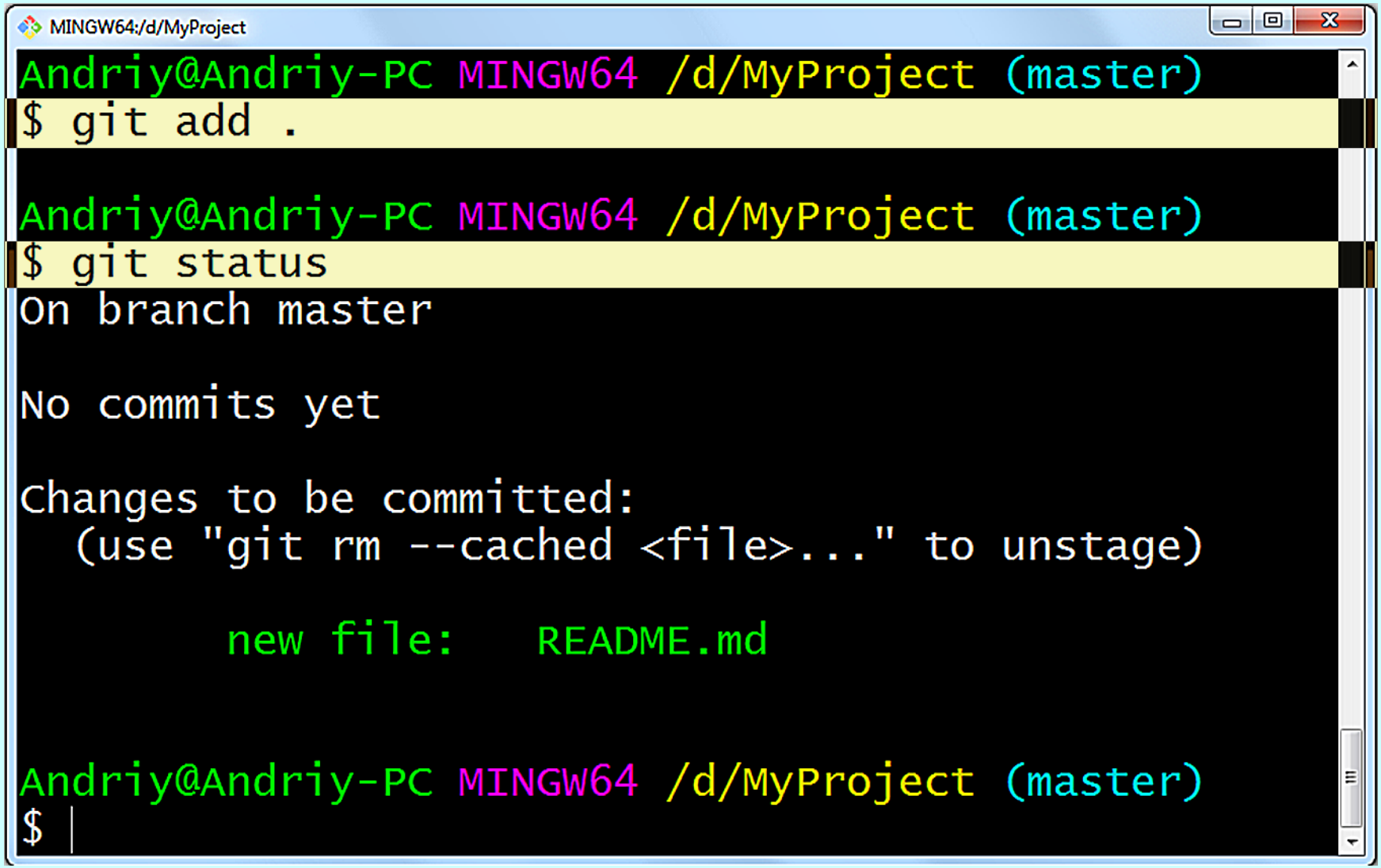
```
git add .
```

Adds all untracked and modified files, even those in subdirectories.

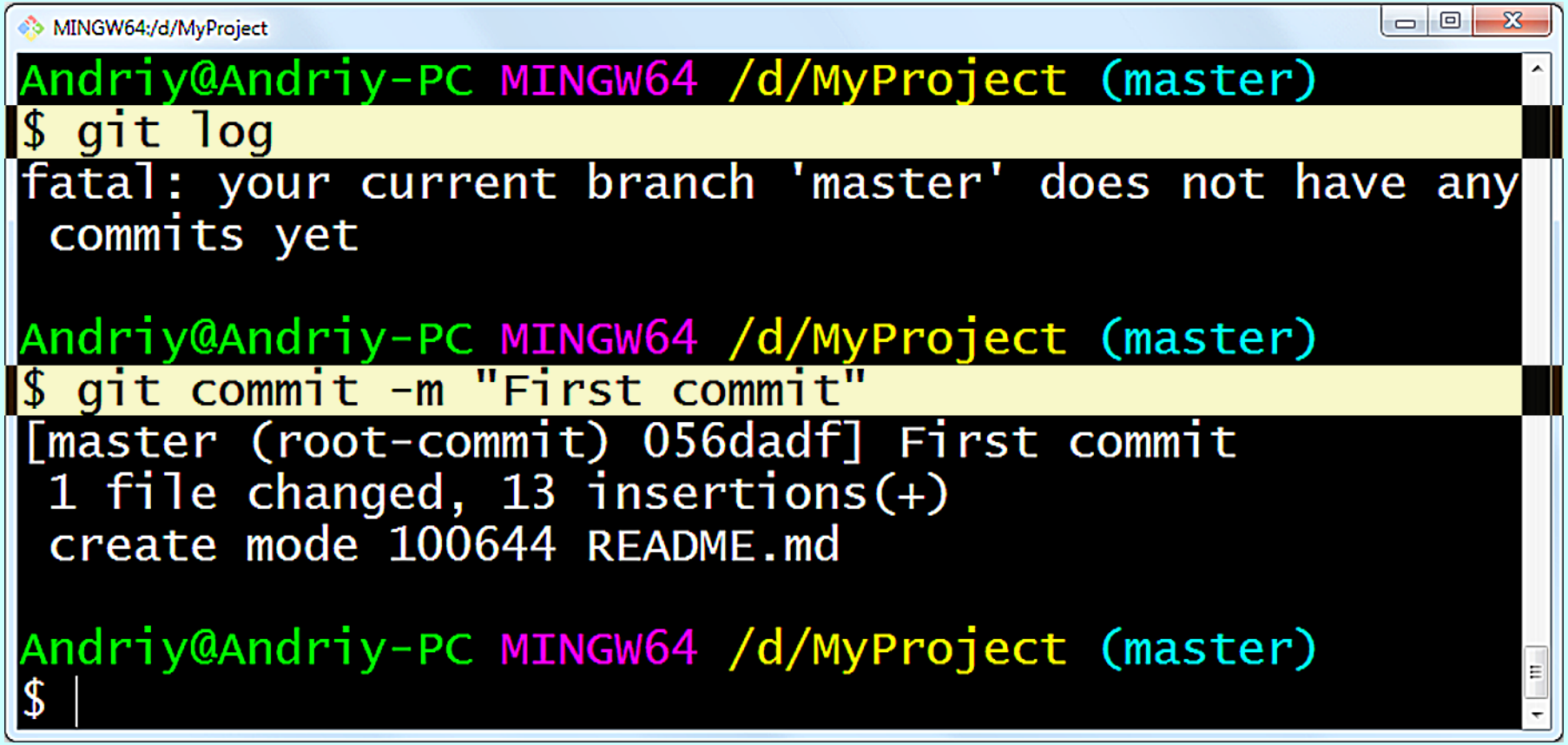
```
git add *.txt
```

Wildcards can be used to add all files of a specific type, in this instance text files.

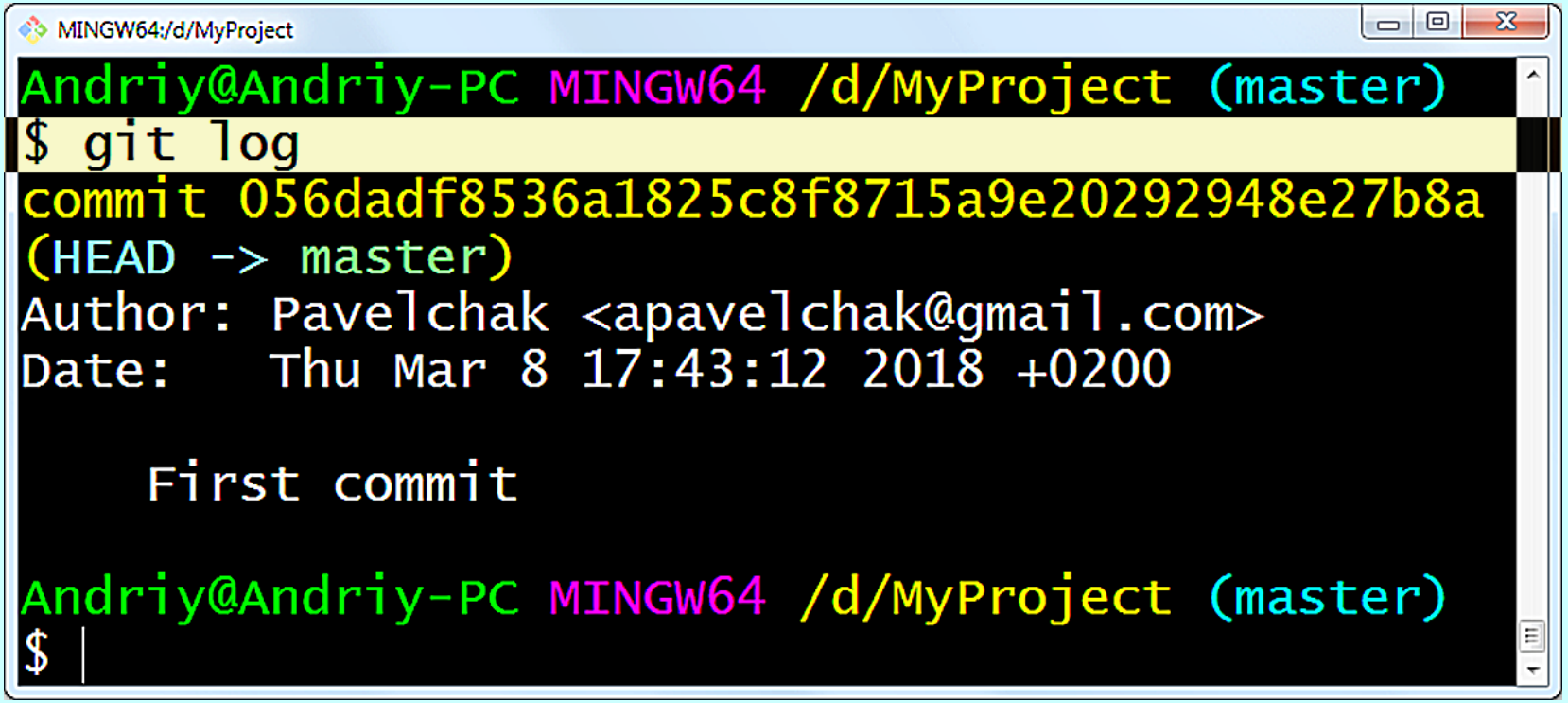
Starting own project (**Add**)



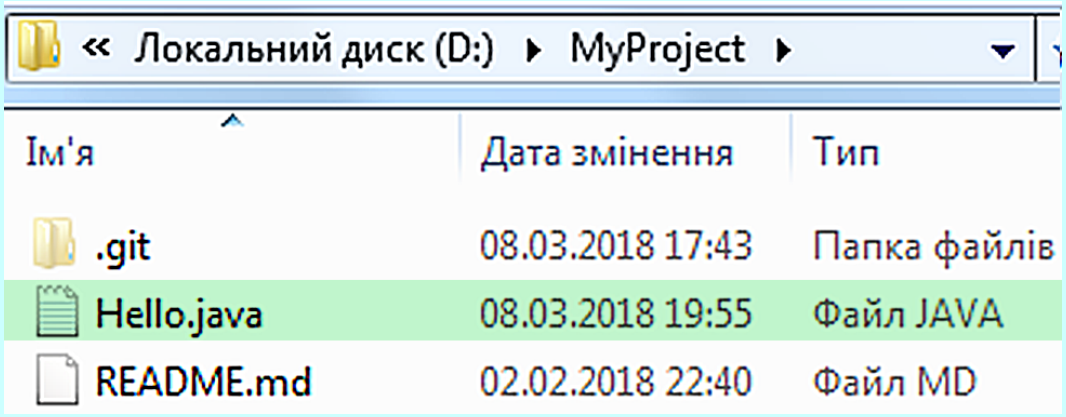
Starting own project (**Commit**)

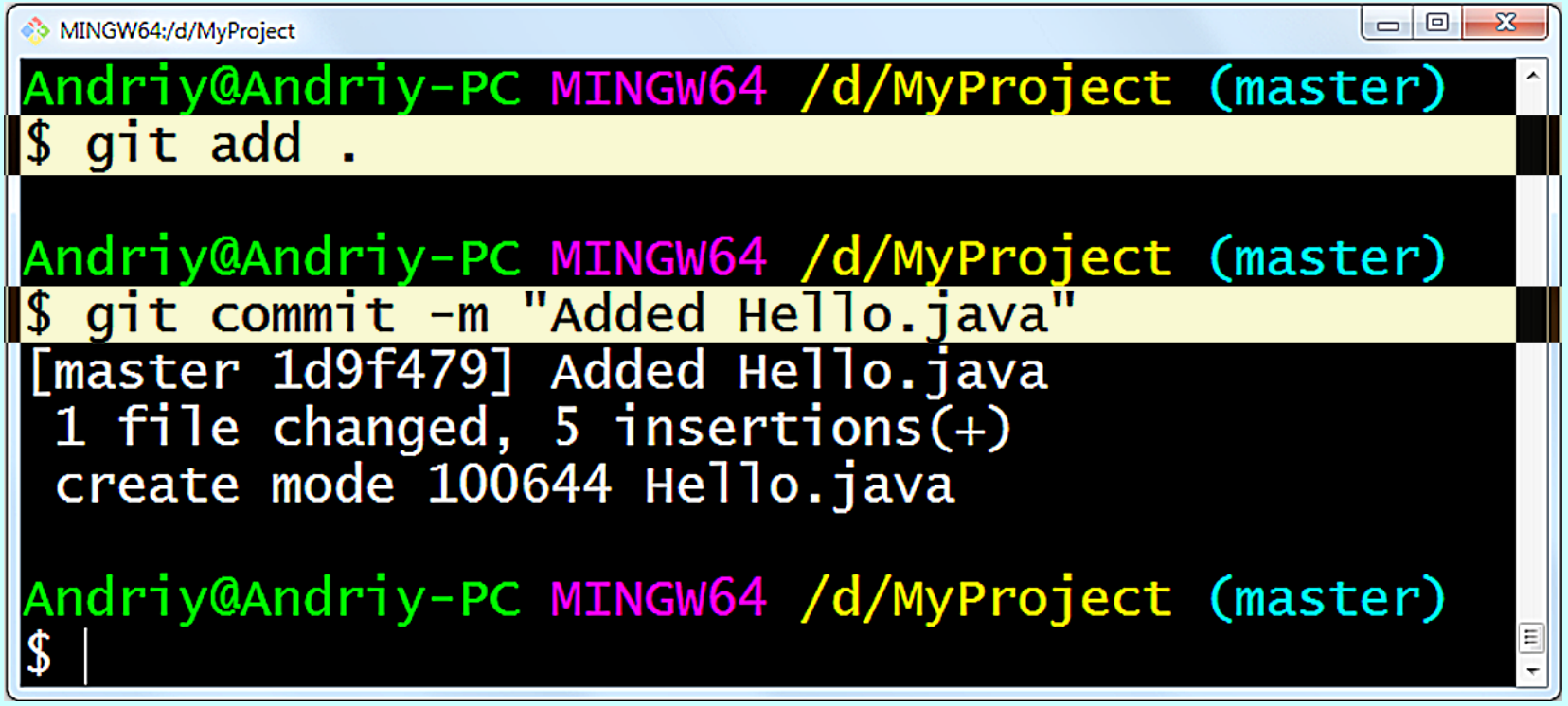


Starting own project (**Log**)

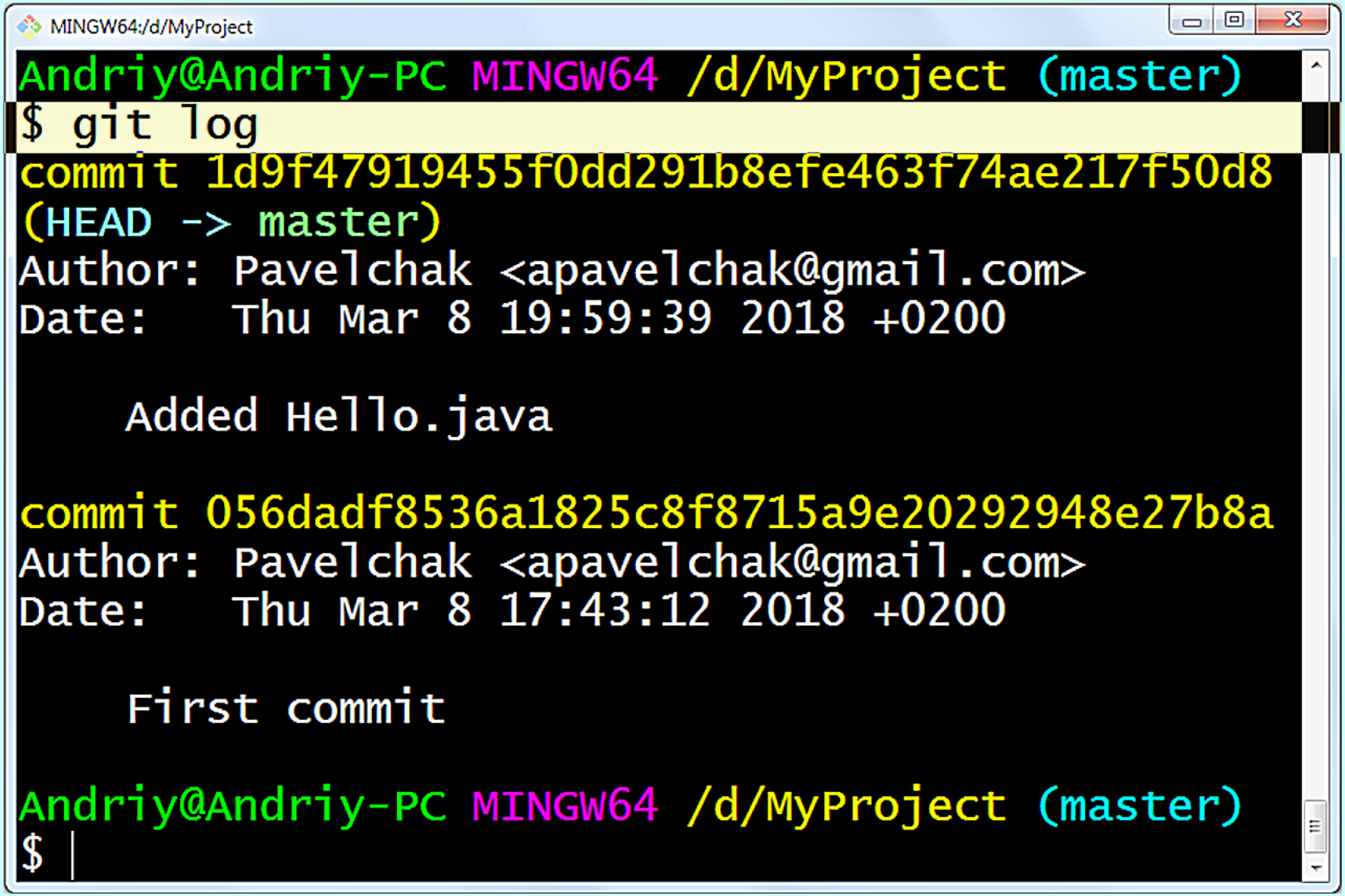


Starting own project (**Add** + **Commit**)





Starting own project (**Log**)



git checkout <*branchname*>

Switches to the specified branch.

git checkout <*commit*>

Detaches the commit and sets to working directory to that snapshot.

git checkout <*branch*>~[*n*]

Reverts the current commit for that branch to the number specified in [n].

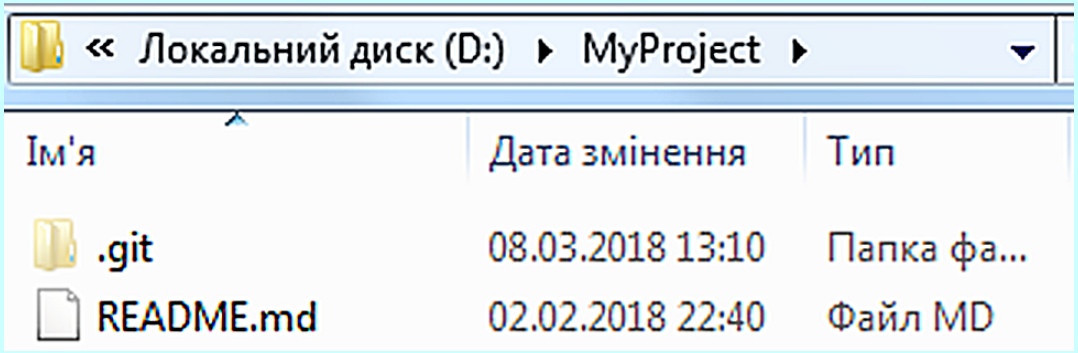
git checkout <*branch*>^

Reverts the previous commit for that branch.

Starting own project (**Checkout**)

MINGW64:/d/MyProject

```
Andriy@Andriy-PC MINGW64 /d/MyProject ((1d9f479...))
$ git checkout 056dad
Previous HEAD position was 1d9f479... Added Hello.ja
va
HEAD is now at 056dadf... First commit

Andriy@Andriy-PC MINGW64 /d/MyProject ((056dadf...))
$
```



Starting own project (**Branch**)

git branch

Displays a list of existing branches and indicates the current branch.

git branch <*name*>

Creates a new branch with the specified name.

git checkout -b <*name*>

Creates a new branch with the specified name and checks it out.

git branch -d <*name*>

Deletes branch.

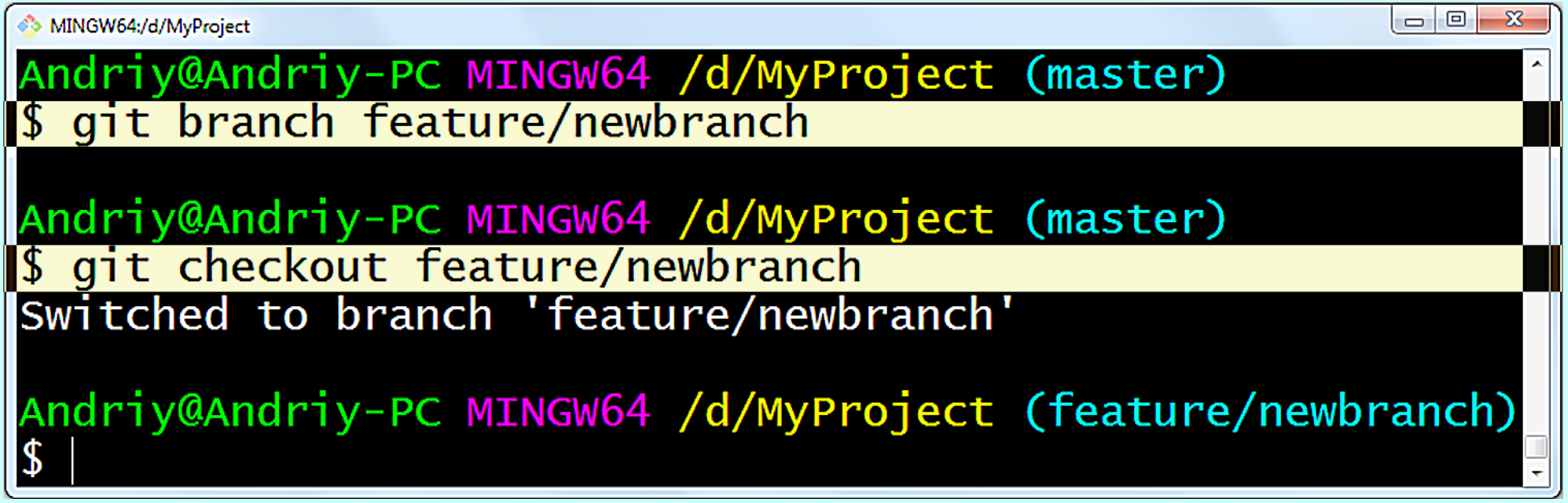
git branch -D <*name*>

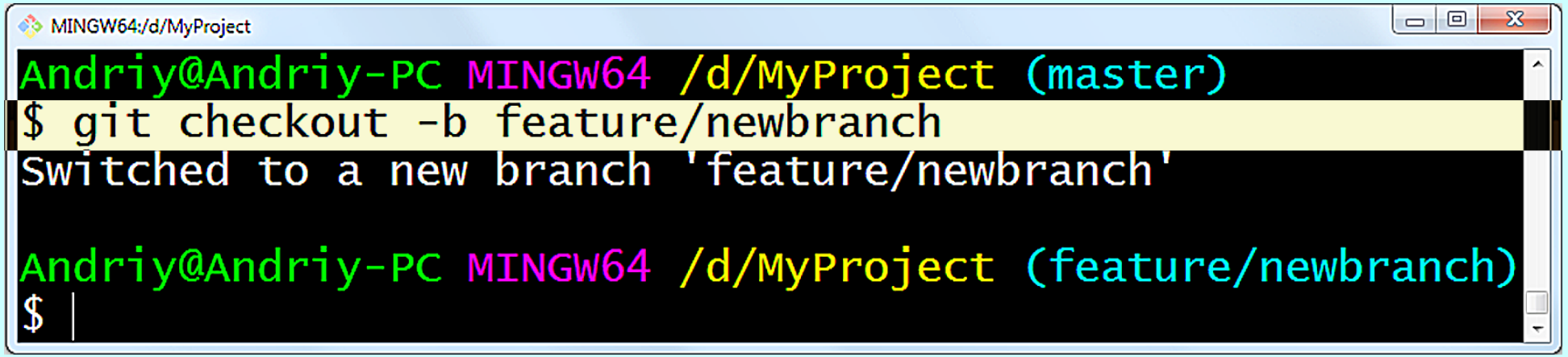
(regardless of its merged status)

git branch -r

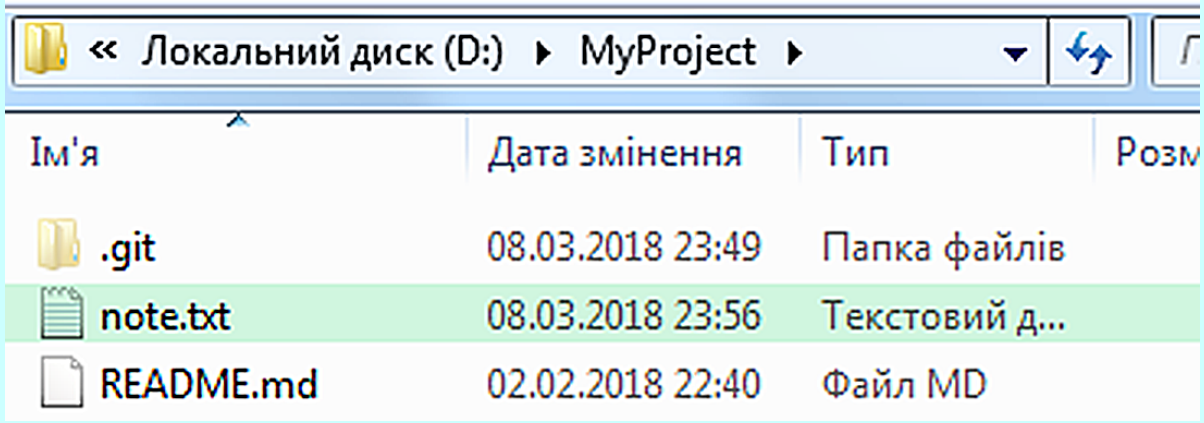
List remote branches.

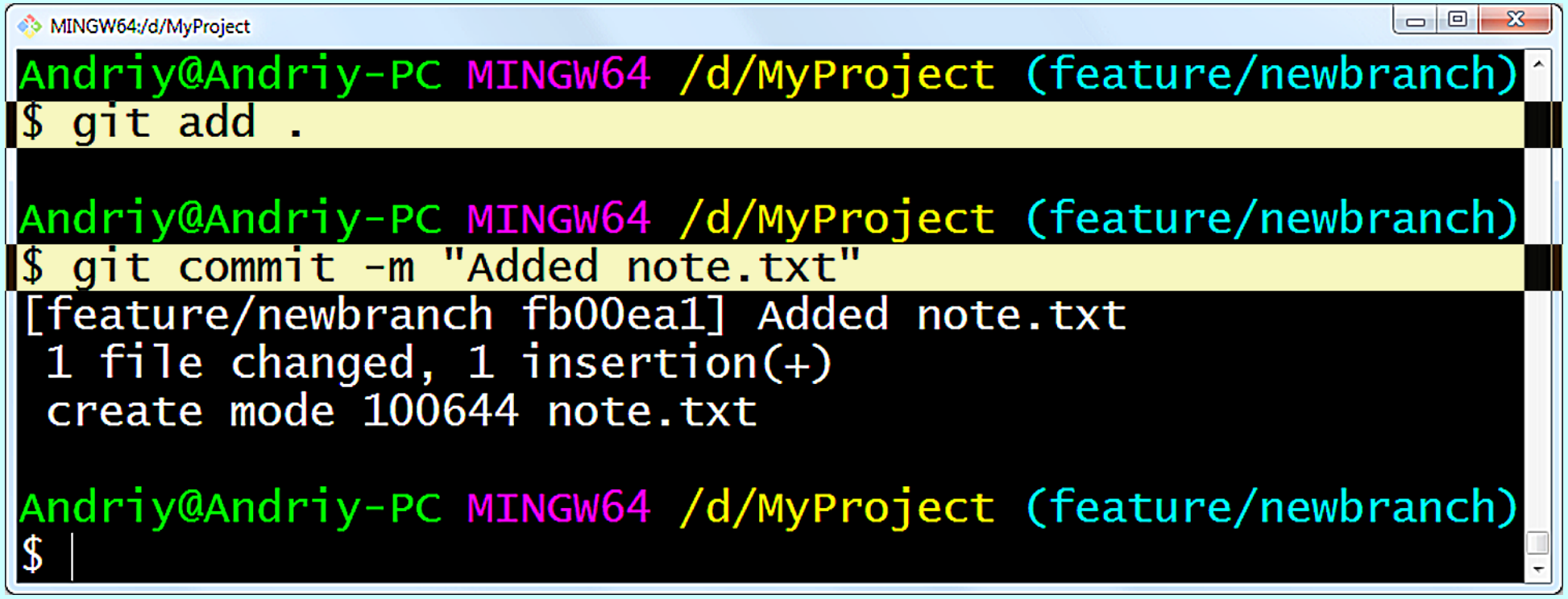
Starting own project (**Branch**)





Starting own project (**Add** + **Commit**)



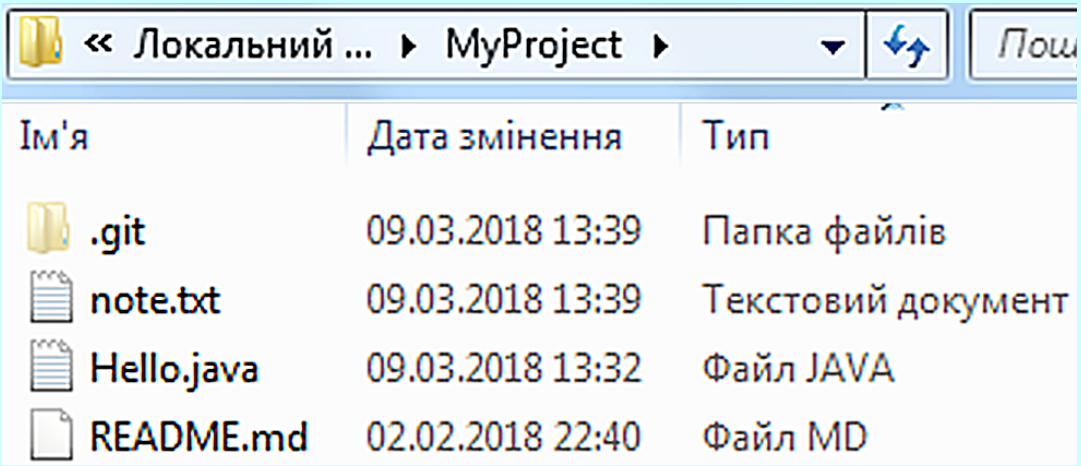


Starting own project (**Merge**)

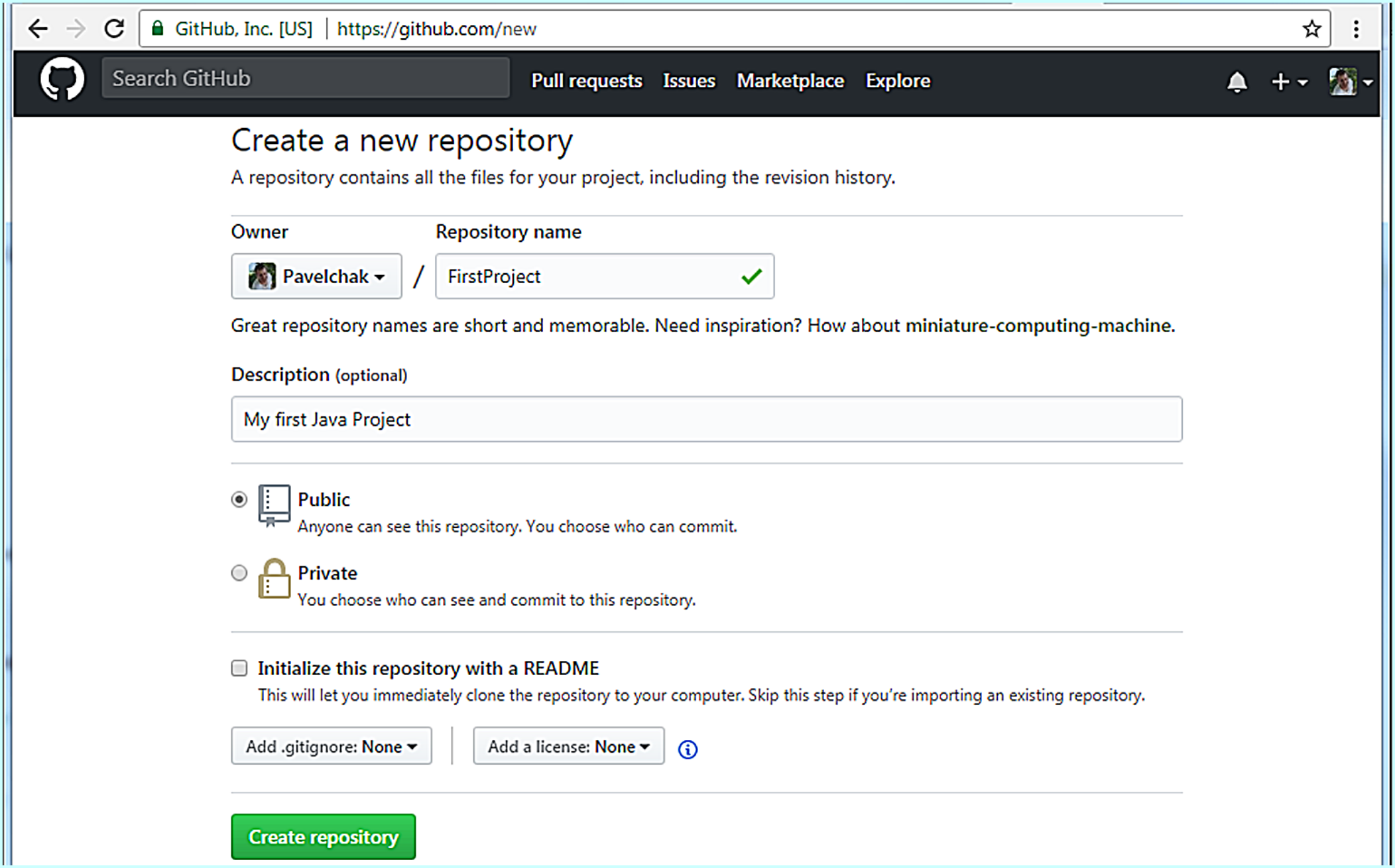

```
MINGW64:/d/MyProject
Andriy@Andriy-PC MINGW64 /d/MyProject ((1d9f479...))
$ git merge feature/newbranch -m "merge f/newbranch"
Merge made by the 'recursive' strategy.
 note.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 note.txt

Andriy@Andriy-PC MINGW64 /d/MyProject ((240333e...))
$
```

Without commentary

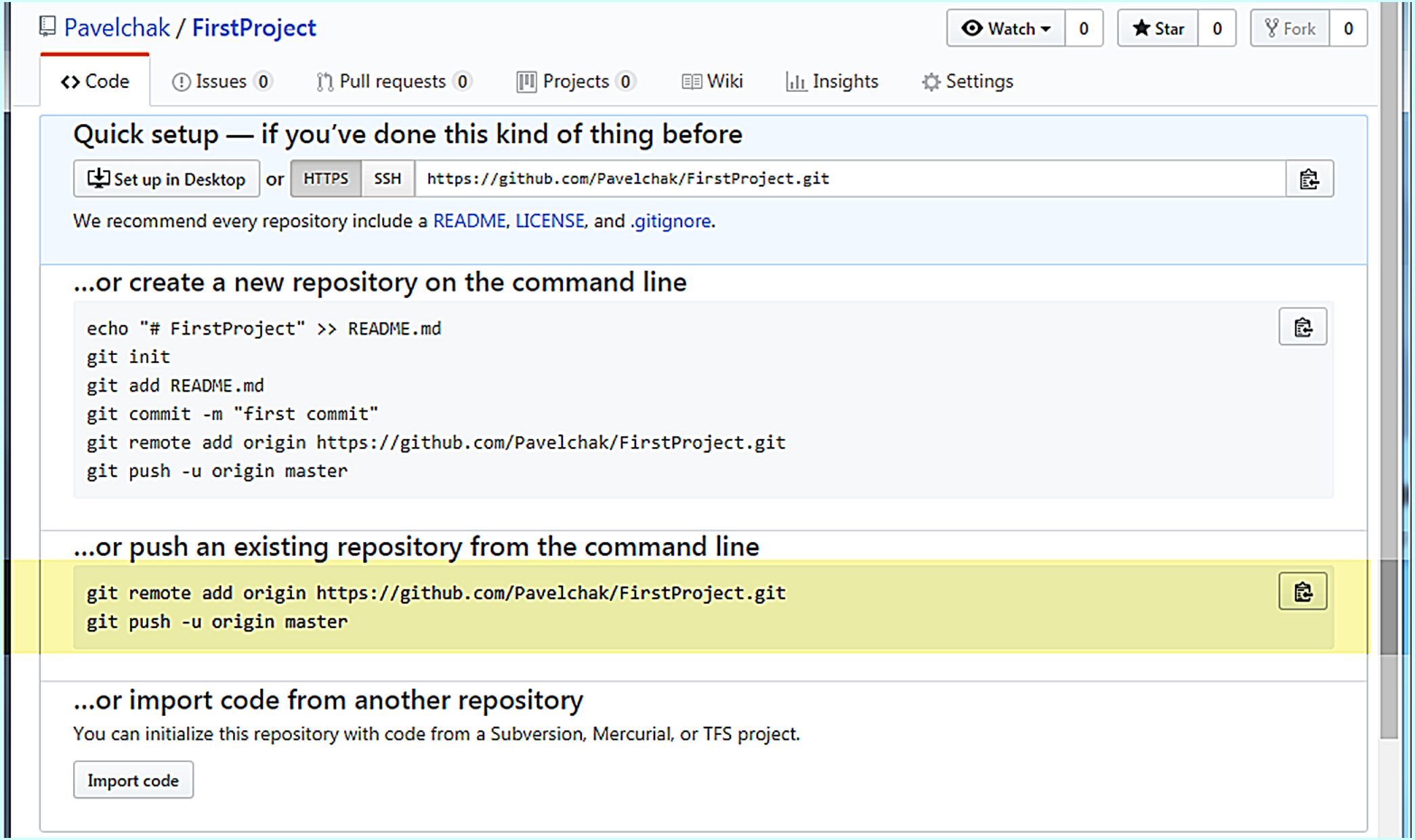
press Esc
write ":q"
press enter



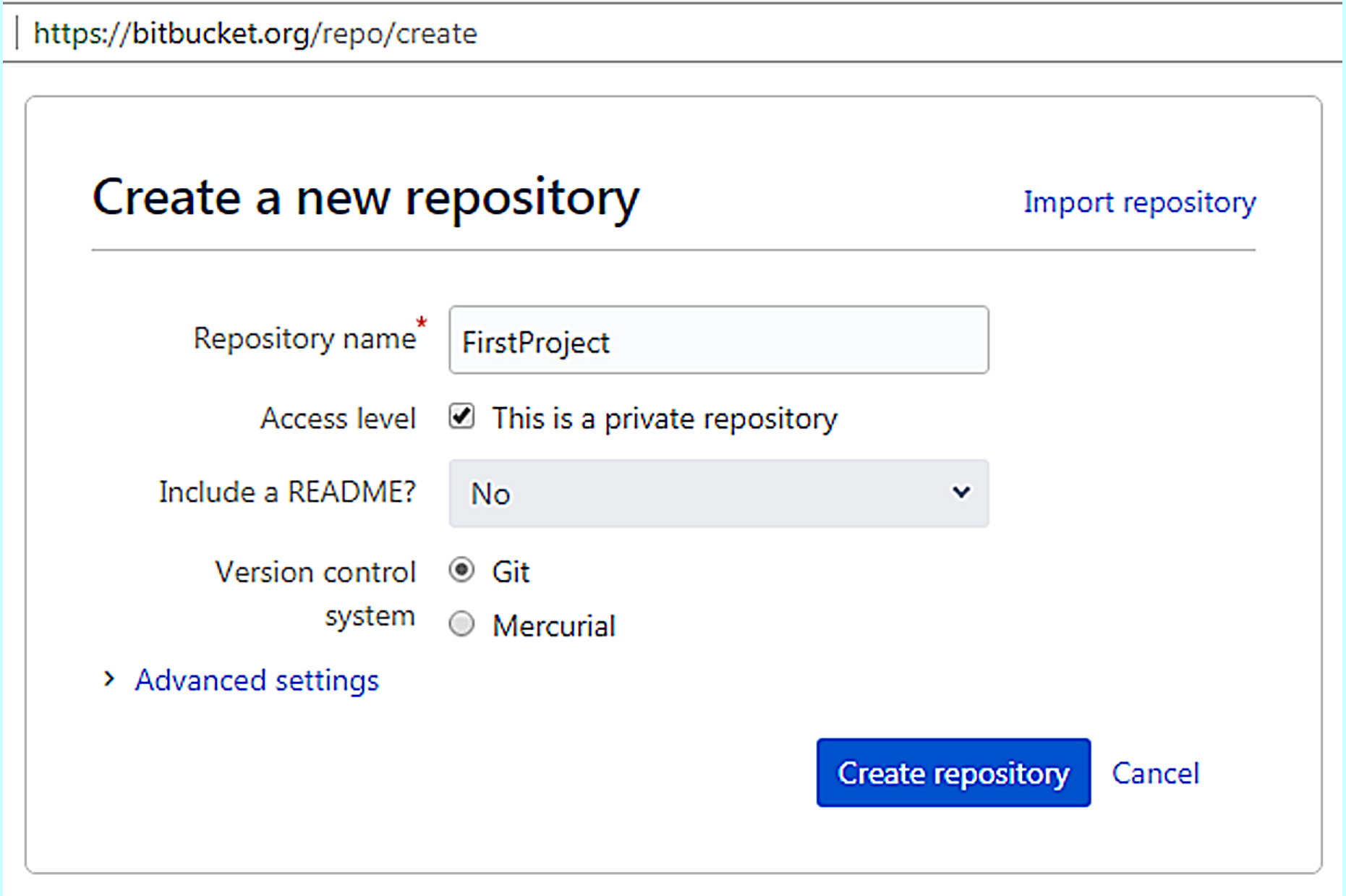
Create a new repository on GitHub



Create a new repository on GitHub



Create a new repository on Bitbucket



Create a new repository on Bitbucket

FirstProject

- Overview
- Source
- Commits
- Branches
- Pull requests
- Pipelines
- Deployments
- Downloads
- Boards
- Settings

Andrii Pavelchak / FirstProject

Source

Code helps me work better

Start by adding a README or pushing up some code. Learn how

Get started the easy way

Creating a README or a .gitignore is a quick and easy way to get something into your repository.

Create a README | Create a .gitignore

Get started with command line

I have an existing project

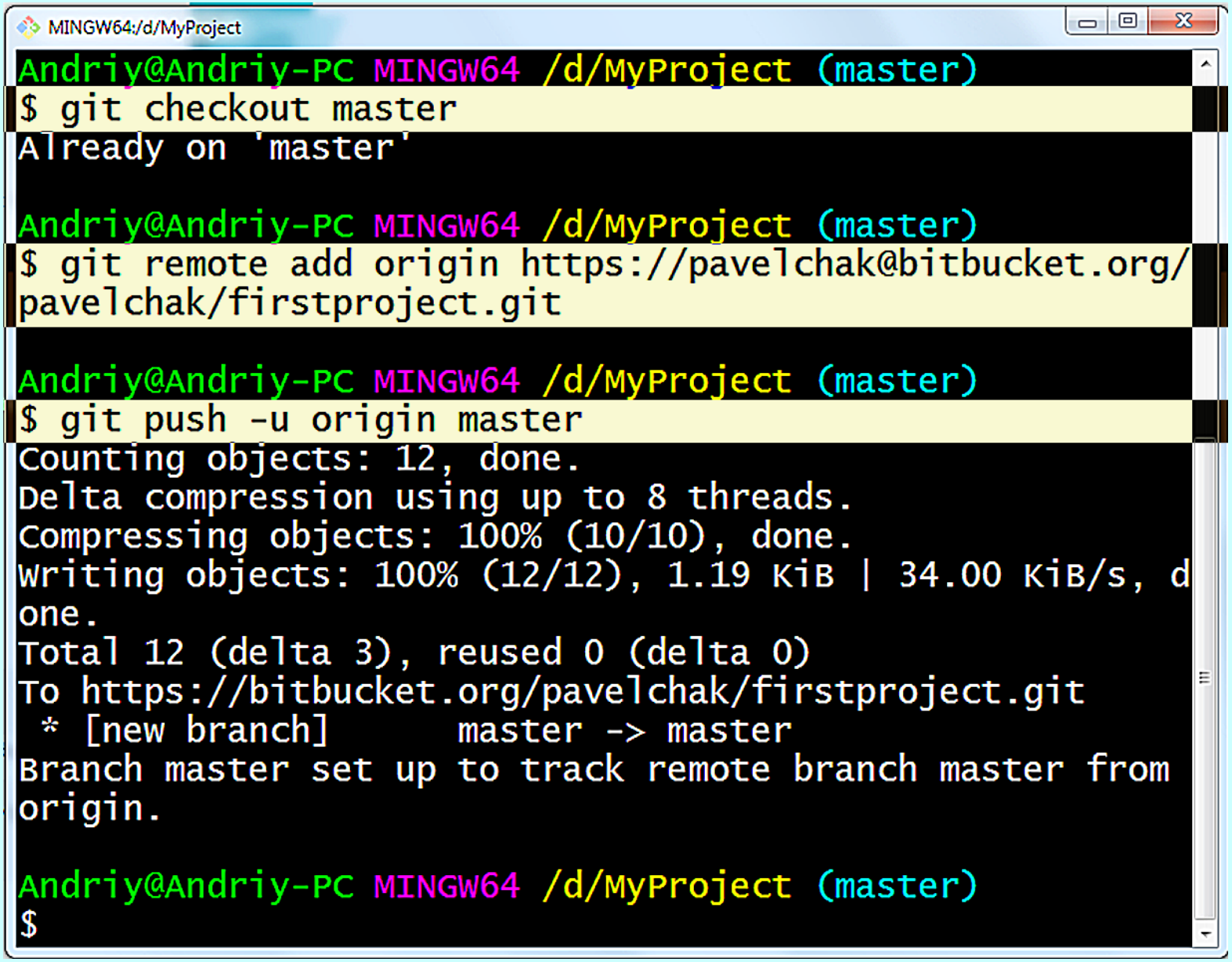
Step 1: Switch to your repository's directory

```
1  cd /path/to/your/repo
```

Step 2: Connect your existing repository to Bitbucket

```
1  git remote add origin https://pavelchak@bitbucket.org/pavelchak/firstproject.git
2  git push -u origin master
```

Starting own project (**Push**)



Starting own project (**Fetch**)

Updates local repositories from remote repos. These changes are not merged by default and must be merged manually. This allows you to pull down remote changes, inspect them, and then choose which changes to update locally.

```
git fetch
```

Will fetch from the remote repository tracked to the current branch.

```
git fetch origin
```

Copies all branches from the remote repo and stores them locally.

Starting own project (**Pull**)

Similar to fetch except for the fact that remote objects are both downloaded and then merged into the local repository. This has the effect of automatically updating the local repository to match the remote, while fetch will download the files but not update your local copies.

```
git pull
```

Pulls the current remote repository and merges it.

```
git pull origin <branchname>
```

Pulls the remote branch specified and merges it.

Starting own project (**Rebase**)

Flattens commit histories into a linear sequence

git rebase <*commit*>

Pulls the current remote repository and merges it.

Moves the current branch to the specified commit. The practical result of this is that it creates a linear commit history from multiple branches. Commits within a branch are moved to the end of the current branch. This is typically followed up by a merge which moves the current commit forward to the last added commit.

The .gitignore File

To ignore files from being tracked and remove them from the untracked file list, create a ".gitignore" file in your project directory, which list the files to be ignored, as follows:

.gitignore

```
# Java class files
*.class

# only ignore Log file in the current directory, not subdir
/project.log

# Can use regular expression, e.g., [oa] matches either o or a
*.[oa]

# ignore all files in the .idea/ directory
.idea/
```

https://try.github.io

Got 15 minutes and want to learn Git?

Task

1. Create an account on bitbucket.org. Add a photo to your profile.

2. Create a repository on bitbucket.org. In a group of 4-5 people, push on bitbucket.org a text with 40 tapes (1 tape at a time).