



# Git Version Control

Text: EN

Software: EN



# Table of contents

1. Getting Started
  - 1.1. Course Objectives
  - 1.2. Agenda
2. Introduction
  - 2.1. What is a Version Control System?
  - 2.2. Revisions
  - 2.3. Generations of Version Control
  - 2.4. Local Version Control
  - 2.5. Centralized Version Control
  - 2.6. Distributed Version Control
  - 2.7. Git
  - 2.8. Online Git Services
  - 2.9. Git Commands Toolbox
  - 2.10. Command Line Interface versus GUI
  - 2.11. Installing Git
3. Basic Commands
  - 3.1. Manual Pages
  - 3.2. Configuring Git
  - 3.3. Initializing a Repository
  - 3.4. Working Tree States
  - 3.5. Repository Status
  - 3.6. Adding Files
  - 3.7. Committing
  - 3.8. Commit Identifiers and HEAD
  - 3.9. Displaying History Logs
  - 3.10. Using Gitk
  - 3.11. Resetting
  - 3.12. Checkout
  - 3.13. Removing Files
  - 3.14. Moving and Renaming Files
  - 3.15. Gitignore
  - 3.16. Stashing
  - 3.17. Specifying Revisions
4. Branches
  - 4.1. What is a Branch?
  - 4.2. Purpose of Branches
  - 4.3. Merging Branches
  - 4.4. Listing Branches
  - 4.5. Creating Branches
  - 4.6. Switching Branches

- 4.7. Detached HEAD
- 4.8. Removing and Renaming Branches
- 4.9. Practical Branching Strategies
- 4.10. Specifying Revisions Revisited
- 5. Merging
  - 5.1. Merging Rationale
  - 5.2. Branch Types for Merging
  - 5.3. Performing a Three-Way-Merge
  - 5.4. Merge Conflicts
  - 5.5. Resolving Merge Conflicts
  - 5.6. Useful Merge Arguments
  - 5.7. Using a Mergetool
  - 5.8. Fast Forwarding
  - 5.9. Octopus Merges
  - 5.10. Other Merge Strategies
  - 5.11. Rebasing
  - 5.12. Rebasing vs Merging
  - 5.13. Squashing
  - 5.14. Interactive Rebase
  - 5.15. Cherry Picking
  - 5.16. Merging Challenges
- 6. Remotes
  - 6.1. What is a Remote?
  - 6.2. Creating a Remote Repository
  - 6.3. Transfer Protocols
  - 6.4. Cloning
  - 6.5. Configuring Remotes
  - 6.6. Remote and Tracking Branches
  - 6.7. Fetching
  - 6.8. Pulling
  - 6.9. Pushing
  - 6.10. Forking
  - 6.11. Pull Requests
  - 6.12. Forking-Pull Requests Overview
- 7. Security
  - 7.1. Remote Access Using SSH
  - 7.2. Signing Commits With GPG Keys
- 8. Extras
  - 8.1. Creating Tags
  - 8.2. Creating Command Aliases
  - 8.3. Filtering Branches
  - 8.4. Creating Patches
  - 8.5. Additional Commands

## 9. References

### 9.1. Useful References

# 1. Getting Started

# 1.1. Course Objectives

Learn the basics of using Git as a Version Control System (VCS)

---

## 1.2. Agenda

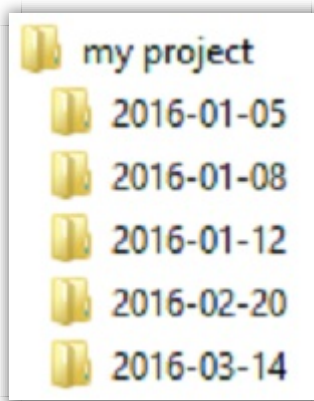
- Introduction
  - Basic commands
  - Branches
  - Merging
  - Remotes
  - Security
  - Extras
-

## 2. Introduction



## 2.1. What is a Version Control System?

- A VCS is a system that records changes to a (set of) file(s) over time so that you can recall specific versions later
- The simplest scheme provides a backup of the project at a point in time
  - What you would 'invent yourself'

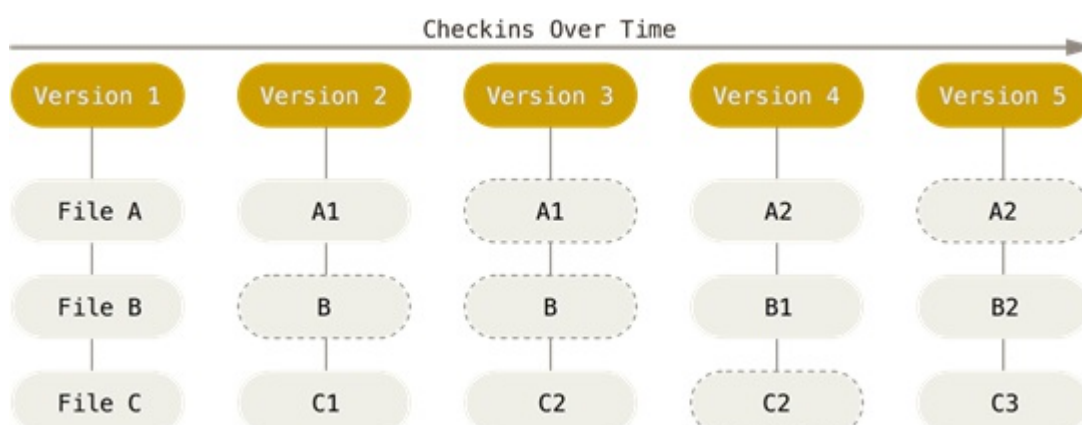


- More advanced schemes offer many additional advantages
  - Sharing work with other people or machines
  - Separating work into separate branches that can use each other's changes
  - More reliable backups

---

## 2.2. Revisions

- Version Control Systems have the concept of a revision
  - Each change is recorded in a graph with a unique ID and is part of a "transaction log"



- Can be done internally either by storing
  - Only the difference between versions (used by Subversion)
  - Storing a complete snapshot of the entire file (used by Git)

## 2.3. Generations of Version Control

- Many VCS exist

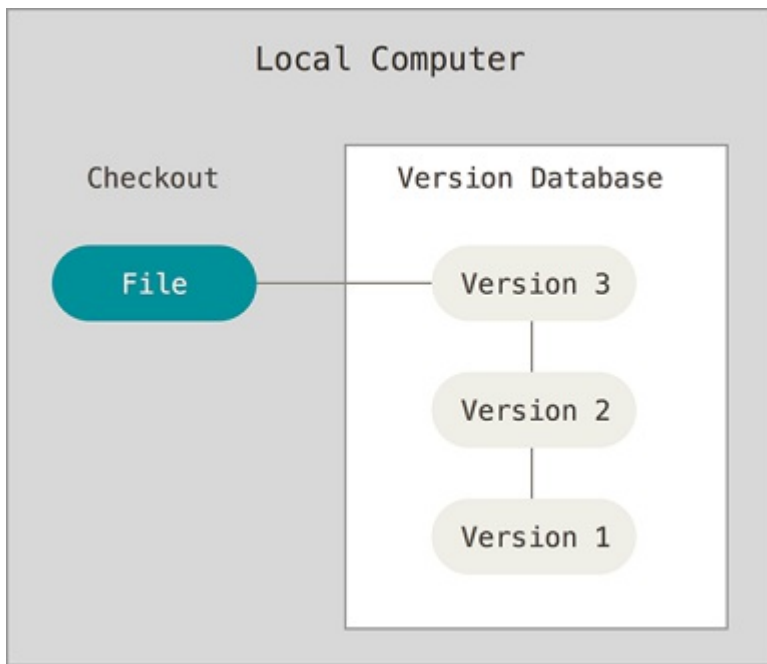


- They can be separated into three generations:
  - 1st: Local
  - 2nd: Centralized
  - 3rd: Distributed

---

## 2.4. Local Version Control

- Locally keeps track of changes
  - Not easy to share work with others
- High risk of losing work
  - Everything is stored on a single machine

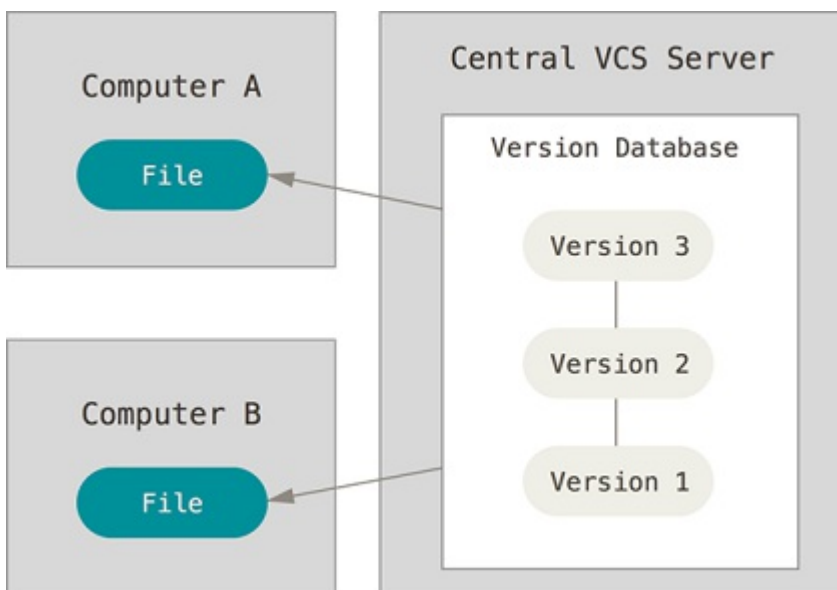


- Example: RCS

---

## 2.5. Centralized Version Control

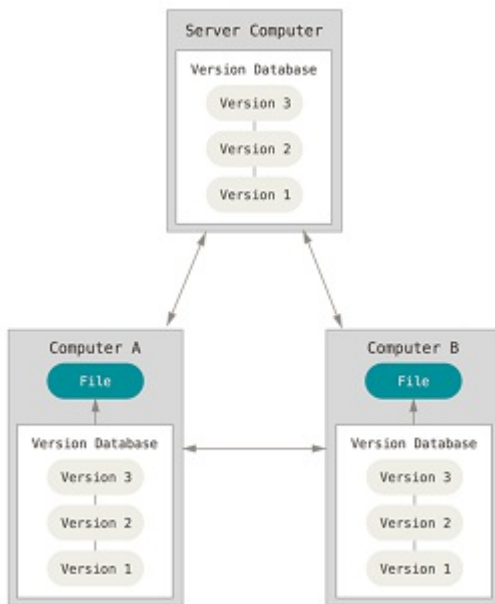
- Tracks changes into a central (shared) database
  - Local machine only has the latest 'snapshot' (working copy)
- Facilitates sharing and syncing work
- Still has a single point of failure



- Examples: CVS, SVN

## 2.6. Distributed Version Control

- Keeps a full history of the entire project on each machine
  - Requires more local storage, but low risk of losing work
- Offers infrastructure to synchronize with others remotely (peer-to-peer)
  - Often combined with a central repository (e.g. GitHub)



- Examples: Git, Mercurial

---

## 2.7. Git

- Git is a modern 3rd generation distributed VCS
  - Free and Open Source software (GPLv2)



- Created by Linus Torvalds to deal with the complexity of the Linux kernel
  - Scales well from small to extremely demanding projects
  - Very popular in the Free and Open Source world
    - But now also being adopted by Microsoft (who actually have their own VCS: TFS)
- Also used as a deployment tool to cloud services
- Mastering Git can be quite challenging

- Complex and has quite a steep learning curve
- You will learn it over time, by using it a lot, and making some mistakes

In case you were wondering where the name Git comes from, Linus Torvalds actually (jokingly) named it after himself: "I'm an egotistical b\*d, and I name all my projects after myself."

git  
noun \ 'git\  
: a stupid or worthless person (especially a man)

---

## 2.8. Online Git Services

- Git benefits from some popular online cloud services
- GitHub (<https://github.com/>)
  - Free for Open Source (public) projects
  - Paying for commercial (private) projects
- BitBucket (<https://bitbucket.org/>)
  - Free for both Open Source and commercial projects
  - Paying for more advanced cases (support for many users)
- GitLab (<https://about.gitlab.com/>)
  - Has a community edition which itself is FOSS

---

## 2.9. Git Commands Toolbox

- Always remember that git is an advanced piece of software
- Many commands will be used to control git
- Most of these commands serve multiple purposes, depending on the situation they are used in
  - You can compare this with a toolbox
    - For example: a hammer can be used for many things, it is impossible to produce a complete list of things you might ever want to use a hammer for

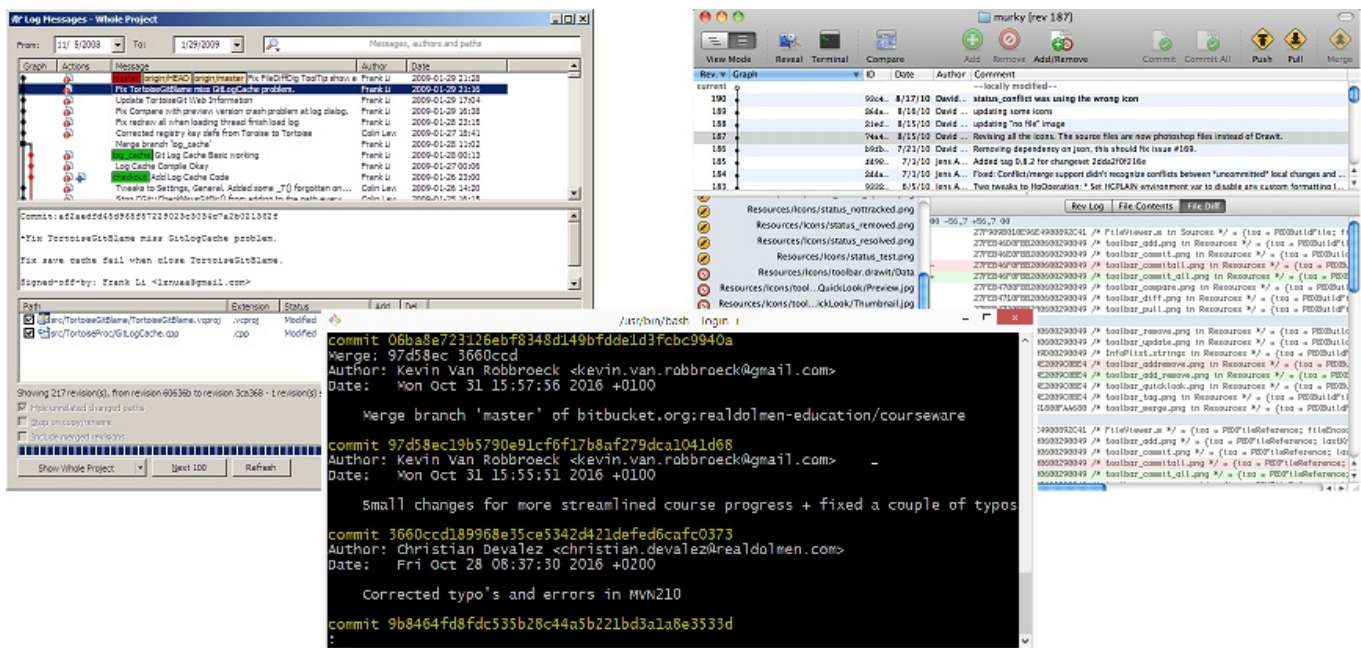


- We will show some typical ways to use some of the Git commands to help you get

started

## 2.10. Command Line Interface versus GUI

- It is important to understand that to master Git, the most important thing is to learn to use the command line interface
  - Afterwards, the GUI tools just make it more convenient to work with
    - TortoiseGit, SourceTree, Eclipse, Team Explorer, ...



## 2.11. Installing Git

- Git can be downloaded from <https://git-scm.com/download> and is available for all major platforms

# Downloads

Older releases are available and the Git source repository is on [GitHub](#).

Latest source Release

## 2.10.2

Release Notes (2016-10-28)

Downloads for Windows

```
$ git --version  
git version 2.8.1.windows.1
```

- A wide range of GUI clients is also available:
  - <https://git-scm.com/downloads/guis>

Note that all GUI tools are just using the GIT commands behind the scenes. They all work similarly, but many are restricted to only a single platform. For this reason in-depth coverage of any of the GUI tools is considered out-of-scope for this course.

---

## 3. Basic Commands



## 3.1. Manual Pages

- Git has a large amount of commands, each having an enormous amount of arguments
  - It's impossible to cover (or even know) them all
- When unsure how to use a command, check its manual pages

```
$ git rebase --help          # Works for any command
```

### git-rebase(1) Manual Page

#### NAME

git-rebase - Reapply commits on top of another base tip

#### SYNOPSIS

```
git rebase [-i | --interactive] [options] [--exec <cmd>] [--onto <newbase>]
           [<upstream> [<branch>]]
git rebase [-i | --interactive] [options] [--exec <cmd>] [--onto <newbase>]
           --root [<branch>]
git rebase --continue | --skip | --abort | --edit-todo
```

## 3.2. Configuring Git

- Git stores its configuration in 3 levels
  - More specific levels override more general levels

Level	Scope	Location
System	Per machine	In %GIT_HOME%/etc/gitconfig (Win) or /etc/gitconfig (Linux)
Global	Per user	In the user's home directory ~/.gitconfig
Local (default)	Per project	Inside project's .git/config

- Configuration settings can be listed using `git config --list`

```
$ git config --list          # Shows all levels
$ git config --list --show-origin # Also shows from which config file the settings
comes
$ git config --list --global   # Shows only global level (other options are --
system and --local)
```

```
# Example output:
```

```
diff.astextplain.textconv=astextplain
rebase.autosquash=true
user.name=Jimi Hendrix
user.email=jimi.hendrix@mail.com
core.autocrlf=true
```

Actually there is also a fourth level: per file, which is not very useful for most cases. The most practical ones are global and local (marked bold)

- Configuration can be edited using `git config`
  - Alternatively you can edit the files directly (dangerous)

```
$ git config --global --add user.name "John Doe"      # Add config property on global
level (multivalue)
$ git config --local user.email john.doe@example.com # Set or replace config property
on local level
$ git config --unset --local user.signingkey          # Removes user.signingkey from
local level
```

- There are many configuration properties to configure
  - The most important ones (for now) are `user.name` and `user.email`
    - This will make sure Git knows who you are when you interact with it
    - Always make sure you set them (either `local` or `global`)
  - Many more config properties can be set:

```
# Example to configure alternative text editor from committing changes (see later)
$ git config --global core.editor "'C:/Program Files/Notepad++/notepad++.exe' -
multiInst -notabbar -nosession -noPlugin"
```

---

## 3.3. Initializing a Repository

- Any directory can be initialized with a git repository using `git init`
  - Creates a subdirectory called `.git/`

```
$ mkdir my-project
$ cd my-project      # Make sure you run git init inside a directory (may be
existing one)!
$ git init
Initialized empty Git repository in ~/my-project/.git/
```

- Over time, the `.git/` directory will contain the entire project history
  - Everything is kept locally (don't remove it or you'll lose all local changes)
  - Do not manually edit or remove this directory's contents!
- Next to the `.git/` directory there will be the project's resources called work tree or working copy

- Contains the contents (if any) of the project of a specific version (defaults to latest)
- `git init` does not save the work tree
  - Still needs to be added later
- A bare git repository can be created when you don't want a work tree
  - There will be no `.git/` directory; history is directly inside project itself

```
mkdir shared-project.git    # By convention, bare (sharing) repositories end with
.git
cd shared-project.git
git init --bare
```

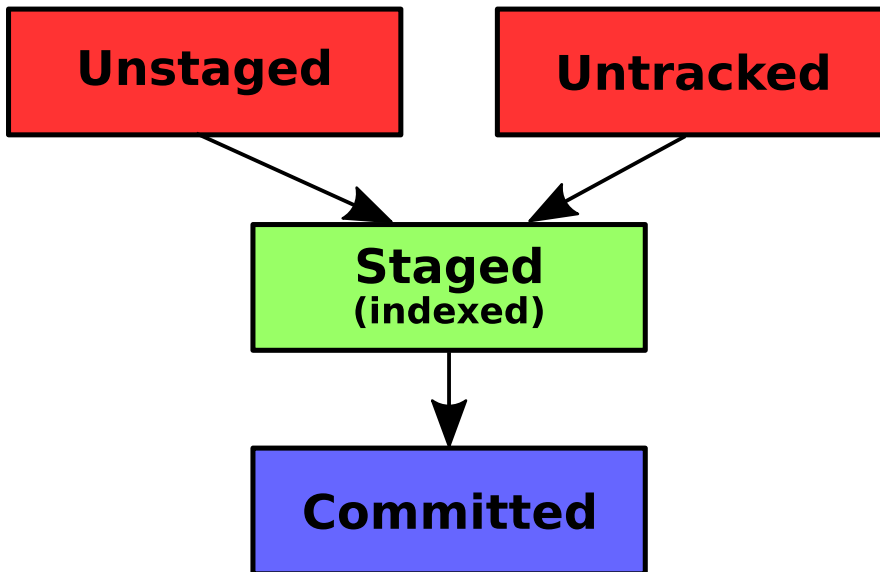
- Useful if your repository is intended to be shared with others, rather than to be worked upon directly
  - For example: repository will be hosted on a separate server or shared drive
  - Synchronizing with a shared repository will be covered later

---

## 3.4. Working Tree States

- There are 4 states that file changes in the working tree can be in
  - E.g. creation, modification and deletion of files; not the files themselves

State	Description
Untracked	Newly added files not previously known to Git (shown red)
Unstaged	Changes on a file are detected, but will not be included in the next commit (shown red)
Staged (aka Indexed)	Changes on a file are detected and will be included in the next commit (shown green)
Committed	The file is saved into Git and has no changes since. It is considered 'clean'.



It's very important to understand that git keeps track of changes within a file. It is not the file in total that is tracked, one file can have multiple changes, each of which can be tracked separately by Git.

- Saving a number of changes permanently into Git's history is called committing and is one of the key actions to perform
  - Staging area can be seen as a "loading dock" for changes to be included in the next commit
- A variety of git commands are available to view and manipulate the state in which a given change resides
  - Some have multiple uses depending on additional arguments

Command	Purpose
git status	Shows the status of all changes in the working tree (use often!)
git add	Adds an unstaged change to the staging area (does not commit it)
git rm	Removes files from the working tree and/or from staging area
git checkout	Reverts a change back to the version that was last committed
git reset	Moves a staged change back to unstaged
git commit	Saves all staged changes to the git repository's local history (inside <code>.git/</code> )

## 3.5. Repository Status

- Use `git status` to see an overview of the current state of all changes
  - You also get a hint on what commands you might want to execute next

```
$ git status
```

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

deleted: tiramisu.md

modified: spaghetti.md

Changes not staged for commit:

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

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

modified: tomato-soup.md

Untracked files:

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

banana-pie.md

- Staged changes show up in green, while unstaged and untracked changes show up in red
  - Committed states are not shown
- `git status` has some interesting arguments

Command	Meaning
<code>git status --short</code>	Shows a more compact output (without hints)
<code>git status --verbose</code>	Also shows what changed in each file

```
$ git status --short
M tomato-soup.md      # Modification unstaged
D tiramisu.md         # Deletion staged
M spaghetti.md       # Modification staged
?? banana-pie.md     # Untracked
```

- Use `git status` a lot!
  - It helps tremendously in learning how Git works
  - It prevents you from making mistakes

## 3.6. Adding Files

- The `git add` command will promote any change from untracked or unstaged to staged
  - This means the change will be included in the next commit
  - By default, changes are not staged, they need to be added explicitly

```
$ git add spaghetti.md # Assume spaghetti.md is unstaged (red)
$ git add banana-pie.md # Assume banana-pie.md is untracked (red)
$ git status
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   spaghetti.md      # Shown in green
    new file:   banana-pie.md
```

- Some other forms of using this command (and many others)

```
$ git add *.txt      # Using wildcards is possible
$ git add --all      # Adds all currently unstaged and untracked changed (all red to green)
```

---

## 3.7. Committing

- Saving changes to the repository's history is done using `git commit`
  - Actually makes sure you persist your changes
  - Only staged (green) changes are committed by default
    - To also commit other changes, add the `--all` argument

```
git commit --message "Added some new recipes"      # Commits with message (always add a message!)
git commit --all --message "A little less pepper"  # Also includes red changes in the commit
```

- Unlike other VCS, this does not send it to a remote server
  - Only resides in project's `.git/` directory; no safety against HDD crash, etc...
- Use `--amend` argument to combine commit with previous one
  - Often used when you forgot to add something or made silly mistake

```
# Will be added to previous commit:
$ git commit --message --amend "A little less pepper and a little more salt"
```

Note: If you forget to pass the `--message` argument, the default configuration of git is to show you a text-editor (probably vim). To exit this program press `ESC` followed by `:q!`. It's a Linux thing :-)

---

## 3.8. Commit Identifiers and HEAD

- Each version committed to the repository's history receives a unique commit identifier
  - Calculated as a hash chain based on current changes and previous commit id
  - In most cases the first few characters are already unique, so you can use the short commit identifier instead

```
# Same thing:
Long commit id:      37df032099ce13a583c1c7f92fa2f88bc21acc3a
Short commit id:     37df032
```

- A reference called `HEAD` points to the latest commit (of current branch)

- Commit identifiers are used with various commands

## 3.9. Displaying History Logs

- Use `git log` to see what's inside the repository's history

```
$ git log
commit 39f85969fd5f1d94ae80eb1ece870aae063a7b37
Author: Jimi Hendrix <jimi.hendrix@mail.com>
Date:   Wed Nov 9 14:53:21 2016 +0100
    Added healthy hamburger recipe
```

- There are several interesting arguments to shape the output

Argument	Purpose
<code>--follow</code>	Also shows history for a single file only
<code>--stat</code>	Also shows summary of changed files
<code>--graph</code>	Shows 'railways' (useful when dealing with branches)
<code>--oneline</code>	Shows summary only (one line per commit)
<code>--decorate</code>	Shows information about branches and remotes (use <code>--decorate=full/short</code> to tune)
<code>--all</code>	Displays also other branches (instead of only the current one)
<code>-p</code>	Shows the actual changes in each file (intended to create patches)

```
$ git log -p
commit 10bb5375cce065f0e8aaa3efdae5f0a52bedc5c1
Author: Jimi Hendrix <jimi.hendrix@mail.com>
Date:   Wed Nov 9 15:16:30 2016 +0100

    A little less pepper and a little more salt

diff --git a/tomato-soup.md b/tomato-soup.md
index 3aa792f..748efd2 100644
--- a/tomato-soup.md
+++ b/tomato-soup.md
@@ -1,3 +1,3 @@
    Make some soup

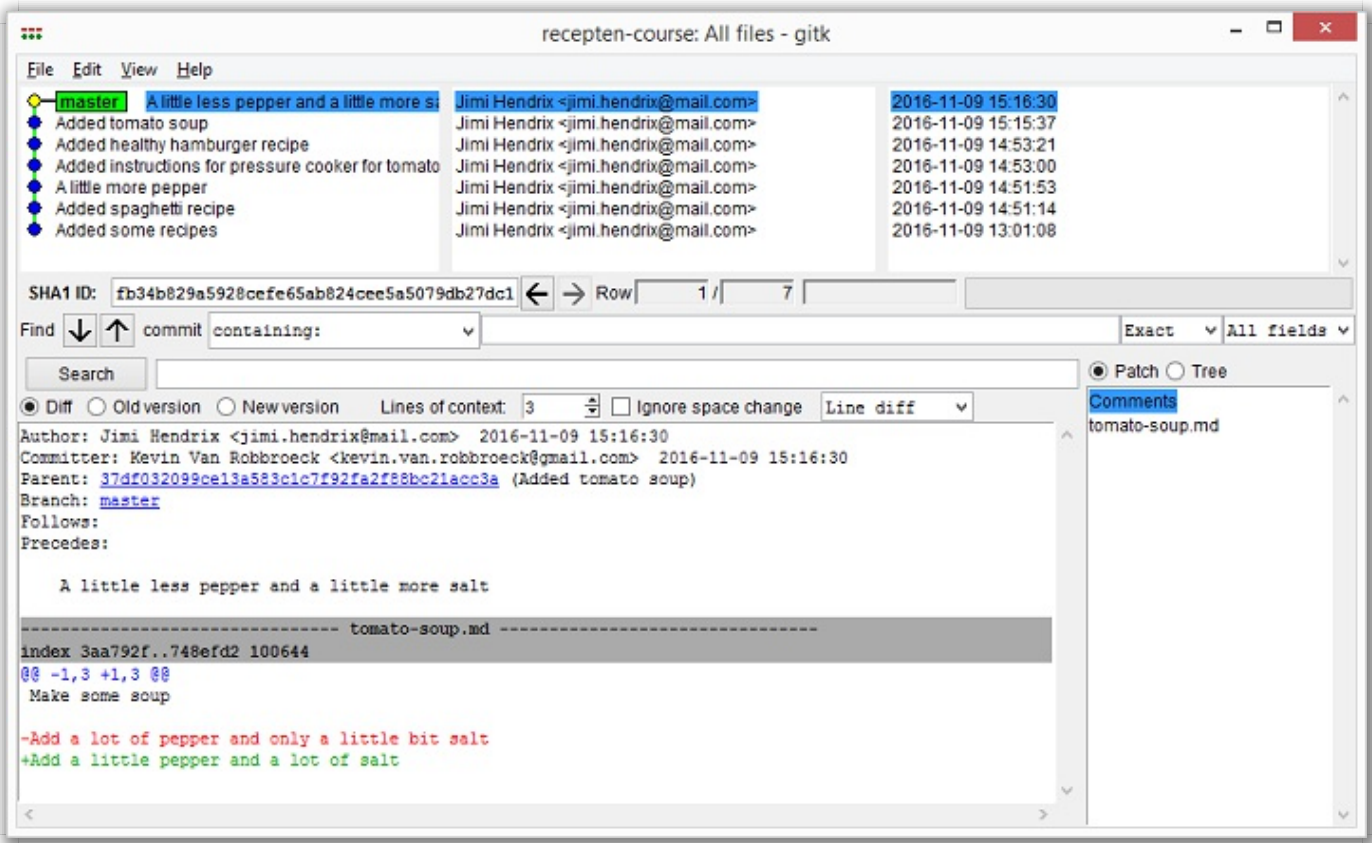
-Add a lot of pepper and only a little bit salt
+Add a little pepper and a lot of salt
```

```
$ git log --oneline
10bb537 A little less pepper and a little more salt
e53e903 Added tomato soup
39f8596 Added healthy hamburger recipe
596b3f2 Added instructions for pressure cooker for tomato soup
cda547d A little more pepper
```

```
8d3166c Added spaghetti recipe
d81e67b Added some recipes
```

## 3.10. Using Gitk

- Using the command line interface to explore history can be tedious
  - Alternatively use the built-in `gitk` command (add `--all` for all branches)



## 3.11. Resetting

- The `git reset` command has two important forms of use
  - Move a staged change back to unstaged or untracked
    - This is opposite of `git add`
    - Does not change the working tree (e.g. files on disk)
  - Reset the repository's HEAD (most recent version in history) back to a previous version
    - Possibly also changing the staging area (index) and/or working tree (e.g. files on disk)
    - Depends on arguments like `--soft` and `--hard`
- First form: use `git reset <path>` to evict staged changes



```
$ git reset spaghetti.md      # Assume spaghetti.md is staged (green)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   spaghetti.md    # Shown in red
```

- This is effectively the opposite of what `git add <path>` would do
- No risk of losing work
  - No changes in the working tree (on disk) are affected, only removes from staging
- Second form: use `git reset <mode> <commit>` to move the HEAD back to a previous version
  - Mode argument controls what happens to working tree and/or staging

Mode	Result
--hard	Removes all changes. Warning: You may lose work!
--soft	Keeps reverted changes in staging (green)
--mixed (default)	Keeps reverted changes in untracked or unstaged (red)

- Some typical use cases

```
# Undo the last commit(s):
$ git reset --hard HEAD~      # Throw away last commit
$ git reset --soft d81e67b    # Undo all up to given commit (keep everything green)

# Revert (and throw away) all staged (green) local changes from working copy
# Does not remove untracked files nor unstaged files (red)
$ git reset --hard HEAD
```

## 3.12. Checkout

- Use `git checkout` to restore the working tree to any previous version

```
$ git checkout ac36590 hamburger.md # Checkout the file hamburger at given commit id
$ git checkout HEAD *.md            # Checkout all .md files to latest version
(override local changes)
$ git checkout HEAD *                # Restore everything to the latest version
$ git checkout -- tiramisu.md        # Sometimes needed to disambiguate path names
from branches or args
```

- There is some overlap with `git reset` in certain use cases
  - Keep in mind that `git checkout` is about changing the working tree (changing actual files) while `git reset` is about changing only the index (staging)

- Checkout also has a second form of use concerning branches, which is covered later

---

## 3.13. Removing Files

- To stage a file for removal use the `git rm` command
  - Has some interesting arguments

Argument	Purpose
<code>--cached</code>	Stages the file for deletion (green) + keeps the file around as untracked
<code>-r</code>	Recurse to subdirectories if the specified name is a directory

```
$ git rm hamburger.md           # Removes hamburger.md and makes that change staged (green)
$ git rm --cached hamburger.md  # Keeps hamburger.md but stage it for removal (green)
```

- Do not confuse this with different things:
  - A regular delete would remove the file and make that change unstaged
  - Staging for removal is not at all the same as unstaging

```
$ rm hamburger.md              # Removes hamburger.md and makes that change unstaged (red)
$ git reset hamburger.md       # Unstaged (make red) a previously staged change (green) on hamburger.md
                                # Note: if hamburger.md is not even changed, this is a no-op
```

---

## 3.14. Moving and Renaming Files

- Renaming (and moving) files is done with `git mv`
  - The rename will be staged for commit

```
$ mkdir soups
$ git mv tomato-soup.md soups/tomato.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:    tomato-soup.md -> soups/tomato.md    # Shown in green
```

- Do not move or rename using filesystem commands
  - They will be registered as a deletion + addition (looses link in history)

## 3.15. Gitignore

- Some files in a project may always need to be excluded from committing
  - They mustn't become part of the project's history
  - It is impractical to always ignore them and leave untracked (red)
- For this git has the `.gitignore` file
  - Contains a list of files/directories that must be ignored by git

```
.project          # Ignore project metadata file (used by Eclipse)
.classpath        # Another Eclipse metadata file
build/           # Ignore temporary build folders
```

- These entries will never be shown in `git status`

## 3.16. Stashing

- To clean your working tree without losing any work, use `git stash`
  - Saves your working changes on a stack, which you can later pop
  - Very useful when 'something urgent comes up'

```
$ git stash      # All changes are stored on top of stack, working copy clean
$ git status
On branch master
nothing to commit, working directory clean

# Some time later...

$ git stash pop # Restores all work, ready to continue
```

- Some other options

Option	Meaning
<code>git stash list</code>	Shows the contents of the stash (how many frames)
<code>git stash apply</code>	Same as pop, but does not remove from top (e.g. 'peek')
<code>git stash drop</code>	Remove top without applying
<code>git stash clear</code>	Remove entire stash

## 3.17. Specifying Revisions

- It is sometimes inconvenient to work with commit id directly
  - They must be looked up before use :-)

- Git provides a special syntax `COMMIT~n` to navigate over the history

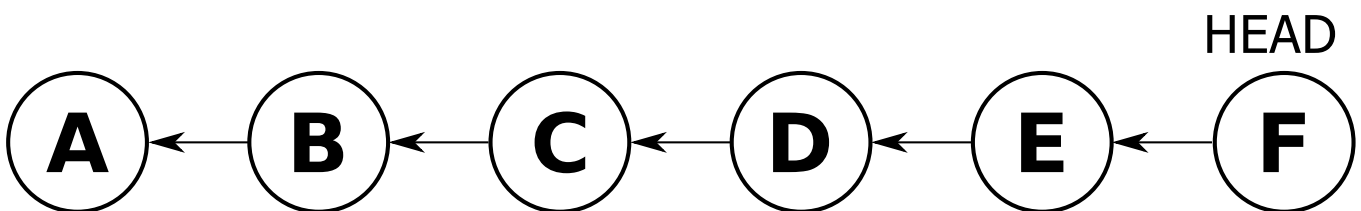
Syntax	Meaning
<code>COMMIT~5</code>	Selects the 5-th ancestor (parent's parents) relative to <code>COMMIT</code>
<code>COMMIT~</code>	Shorthand for <code>COMMIT~1</code>
<code>COMMIT~0</code>	Alias for <code>COMMIT</code>
<code>:/Hello</code>	Selects youngest revision that has commit message starting with <code>Hello</code>

- Use as follows

```
git checkout d3ab7cb~~~~ # 4 versions up from d3ab7cb
git checkout d3ab7cb~4   # Same as previous
git reset HEAD~          # The before-last commit
git checkout HEAD~0      # HEAD itself
git checkout :/Fixed bug XYZ # Commit with given commit message
```

- Some visual examples (pretend letters are actually commit ids)

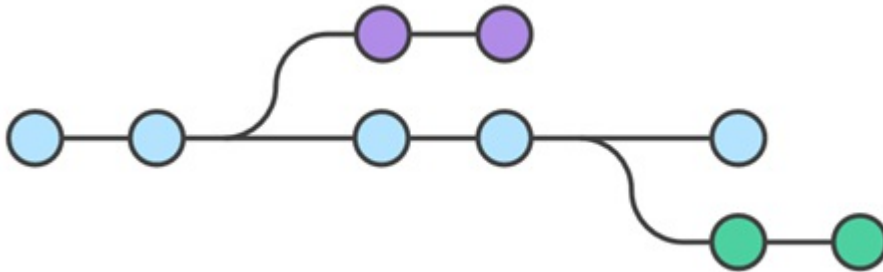
Expression
<code>C = F~3</code>
<code>F = HEAD~0</code>
<code>B = E~~~</code>
<code>D = E~1</code>
<code>A = HEAD~5</code>



## 4. Branches

## 4.1. What is a Branch?

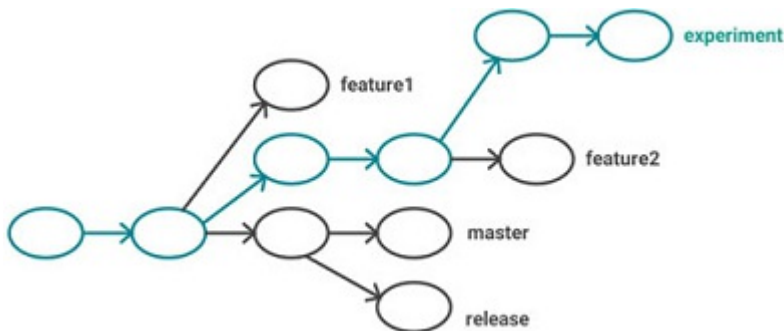
- A branch is a point in the history of a project where two (or more) parallel tracks are created
  - Each of these tracks can evolve separately



- Think of it as a "parallel universe" or an "alternate reality"
  - There can be multiple latest versions at the same time

## 4.2. Purpose of Branches

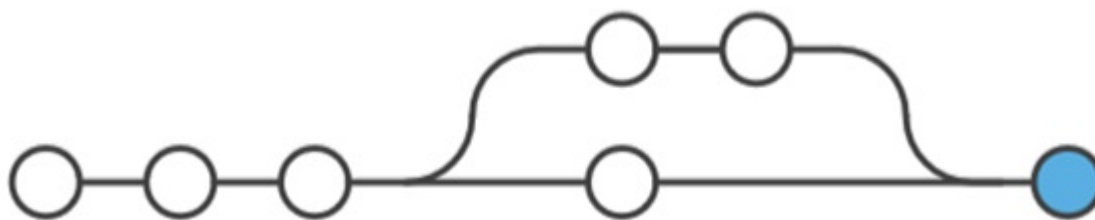
- During development we often create (fork) branches for various reasons
  - To allow developers to work on different parts of the project without conflicting with each other
  - To separate different editions of the application from each other
    - E.g. production, development and experimental code
  - To be able to easily switch the working tree when required to work on different parts simultaneously



- Projects will have to carefully decide on a strategy that fits their needs
  - This is why you often encounter software releases as 'alpha', 'beta', 'nightly', etc...

## 4.3. Merging Branches

- Branches can (optionally) be joined back together later, which is called merging



- This can cause problems
  - Since both branches can develop their own "alternate reality", some of these may conflict with each other
    - Imagine you die in one reality, but you marry and get children in an other one. How would these combine back into a canonical reality?
- We'll take a look at how to solve these problems in a dedicated chapter

## 4.4. Listing Branches

- Use `git branch` to display the branches currently in the repository

```
$ git branch
  experiments
* master          # The star marks the branch you are currently on (green)
  release
```

- Every git repository has at least one default branch called master
  - This branch will often be used to aggregate all work done by the entire team
  - So far we've been working on the master branch
- Some interesting arguments

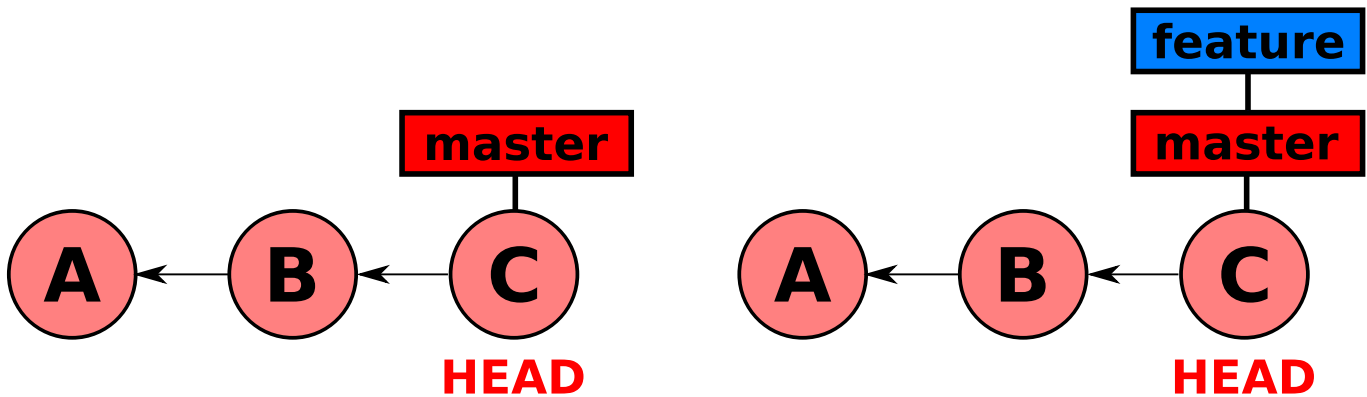
Argument	Purpose
<code>--verbose</code>	Also shows the last commit (HEAD)
<code>--all</code>	Also shows remote branches (covered later)

## 4.5. Creating Branches

- To create a branch, use `git branch <name> <commit>`
  - The branch is created on the version you are currently on

```
$ git branch feature          # Assume starting on branch master (name can be freely
chosen)
                             # <commit> defaults to HEAD, alternative 'git branch
```

```
feature cb54f3b'
$ git branch
  feature
* master          # Branch created but not currently on it (no star)
```



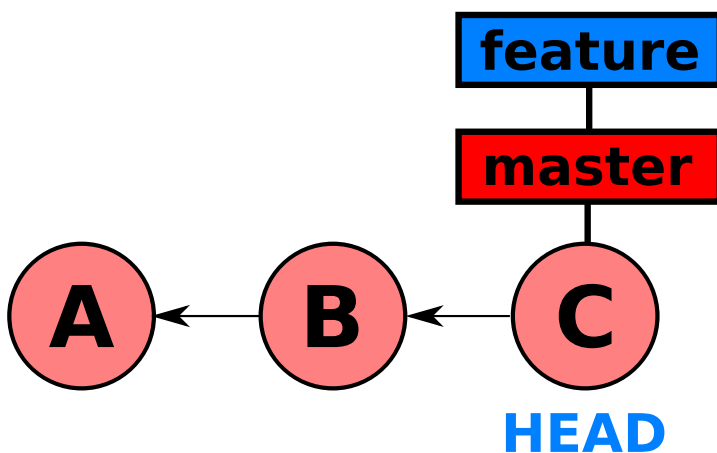
- Note that this does not automatically switch you to that branch
  - E.g. `HEAD` remains on master branch; following commits still done on master

## 4.6. Switching Branches

- Switching branches is done with `git checkout`
  - Only switches the `HEAD` to another branch
  - Any following commits will now happen on the feature branch

```
$ git checkout feature
Switched to branch 'feature'
$ git branch
* feature
  master    # Star has moved
```

- Use `git checkout -b feature` to create and switch directly

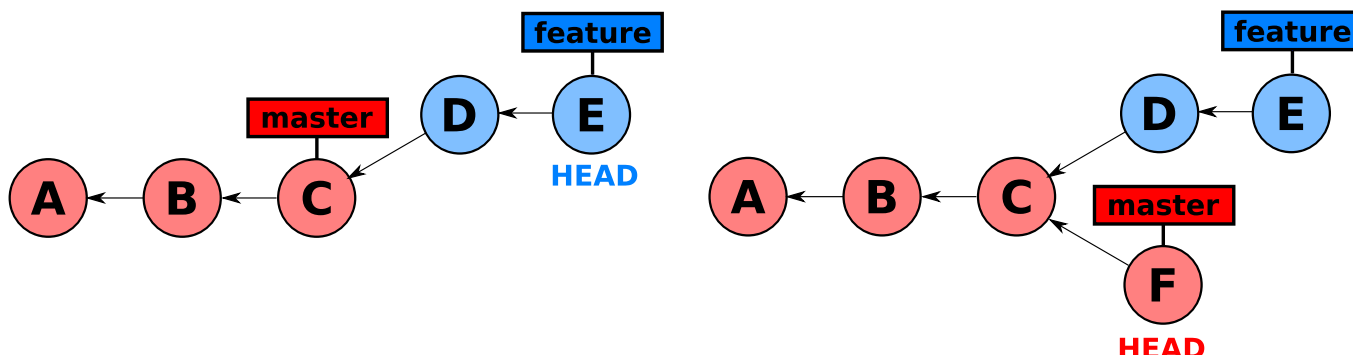


- Any commits done on a branch will evolve only that "reality"

```
$ git checkout feature # Move HEAD to C on feature
```



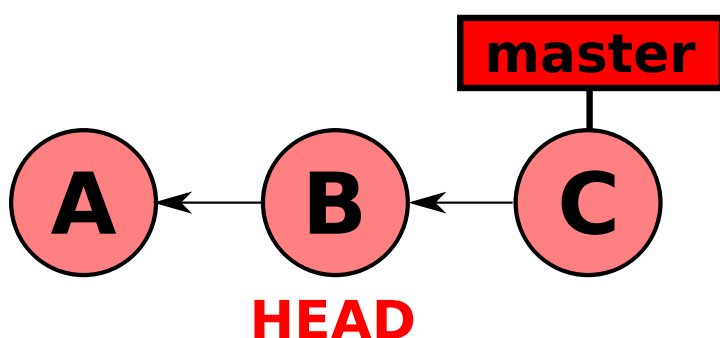
```
# Do some work
$ git commit -m "D"
# Do some work
$ git commit -m "E"
$ git checkout master # Move HEAD to C on master
# Do some work
$ git commit -m "F"
```



## 4.7. Detached HEAD

- Normally HEAD is pointing to the most recent version of a branch
- You can also checkout a specific (not the latest) version by commit id
  - This will end up with a detached HEAD

```
$ git checkout b78d6af # Not the same as 'git checkout b78d6af hamburger.md'!
You are in 'detached HEAD' state.
```



- Dangerous: any following commits will be dangling!
  - They are not part of any branch
  - Switching to something else at this point may cause you to loose work!

The truth is that when leaving a dangling commit, your work will not likely be lost immediately. However, git does clean up (garbage collect) any dangling commits, so it definitely will be lost in the near future. To prevent this, you should create a branch out of your dangling commit before leaving it (or soon after)!

The important thing to remember: be careful when dealing with a detached HEAD. If you have no clear intent with this be sure you switch to a branch before continuing work!

## 4.8. Removing and Renaming Branches

- Branches can be removed using `git branch --delete <branch>`

```
$ git branch --delete feature    # Does not proceed when work would be lost
$ git branch -d feature         # Same thing (lower case!)
```

- You are protected from losing work
  - When git detects commits that have not been integrated yet it refuses
    - You can override by adding the `--force` argument

```
$ git branch --delete --force feature    # Deletes without safety net. You may loose work!
$ git branch -D feature                  # Same thing
```

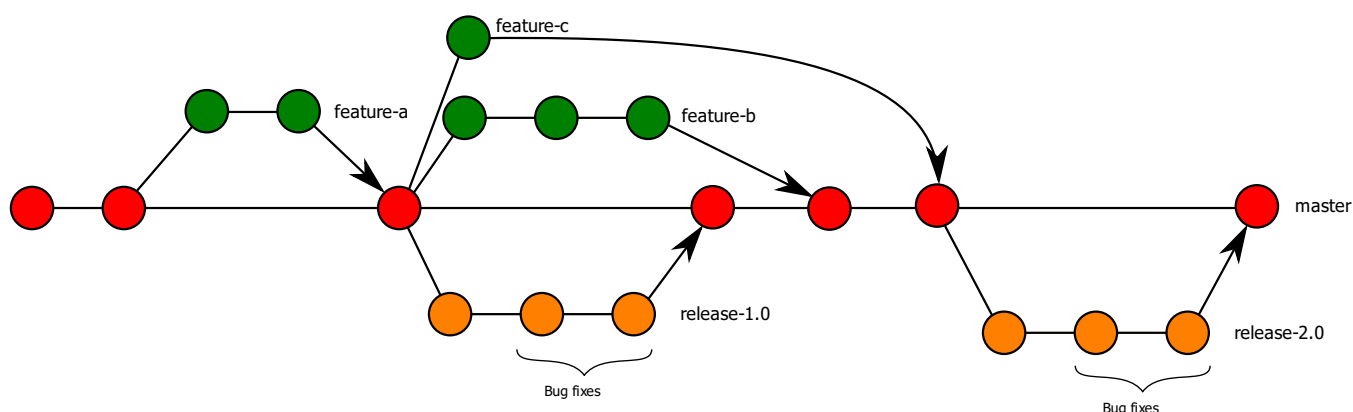
- Branches can be renamed using `git branch --move <old> <new>`

```
$ git branch --move feture feature    # Whoops, fixed a typo in my branch name
```

---

## 4.9. Practical Branching Strategies

- Part of Software Configuration Management (SCM)
- Trade-off between risk and productivity
  - No "silver bullet"; each team needs to decide what is most suitable for them
- Some approaches include
  - Branch per release
    - Each major version of the software is maintained in a single branch
    - Integration between branches is low
  - Branch per feature or topic
    - Developers make a task for each feature they work on
    - Integration between branches is high
  - Branch per platform
    - For each supported target platform or technology stack, a separate branch is maintained
    - Integration happens from a shared "common" branch
- Let's explore a hybrid form that is used a lot in practice
  - You can use this as the basis for your own decisions



- Depending on the circumstances, you can simplify or extend this
  - See <http://nvie.com/posts/a-successful-git-branching-model/> for an extended form

## 4.10. Specifying Revisions Revisited

- The special `COMMIT~n` syntax can be extended to navigate over branches

Syntax	Meaning
<code>COMMIT^2</code>	Selects the 2-nd parent (when joining a n-way merge) relative to <code>COMMIT</code>
<code>COMMIT^</code>	Shorthand for <code>COMMIT^1</code>
<code>COMMIT^0</code>	Alias for <code>COMMIT</code>

- Used the same way as the `~` variant

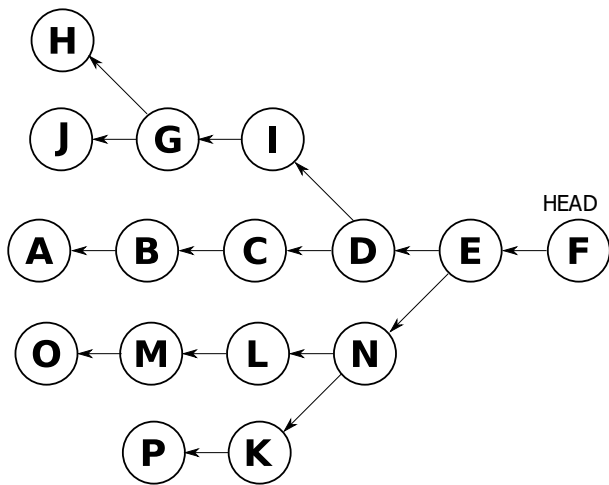
```
git checkout d3ab7cb^0      # Revision d3ab7cb itself
git checkout d3ab7cb^2      # The second parent from which d3ab7cb is the merge
result
git reset HEAD^^            # Would be the same as HEAD~~
git checkout HEAD~^2        # One commit before HEAD was a two-way merge
                             # we go one hop further still taking the second branch
```

- You can imagine it's an road network and you get instructions to navigate somewhere
  - E.g. `HEAD~^2~^`: "On the next crossroads, take the second exit, then two intersections further, take the first exit"
- Some visual examples (pretend letters are actually commit ids)
  - Branch ordering is left to right from arrow (In reality check `git log --all` for this)

Expression
<code>O = F^^2~~ (or F^1^1^2^^)</code>
<code>G = D^2^ (or D^2~)</code>
<code>P = E^^^ (or E~3)</code>

$M = N^{2^2}$  (or  $N^{2 \sim 1}$ )

$A = F^{2^{2^{2^2}}}$  (or  $F^{\sim 2^{2 \sim 2}}$ )



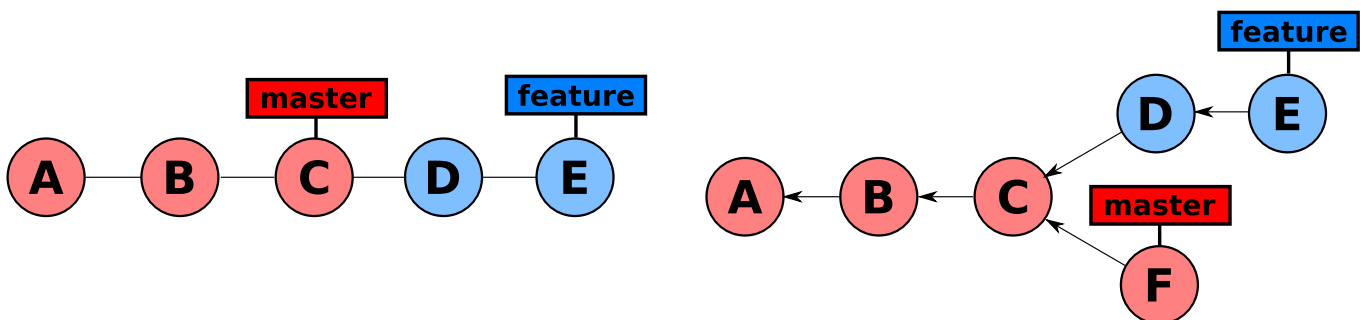
## 5. Merging

## 5.1. Merging Rationale

- We previously covered that branches can be joined back together
  - More formally we say that branches can be merged or integrated (into other branches)
- This is useful and necessary because team members can not indefinitely work "next" to each other on separate branches
  - Sometimes you need access to changes that someone else made
  - When the project is released, all changes from all team members must be combined
  - Bugfixes need to be applied to multiple branches

## 5.2. Branch Types for Merging

- There are two ways branches may turn out
  - Only one of the two evolves
    - One branch is simply in "the past"
    - Merge problems can never happen
  - They both evolve
    - Both branches have a different "now"
    - Merge problems may happen

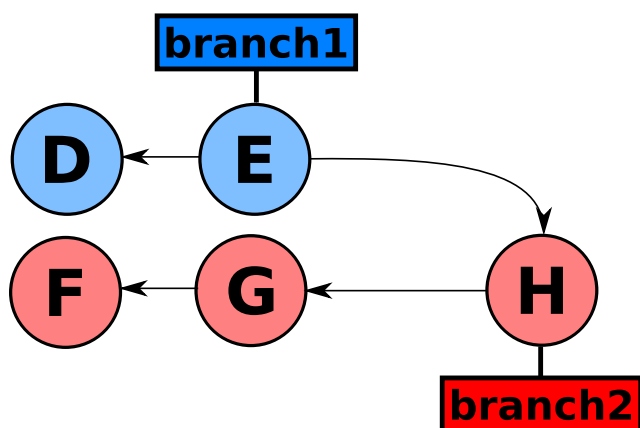


## 5.3. Performing a Three-Way-Merge

- Performing a three-way-merge is done using the `git merge` command

```
$ git checkout branch2 # Switch to the target (receiving) branch first
$ git merge branch1    # Merges all changes from branch-b (donor) into branch-a
```

- This will create a new commit on target branch that contains the combined result of both branches
  - E.g. All changes on donor branch appear in target branch as a single commit



- Does not remove the donor branch! Both branches continue to exist!

## 5.4. Merge Conflicts

- We previously covered that merging branches may cause problems
  - More formally we call this a merge conflict
- When there is no conflict, Git merges the results automatically
- When there is a conflict, manual intervention is needed to resolve it

Non conflicting	Conflicting
Two team members each change a different file	Two team members change the same line in the same file
Two team members modify the same file on different lines	One member removes a file that another member modifies

- Being able to resolve merge conflicts is an important skill when working with Git

## 5.5. Resolving Merge Conflicts

- When attempting to merge conflicting changes, Git fails with a `CONFLICT`

```
$ git checkout master          # Want to merge into master (target)
$ git merge tiramisu-refinements # Donor branch called 'tiramisu-refinements'
Auto-merging tiramisu.md
CONFLICT (content): Merge conflict in tiramisu.md
Automatic merge failed; fix conflicts and then commit the result.
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)
```

```
both modified:   tiramisu.md
```

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

- The working tree is now in merging stage
  - No commit has been done, need to resolve conflict manually first
  - Conflicting files will now contain both "realities"
- Conflicting "realities" are inside the related files

```
<<<<<<< HEAD
6 coffee spoons of Amaretto
=====
4 coffee spoons of Amaretto or another liquor of choice
>>>>>>> tiramisu-refinements
```

- This now needs to be manually resolved (often context sensitive)
  - Sometimes you can entirely accept one of the two
  - Sometimes you need to combine the two in an "smart" way (that the computer wouldn't know)

```
6 coffee spoons of Amaretto or another liquor of choice
```

- When finished, add and commit the changes

```
$ git add tiramisu.md      # Adding marks the conflict as "resolved"
                           # Repeat for all other conflicting files
$ git commit --message "Merged tiramisu-refinements into master. Woohoo!"
```

## 5.6. Useful Merge Arguments

- Use `--no-commit` to prevent an automatic commit even when there were no conflicting changes
  - Useful if you want to verify if the merge causes any regressions
    - e.g. "mismatch" is likely (Git does not know the used programming language)

```
$ git merge myTarget --no-commit    # Never auto-commit Always required manual commit!
```

- Use `--abort` to cancel an ongoing merge attempt
  - Useful if you want to ask a colleague for help

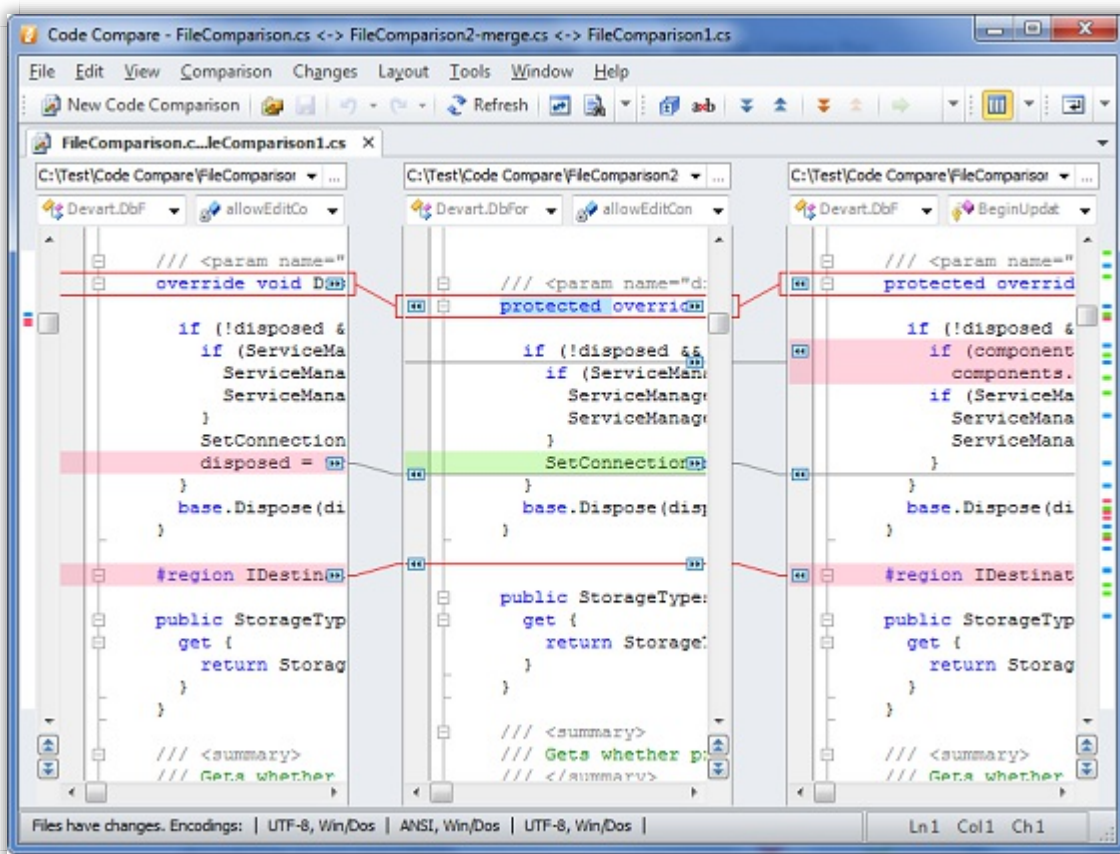
```
$ git merge myTarget          # Don't know what you're getting into!
CONFLICT (content): Merge conflict ...
                               # Oh boy, I need to get out of this mess
_ _;
$ git merge --abort           # Eject, Eject!
```



A **mismatch** is a case where Git's merge strategy does not detect two conflicting changes, and thus the resulting auto-merge is incorrect. This happens usually when there is no direct conflict (i.e. not editing the same line in the same file), but the result is invalid for the source format the file is written in. The most likely case of such a mismatch is a duplicate symbol definition such as the same procedure being declared twice (in different locations within the source file).

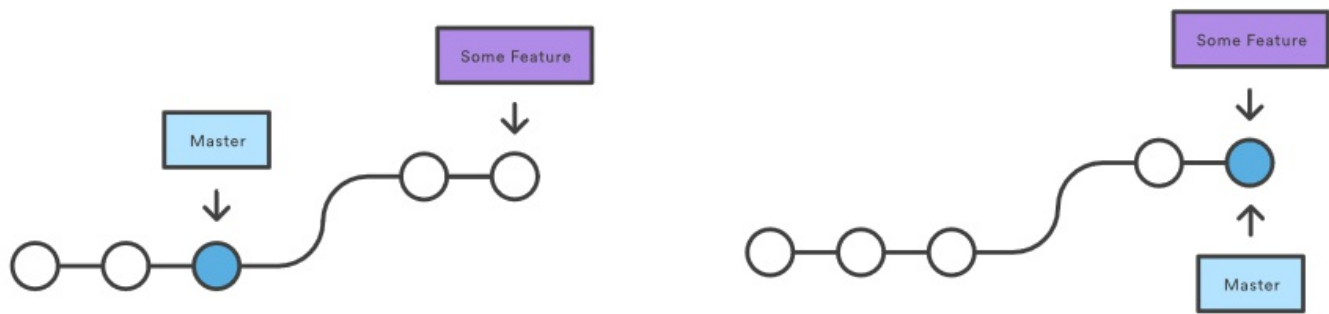
## 5.7. Using a Mergetool

- Using the CLI for resolving merge conflicts can be tedious
  - Many GUI tools exist for this purpose
    - Can be used with `git mergetool` if so configured (defaults to `vimdiff`)



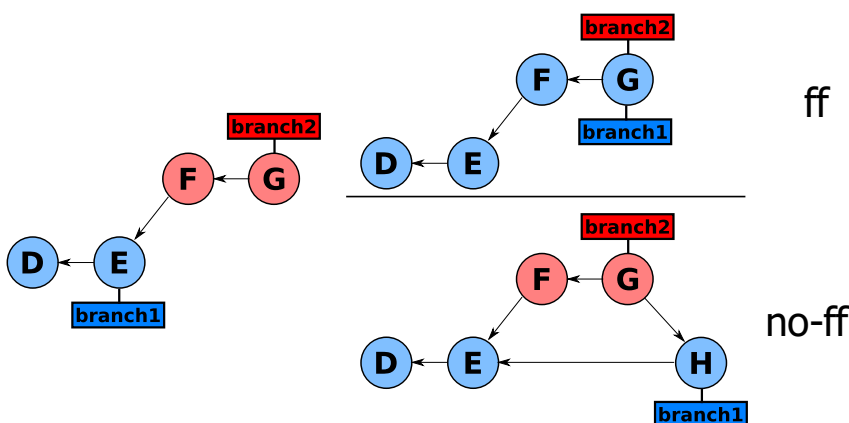
## 5.8. Fast Forwarding

- When merging two branches that have not both evolved, a merge can never have a conflict
  - One branch will always be a continuation of the other's "reality"



- Merging these two branches will result in a fast forward, where `master` can simply be moved further downstream to `some feature`
- You can control the behaviour of fast-forward merges using arguments

Argument	Meaning
<code>--ff</code> (default)	Performs a fast-forward if possible
<code>--no-ff</code>	Always merge with an extra commit
<code>--ff-only</code>	Refuse to merge if fast-forward is not possible

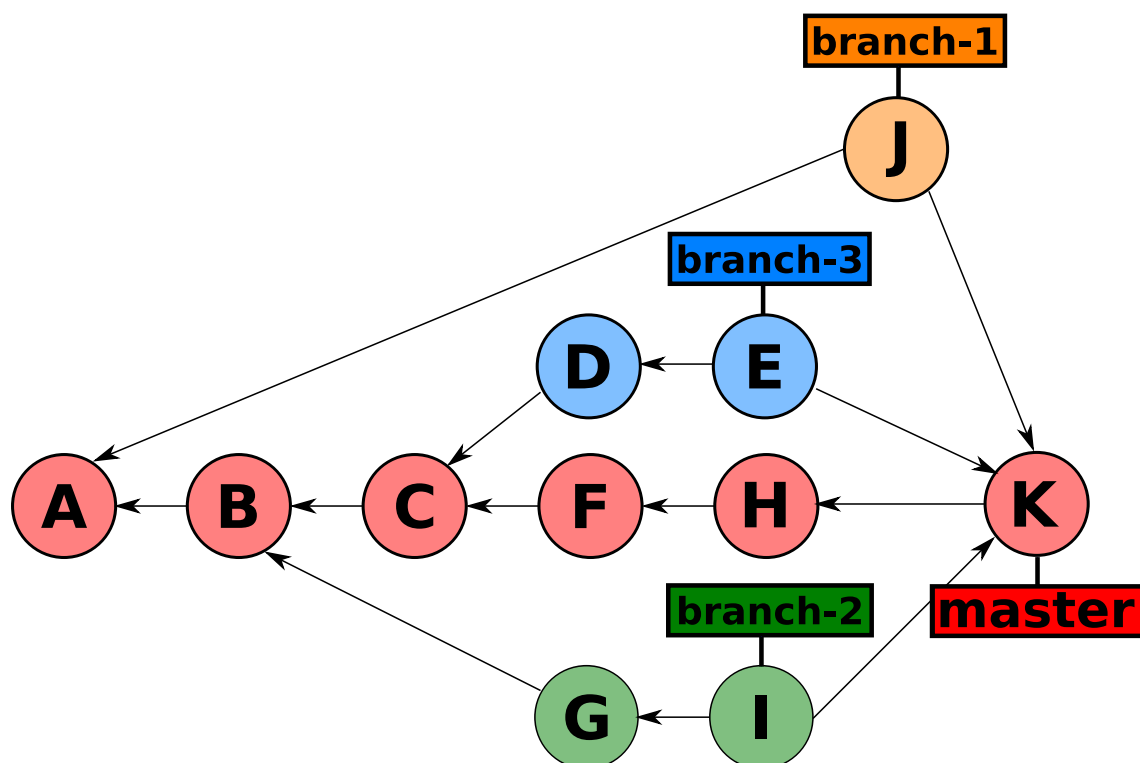


- Useful when
  - You want to preserve a topic branch in the history (`no-ff`)
  - You want to enforce that previous actions ended up with a FF (see `rebase`) (`ff-only`)

## 5.9. Octopus Merges

- Git allows merging multiple branches at the same time, called an octopus merge

```
$ git checkout master          # Master is target branch
$ git merge feat-a feat-b feat-c # Merge feat-a, feat-b and feat-c into master at
the same time
```



## 5.10. Other Merge Strategies

- Using `git merge --strategy=<strategy>` you can choose several other merge strategies

Strategy	Usage
resolve	Fast 3-way-merge
recursive	Default when merging a single branch (can optionally add <code>--strategy-option=ours</code> or <code>theirs</code> to choose which lines get precedence)
octopus	Default when merging multiple branches at the same time
ours	Ignore any conflicting files in favour of the target branch (not the same as <code>recursive -X ours</code> )

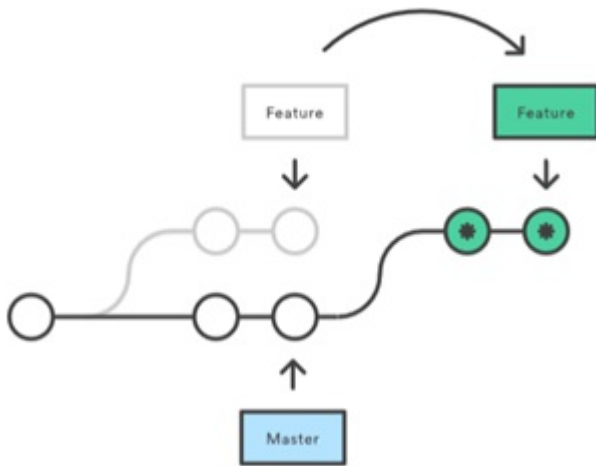
```
$ git merge --strategy=recursive --strategy-option=ours feature-x    # Automerge
accepting our changes
$ git merge --strategy=resolve feature-x
```

- Normally it is not necessary to manually change the utilized merge strategy
  - A choice is made automatically based the number of branches that get merged

Note: The difference between `recursive -X ours` and `ours` is that the first acts on line level, where the second acts on file level.

## 5.11. Rebasing

- `git rebase` is an alternative to merging
  - merge integrates an entire branch as a single commit (or fast-forward)
  - rebase moves an entire branch so it is rooted at another commit
    - This "transplants" a reality to occur in series rather than in parallel with a target branch



- This is done by individually replaying all commits of a selected range upon the target branch

An interesting side effect of rebase is that it flattens out the graph and avoids "merge loops".

- Rebasing is done as follows:
  - Each individual commit is replayed upon the receiving branch (here master)

```
$ git checkout feature    # Checkout the branch that needs to be moved (not the
                           receiving branch!!!)
$ git rebase master       # Rebase feature-x on HEAD of master (does not move
                           master!!!)
Applying: Added tiramisu refinements
Applying: Changed hamburger recipe
Applying: More bolognese added
Applying: Added meatballs to tomato soup
```

- Reapplying commits can also cause conflicts
  - In this case, git pauses the rebase and offers you three choices

Choice	How to choose	Meaning
continue	<code>git rebase --continue</code>	Resume the rebase (after you have manually solved conflict)
skip	<code>git rebase --skip</code>	Drops this commit (it is not relevant in this "reality")
abort	<code>git rebase --abort</code>	Bail out, and cancel the entire ongoing rebase operation

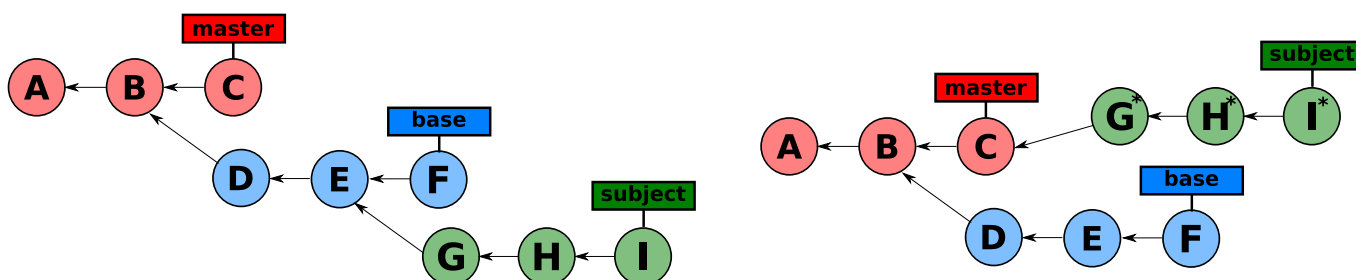
- Example of a conflicting rebase:

```
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: Added tiramisu refinements
Using index info to reconstruct a base tree...
M      tiramisu.md
Falling back to patching base and 3-way merge...
Auto-merging tiramisu.md
CONFLICT (content): Merge conflict in tiramisu.md
error: Failed to merge in the changes.
Patch failed at 0001 Added tiramisu refinements
The copy of the patch that failed is found in: .git/rebase-apply/patch

When you have resolved this problem, run "git rebase --continue".
If you prefer to skip this patch, run "git rebase --skip" instead.
To check out the original branch and stop rebasing, run "git rebase --abort".
```

- You can also rebase the difference between two branches onto a third branch
  - E.g. rebase `subject` from common ancestor with `base` onto `master`

```
$ git rebase --onto master base subject
```



- The result of rebasing is always a fast-forward capable branch topology
  - Can then easily be merged the normal way

```
$ git checkout master
$ git merge subject    # Will never ever have any merge conflicts!
```

## 5.12. Rebasing vs Merging

- Rebasing can have an advantage over merging
  - Difficult merges can be processed one-by one
  - You can shape the way commits are shown in history
  - You can control what part of a branch you want to merge
- It can also have disadvantages
  - You're actually rewriting history
- Is your history a record or rather a story of what happened?

- A record is what actually happened (ugly truth)
- A story is how the truth is most conveniently explained
- Generally speaking it's a good idea to only rebase local branches
  - Changing the history of what your colleagues did may not be appreciated...
- Remember though `git rebase` is a tool, that knows many uses

## 5.13. Squashing

- It is generally recommended that you commit often
  - However this creates a very bloated history

```
$ git log
4ad9399 Fixed a regression issue related to feature X
3bcc74d Oops accidentally added a file that shouldn't be there
ce93df2 Completed feature X
074aa58 Implemented part of feature X service
ac36590 Added some groundwork for feature X
```

- To prevent this, you can squash together a number of commits into one

```
$ git checkout master
$ git merge --squash feature-x # Adds a single commit onto master containing all
changes from feature-x
                                # Does not perform a 3-way merge! There will be no
history of the merge!
```

## 5.14. Interactive Rebase

- With interactive rebase you can also squash commits
  - Interactive rebase gives fine grained control on what to do with each commit

```
$ git checkout feature-x
$ git rebase --interactive master
```

- Presents VIM editor to select actions for each commit
  - Replace `pick` with your action to perform

```
pick 3c74059 Added tiramisu refinements # similar line for each commit included in
rebase
```

Action	Meaning
pick	Regular rebase (add as commit*)

reword	Pick but change the commit message
squash	Squash into previous commit
fixup	Squash without registering commit message (normally added)
drop	Skip commit (lost!)
...	(some others omitted)

---

## 5.15. Cherry Picking

- Using `git cherry-pick` you can pick a single commit and reapply it elsewhere
  - Useful to take a single commit from another branch, without fully merging

```
$ git checkout master          # Target branch
$ git cherry-pick cdf78bf      # Take commit cdf78b and execute it on HEAD of current
master
```

---

## 5.16. Merging Challenges

- There are also some challenges when team members are performing merges
  - "Mergephobia"
    - When developers are actively avoiding to merge because they expect to have merge problems
  - "Mergephilia"
    - When developers are constantly performing mini-merges back and forth from the main branches
  - "Monstermerge"
    - When saving up too much work before integrating, resulting in huge merge conflicts. This happens often on friday afternoons
  - "Freezemerger"
    - When the project-integrator prevents all other team members from contributing changes when in the process of merging
- These phenomena occur in most projects to a certain degree. It only becomes problematic in excess...

## 6. Remotes



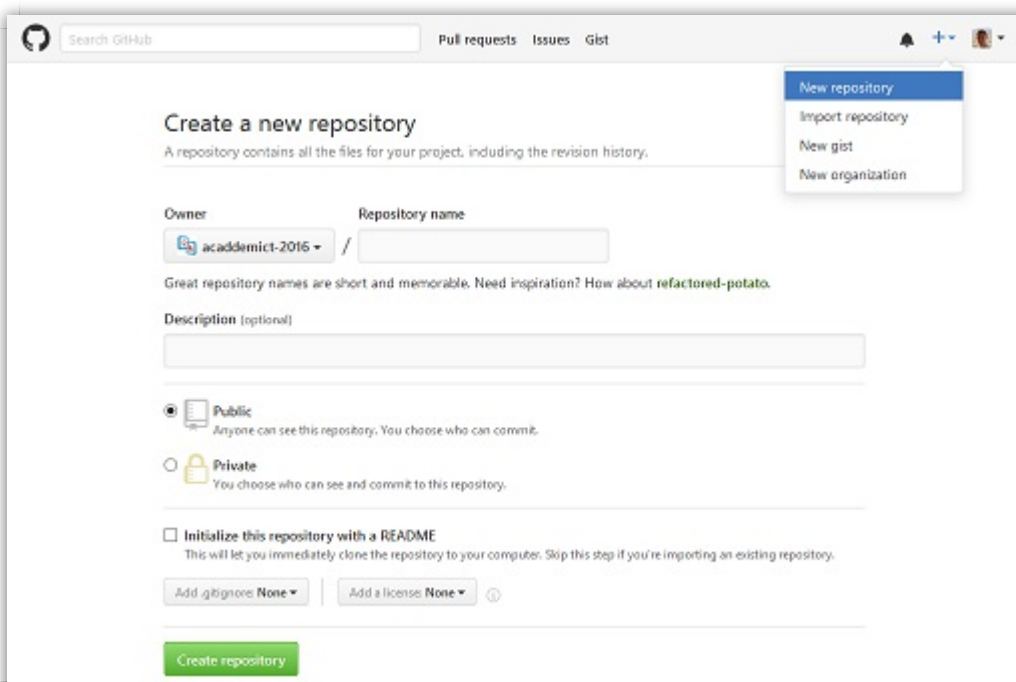
## 6.1. What is a Remote?

- Git is a distributed VCS
  - This means that each team member has a full copy of the project's entire history
  - Developers work locally on their own git repository
    - Committing, branching, merging, all act on the local repository (`.git/`)
- A remote is a link to another copy of the repository that lives elsewhere
  - On a separate server (such as github)
  - Directly on another team member's machine (using a network share)
  - Somewhere else on your own machine (different directory)
- Occasionally, remotes need to be synchronized with to share work with other team members
  - Since local copies can diverge from remotes, so merging is an issue

---

## 6.2. Creating a Remote Repository

- A central server is often used to share work with other team members
  - Can be hosted in the cloud, or on-premise
- GitHub is a popular example of a cloud based solution
  - A new empty repository can be created using the web interface



---

## 6.3. Transfer Protocols

- A remote URL can use any of the supported protocols
  - The remote repository must be made available through this protocol
- Some supported protocols

Protocol	Example URL	Description
file	file:///c:/myprojects/recipes	Remote hosted on same filesystem
ssh	git@github.com:jimi/recipes.git	Remote hosted through SSH tunnel (most secure option)
http	<a href="http://realdolmen.com/projects/recipes.git">http://realdolmen.com/projects/recipes.git</a>	Remote hosted insecurely through simple HTTP
https	<a href="https://realdolmen.com/projects/recipes.git">https://realdolmen.com/projects/recipes.git</a>	Remote hosted through secure HTTP (more secure)

- Configuring a git host that supports these protocols is an sys-ops task, and is out of scope of this course
    - When using a cloud-based solution, this is generally taken care of for you
- 

## 6.4. Cloning

- Download copy of (existing) repository to your machine using `git clone`
  - You'll need the remote URL of the repository

```
# Note: depending on the environment, you may need to provide username/password
$ git clone https://github.com/realdolmen/recipes.git # Create directory 'recipes'
and                                                  # download entire project's
                                                    history into it
Cloning into 'recipes'...
remote: Counting objects: 9, done.
remote: Total 9 (delta 0), reused 0 (delta 0), pack-reused 9
Receiving objects: 100% (9/9), done.
Resolving deltas: 100% (1/1), done.
Checking connectivity... done.
```

```
$ git clone https://github.com/realdolmen/recipes.git myDirectory # Choose directory
name yourself
```

- `git clone` conveniently configures a number of things automatically
    - Downloads entire project's history in `.git/`
    - Checks out `master` branch
    - Configures a remote to your clone URL called `origin`
-

## 6.5. Configuring Remotes

- Remotes can be configured using `git remote`

```
# List all the remotes (verbose also shows URL)
$ git remote --verbose
origin https://github.com/realdolmen/recipes.git (fetch)
origin https://github.com/realdolmen/recipes.git (push)
```

```
# Add a remote with name 'jimi' pointing to location
'https://github.com/jimy/recipes.git'
$ git remote add jimy https://github.com/jimy/recipes.git
```

```
# Renaming remotes
$ git remote rename jimy jimi # Rename remote from
'jimy' to 'jimi'
$ git remote set-url jimi https://github.com/jimi/recipes.git # Change the URL of
the remote
```

```
# Removing remotes
$ git remote remove jimi
```

## 6.6. Remote and Tracking Branches

- When a remote is present, most branches will have a remote equivalent

```
# Display both local and remote branches
$ git branch --all
* master # Local tracking branch
remotes/origin/master # Remote version of master
remotes/origin/feature-x # Remote branch with no local tracking equivalent (yet)
```

- If local branch with same name exists, this is called a tracking branch
  - This happens as soon as you check out the remote branch for the first time

```
$ git checkout feature-x
Branch feature-x set up to track remote branch feature-x from origin.
Switched to a new branch 'feature-x'
```

- Remember: you always work on a local version of the branch!

## 6.7. Fetching

- To download all changes from a remote, use `git fetch`

```
$ git fetch origin # Download latest changes from remote with name 'origin'
remote: Counting objects: 6, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 6 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (6/6), done.
From github.com:jimi/recipes
* [new branch]      feature-x      -> origin/feature-x
```

- Notes:

- Changes do not get 'pushed' to your local copy when they happen, you'll have to periodically download updates using `fetch`
- Fetch only updates remote branches, it never changes any of your local branches

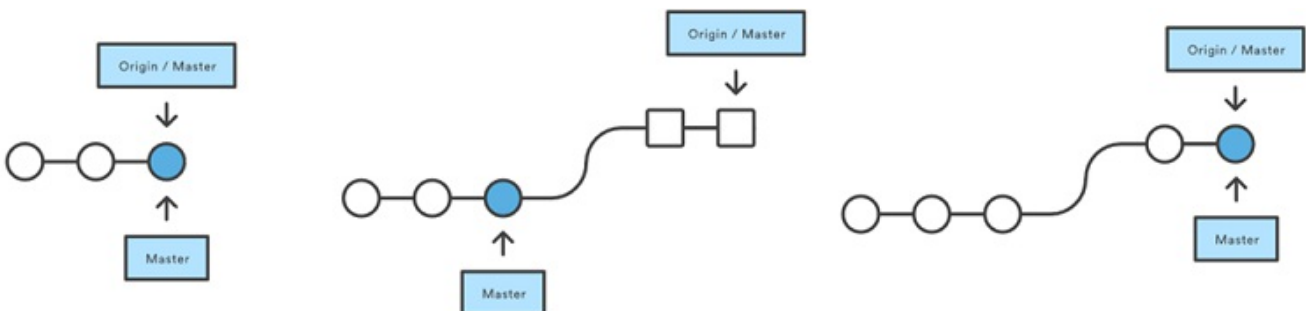
- You can now start interacting with the remote branches

```
# Merge local and remote versions of branches (but wait... there's a better way to do this!)
$ git checkout master
$ git merge remotes/origin/master
```

## 6.8. Pulling

- `git pull <remote> <branch>` downloads updates and immediately integrates (merge or rebase) the updates into your tracking branch
  - Actually nothing more than a shorthand for `git fetch` and `git merge`

```
$ git pull origin master # Fetch from 'origin' and merge into tracking branch 'master'
$ git pull origin feature-x --rebase # Fetch from 'origin' and rebase onto tracking branch 'feature-x'
```



- Notes:

- This performs a merge or rebase, so be prepared to deal with merge conflicts
- Working tree must be clean for this, so consider using `git stash` first
- Unlike fetch, this actually changes your local branch

## 6.9. Pushing

- Upload changes to a remote using `git push <remote> <branch>`

```
$ git push origin master           # Upload all updates of local 'master' branch
to remote 'origin'
$ git push origin --delete deprecated # Delete the branch 'deprecated' on remote
'origin'
```

- Interesting arguments

Argument	Purpose
<code>--all</code>	Push all branches instead of just one
<code>--delete</code>	Delete a branch remotely (local branch remains)
<code>--set-upstream</code>	Link local and remote branches, so you can call <code>git pull</code> and <code>git push</code> without arguments

- If the remote has since changed, the push is rejected
  - You have to pull the changes first (and resolve any conflicts) and push again
    - This is normal, because you can not solve a merge conflict on the remote side
- Depending on the environment you may not have rights to push into (or delete) a remote branch

Note: Moving (renaming) a branch can not be done directly. Instead, you have to rename the branch locally, push it and remove (`--delete`) the old branch remotely.

- It is not clearly defined when to push or pull your changes, but some general rules apply
  - Pull or fetch frequently, so your local copy remains reasonably up-to-date
    - You don't want to be surprised all your work is no longer relevant
  - Push is often done when your task is complete or does not conflict
    - Say once or twice per day (not a strict rule)
    - Make sure you don't push any changes that would actually break things! (blocks your colleagues)
- More advanced users often clean up their local commits (using rebase or squashing) before pushing changes
  - Don't change the history of remote changes though! This would cause massive problems for anyone working on the "old" history...

## 6.10. Forking

- Forking is a feature offered by many cloud services
  - Not really a Git feature
- It is making a `clone` on the server side, to allow you to work on your own version of the project
  - Useful because you won't have rights to directly change the original repository
  - Extremely popular in the Free and Open Source (FOSS) world



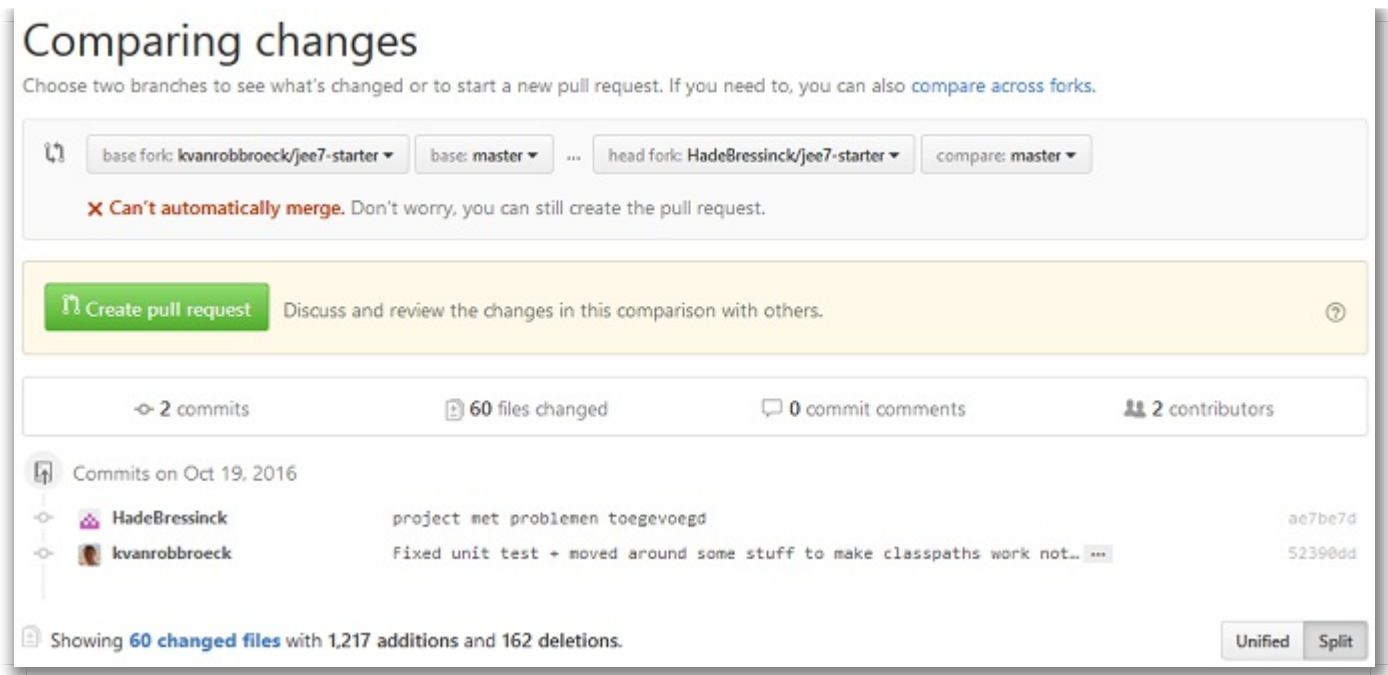
- After forking you can clone your own remote copy and work on that
  - Often a remote is added to the source to be able to pull ongoing changes

```
$ git clone https://github.com/jimi/linux-kernel.git           # Forked via web  
interface  
$ git remote add linus https://github.com/torvalds/linux.git # Add a link to source  
to pull in updates
```

---

## 6.11. Pull Requests

- From forked repositories, you can send pull requests back to the source
  - Suggests contributions you've made to be included in the source
    - You can't push directly, but you can request them to pull
  - Fundamental mechanism used in the FOSS world to contribute

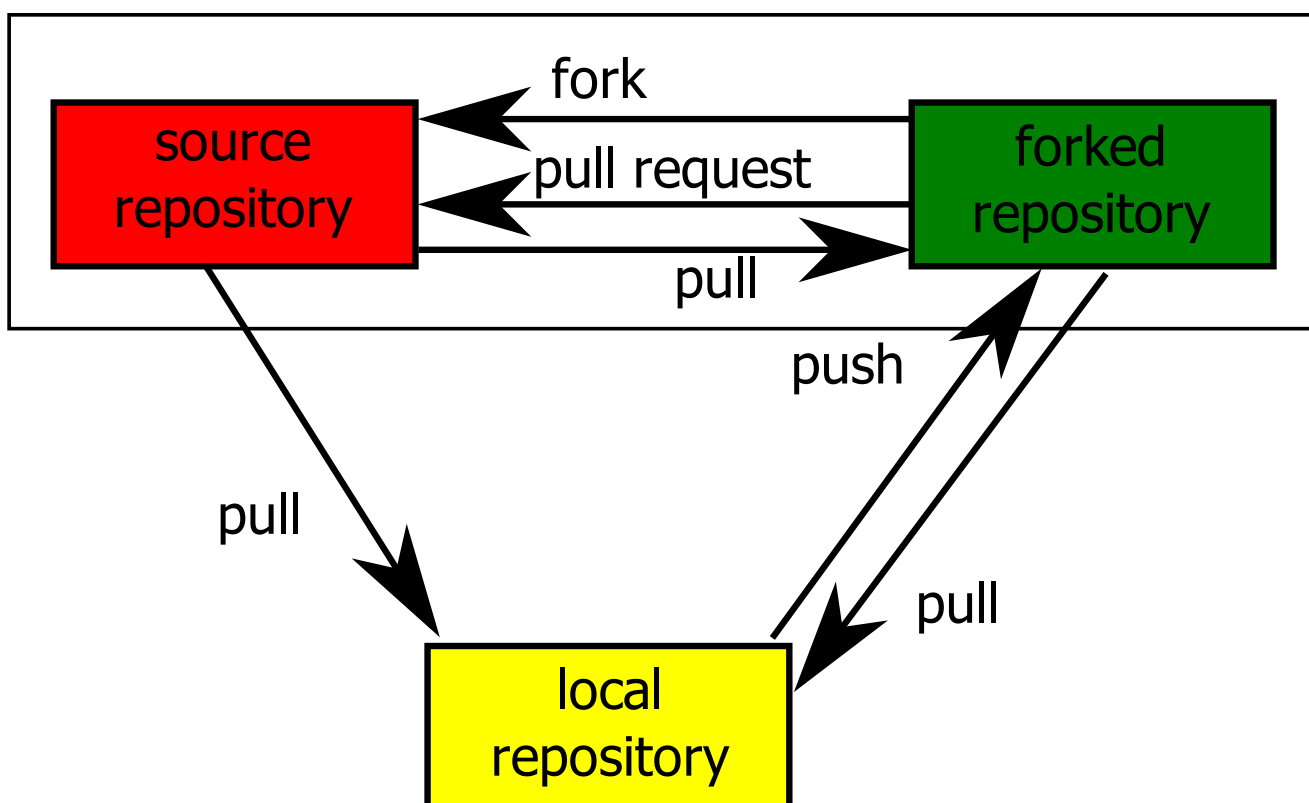


- Note Git also has a pull-request mechanism using `git request-pull`

## 6.12. Forking-Pull Requests Overview

- Overview of the forking / pull requests mechanism

server



## 7. Security



## 7.1. Remote Access Using SSH

- Communication with remotes over HTTPS is not the safest option
  - You have to authenticate using username / password
  - When this information is leaked, your entire account is compromised
- Much safer is to configure Git to use an SSH tunnel
  - SSH (Secure Shell) is a very secure connection to a remote machine using standard protocols
- Setting up SSH is somewhat tedious though:
  - Create a private/public key pair
  - Upload your public key to the server
  - Provide private key password upon communicating with the server
- When configured though (only once), it provides the best security
  - Even if your private key is compromised, your entire account is still safe!
- An SSH key can be generated using the `ssh-keygen` command
  - Make sure you provide a strong password here

```
$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (~/.ssh/id_rsa):           # File to store key
in
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in ~/.ssh/id_rsa.           # Private key
(password protected)
Your public key has been saved in ~/.ssh/id_rsa.pub.           # Public key (safe for
everyone to see)
```

- Copy the contents of your public key file
  - The public key is like an open padlock, of which you own the only (private) key

```
$ cat ~/.ssh/id_rsa.pub # Public key file!!!!
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQCEl3n5yspqf7MIHo0fklK8ZyEfxQg== jimi@mail.com #
Very long string
```

- In the server's web interface, search for an option to leave your public SSH key
  - This example uses GitHub

SSH keys New SSH key

This is a list of SSH keys associated with your account. Remove any keys that you do not recognize.

Title

My Development machine

Key

ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQCAQCel3n5yspqf7MIHo0fkIK8ZyEfxQg== jimi@mail.com

Add SSH key

- Now, you can start using the SSH protocol

```
$ git clone git@github.com:jimi/recipes.git # ssh protocol, not https!!
```

- Whenever your SSH key gets compromised, remove it from the list!
  - Your account is still safe!

## 7.2. Signing Commits With GPG Keys

- Currently we do not have non-repudiation
  - Anyone could push commits using our name and email address
- In an trustless environment, you can use GNU Privacy Guard (GPG) to sign your commits
  - It's included with your Git installation
- Setting it up requires a few steps:
  - Generating a new GPG key
  - Configuring Git to use private key as the signing key
  - Sign commits using the `--gpg-sign` option
  - Configure server to verify commits using your public GPG key
- Generate a new GPG key using `gpg --gen-key`

```
$ gpg --gen-key
# Choose the following options:
# Key type: RSA and RSA
# Key size: higher is better, but 2048 is okay
# Expire: shorter is more secure but less convenient. Don't use non-expiring key!
# Name: your name
# Comment: anything
# Password: choose a strong password!
pub 2048R/22210328 2016-11-15
    Key fingerprint = D459 871A 90B1 EF1F 4496 CE40 63A1 4A5E 2221 0328
uid                               Jimi Hendrix (Guitar Player) <jimi@mail.com>
```

```
sub 2048R/C0FA5236 2016-11-15
```

- Configure Git to use this key as your signing key

```
$ git config --global user.signingkey 22210328 # See output of 'gpg --gen-key' or 'gpg --list-keys'
```

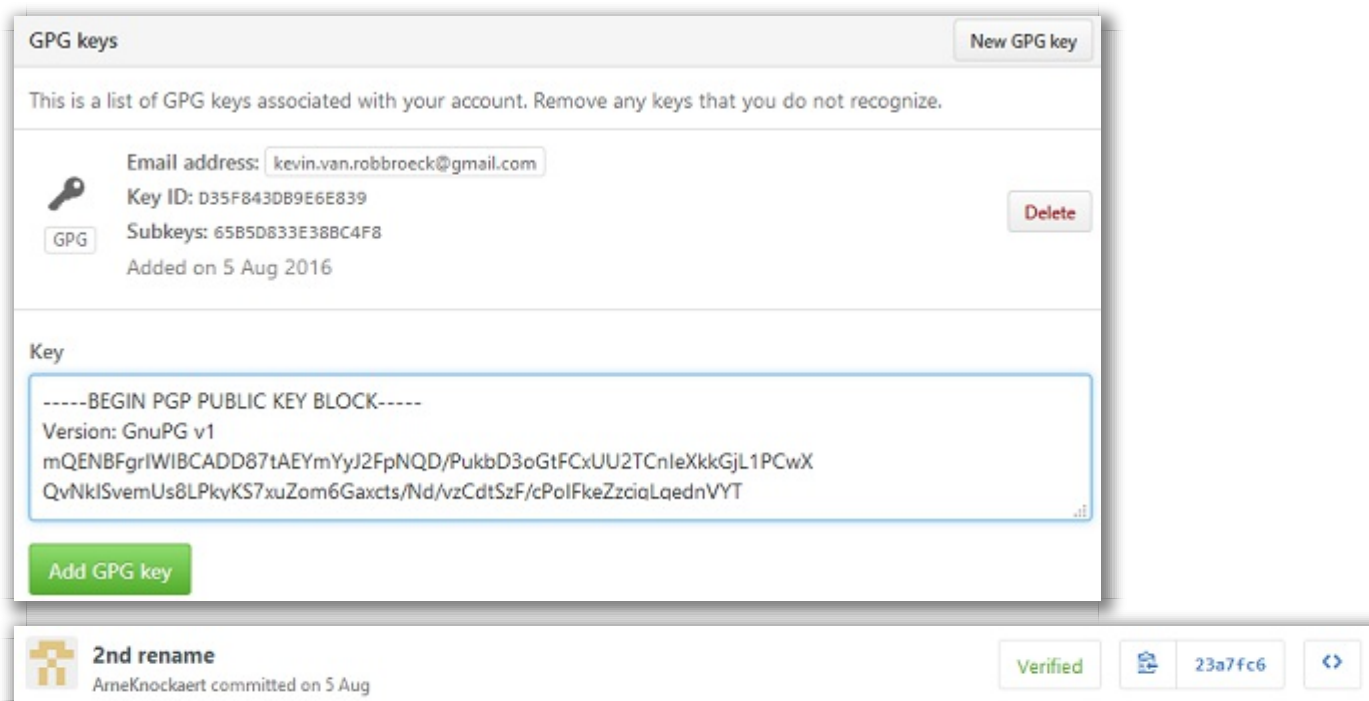
- You can now start signing your commits using `--gpg-sign`

```
$ git commit -m "Added some important work" --gpg-sign
```

```
You need a passphrase to unlock the secret key for
user: "Jimi Hendrix (Guitar Player) <jimi@mail.com>"
2048-bit RSA key, ID 22210328, created 2016-11-15
Enter passphrase: *****
```

- Signing can also be done on merge and push
- To avoid having to always specify `--gpg-sign` you can create an alias
- You may also want to upload your public GPG key to the server
  - Allows the server to verify commits made by you

```
$ gpg --armor --export jimi@mail.com
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: GnuPG v1
mQENBFgrIWIBCADD87tAEYmYyJ2FpNQD/PukbD3oGtFCxUU2TCnIeXkkGjL1PCwX # Very long string
```



- Verify anyone's authenticity manually by using `git log` or `git verify-commit`

```
$ git log --show-signature
commit 321454fd0c643fe8c83ed4f1f83ce28baafe7fa
gpg: Signature made di 15 nov 2016 16:08:48 RST using RSA key ID 22210328
```

```
gpg: Good signature from "Jimi Hendrix (Guitar Player) <jimi@mail.com>"
Author: Jimi Hendrix <jimi@mail.com>
    Added some ketchup to hamburger.md
```

```
$ git verify-commit 321454f
gpg: Signature made di 15 nov 2016 16:08:48 RST using RSA key ID 22210328
gpg: Good signature from "Jimi Hendrix (Guitar Player) <jimi@mail.com>"
```

- This requires the contributor's public key to be present in the keychain
  - Otherwise it will tell you the commit is signed but can not be verified

```
$ gpg --import janis.key      # Colleague's public key exported with gpg --export
```

---

## 8. Extras

## 8.1. Creating Tags

- A tag is a label that can be placed on any existing commit
  - Easier than remembering the commit ID
- Often used to mark a specific revision as the "production release"

```
$ git tag 1.0.0 a9265b2      # Create a tag '1.0.0' for commit ID a9265b2 (defaults to HEAD)
$ git tag                  # List all tags
$ git tag --delete 1.0.0    # Remove tag '1.0.0'
```

- Can also be made visible with `git log --decorate`

```
$ git log --oneline --decorate
...
a9265b2 (tag: 1.0.0) Final bugfix for production release made
...
```

---

## 8.2. Creating Command Aliases

- Using the CLI, you often have to execute the same commands, with a lot of arguments
  - To avoid tediousness, you can create command aliases

```
# Create an alias 'git st' for 'git status'
$ git config --global alias.st 'status'

# Create a handy alias 'git ls' to include a bunch of typical arguments
$ git config --global alias.ls 'log --oneline --stat --graph --decorate --all --show-signature'
```

- They can then be used as follows

```
$ git st
$ git ls
```

---

## 8.3. Filtering Branches

- Using `git filter-branch` you can rewrite commit metadata by means of a filter script
  - Allows you to change the name of a committer, commit message, etc...

```
$ git filter-branch -f --env-filter '
if [ "$GIT_AUTHOR_NAME" = "JimiH" ]          # Filter all occurrences of JimiH
then
    export GIT_AUTHOR_NAME="Jimi Hendrix"    # Rewrite to Jimi Hendrix
```

```
export GIT_AUTHOR_EMAIL="jimi@mail.com"          # Also rewrite email
echo "Rewrite history for $GIT_COMMITTER_NAME"
fi'
```

Environment variable	Meaning
GIT_AUTHOR_NAME	Name of the author of a commit
GIT_AUTHOR_EMAIL	Email of the author of a commit
GIT_AUTHOR_DATE	Date of when the commit happened
GIT_COMMITTER_NAME	Name of the committer (e.g. a merge)
GIT_COMMITTER_EMAIL	Email of the committer
GIT_COMMITTER_DATE	Date of then the activity happened (e.g. a merge)

## 8.4. Creating Patches

- Sometimes it is useful to create a patch that contains a number of changes in a single file
  - You can send them per email for example
- Create a patch as follows

```
# Create a patch that contains all differences from 'master' to 'mybranch'
$ git checkout mybranch
$ git format-patch master --stdout > mychanges.patch
```

- Apply a patch as follows

```
$ git apply mychanges.patch
```

- Useful arguments

Argument	Meaning
--stat	Lists what changes would be made
--check	Check if there would be any merge conflicts

## 8.5. Additional Commands

- There are still a number of commands that we haven't covered

Command	Purpose
git diff	Displays changes between commits (similar to <code>git log -p</code> )

git whatchanged	Similar command to <code>git log --stat</code>
git gc	Removes any dangling commits (like the result of committing on a detached head)

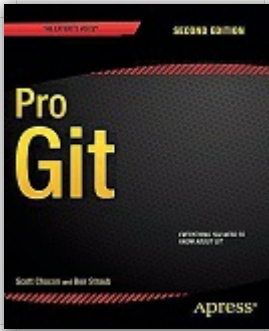
---



## 9. References

## 9.1. Useful References

- Pro Git
  - An excellent book about git, freely available
    - <https://progit2.s3.amazonaws.com/en/2016-03-22-f3531/progit-en.1084.pdf>



Reference	Url
Git manual	<a href="https://git-scm.com/doc">https://git-scm.com/doc</a>
Microsoft article about branching and merging	<a href="https://msdn.microsoft.com/en-us/library/aa730834(v=vs.80).aspx">https://msdn.microsoft.com/en-us/library/aa730834(v=vs.80).aspx</a>
Git branching strategies	<a href="http://www.creativebloq.com/web-design/choose-right-git-branching-strategy-121518344">http://www.creativebloq.com/web-design/choose-right-git-branching-strategy-121518344</a>
A successful Git branching model	<a href="http://nvie.com/posts/a-successful-git-branching-model/">http://nvie.com/posts/a-successful-git-branching-model/</a>



# Git Version Control Exercises

Text: EN

Software: EN



# Table of contents

- 1. Basic Commands Exercises
  - 1.1. Configuration
  - 1.2. Contents
  - 1.3. Creating and populating a repository
  - 1.4. Modifying the repository
  - 1.5. Recovering some changes
- 2. Branches Exercises
  - 2.1. Detached HEAD
  - 2.2. Creating branches
  - 2.3. Deleting branches
  - 2.4. Renaming branches
- 3. Merging Exercises
  - 3.1. Displaying branch graphs
  - 3.2. Merging Branches
  - 3.3. Resolving Conflicts
  - 3.4. Rebasing
- 4. Remotes Exercises
  - 4.1. Initializing a remote repository
  - 4.2. Forking & Pull Requests
  - 4.3. Team Exercise: Pushing and Pulling
- 5. Security Exercises
  - 5.1. Secure access using SSH
  - 5.2. Signing commits using GPG

# 1. Basic Commands Exercises

These exercises will help you understand and get used to the basic Git commands, such as:

- `git config`
- `git add`
- `git commit`
- `git log`
- `git reset`
- `git checkout`
- `git init`
- `git rm`
- `git status`
- `git mv`

---

## 1.1. Configuration

Configure Git using the `git config` command. Set at least the following properties:

- The user.name under which you want to contribute
- The email address which other team members can use to contact you

Make sure these configuration settings are globally set.

---

## 1.2. Contents

We want to store a number of recipes into a git repository. For this there is a file available called `resources-01.txt`, which will be provided by the trainer. Open this file and notice that there are a number of recipes listed in [markdown](#) format. This is a special text format that can be easily converted to HTML. We will use this file throughout the course to provide content to our git repository.

---

## 1.3. Creating and populating a repository

- Create a new empty folder on your computer. This folder will serve as our Git repository.
- Initialize a new git repository using the `git init` command
- Add a new file `hamburger.md` with the contents of said recipe
- Use `git status` to see the state of your current repository. Notice that the file is currently listed as untracked (red).
- Add the file to staging area using `git add`. Running `git status` again shows that the file is now staged (green).
- Commit the file into the repository using `git commit`. Don't forget to add a message!

- Execute the `git log` command to see that the commit is registered in the project's history. Running `git status` again shows that the working tree is clean.
  - Repeat the procedure for the other recipes. Use `git status` and `git log` often. Also try to add two (or more) recipes in a single commit.
    - `nasi-goreng.md`
    - `spaghetti.md`
    - `nasi-goreng.md`
    - `tomato-soup.md`
    - `pancakes.md`
    - `tiramisu.md`
- 

## 1.4. Modifying the repository

Let's make some changes in our repository.

- Remove `hamburger.md` and commit the results. Use `git rm` for this.
  - Add some changes to some recipes (listed next), and commit the changes together using `git add` and `git commit`. Also use `git status` a lot while doing this.
    - Open the file `pancakes.md` with a text editor do some modifications on it:
      - Increase the number of eggs by one
      - Add additional instructions to the end, stating that "The user can top up with syrup or sugar when serving".
    - Similarly edit the recipe `nasi-goring.md` by making changes to your preferences.
  - Restructure the recipes using folders, and commit the results using `git mv`, `git add`, `git commit`:
    - Add a folder `soups` and move `tomato-soup.md` to it under a new name `tomato.md`
    - Add a folder `desserts` and move `tiramisu.md` and `pancakes.md` to it
  - Check on the progress of your project using `gitk`. This shows you a graphical view of everything you've done so far.
- 

## 1.5. Recovering some changes

Whoops, changed our minds. We didn't want to delete the `hamburger.md` recipe after all. We need to get it back.

- First use `git log` or `gitk` to find the commit id right before deleting `hamburger.md`. Take note of this ID (only the first few characters is enough).
- Checkout the `hamburger.md` recipe from that commit ID using `git checkout <id> hamburger.md`. This will recover the previously deleted `hamburger.md` recipe and keep it in



staging.

- Unstage `hamburger.md` from the staging area (green) to unstaged (red) using `git reset`. Use `git status` in between all your actions to verify what's going on.
- We want to add `hamburger.md` back into our latest version, so move it to staging again using `git add` and commit using `git commit`. The recipe is now recovered!
- "Accidentally" remove all the ingredients from `tiramisu.md` but do not commit. This was a mistake! We need to recover the ingredients! Use `git checkout tiramisu.md` to restore it. (do you know what command to use when you actually did commit it? It would be `git reset --hard HEAD~`).

## 2. Branches Exercises

These exercises will help you understand and get used Git commands related to `_branches`:

- `git branch`
- `git checkout`

For this exercise, begin with the starter provided by the teacher.

---

## 2.1. Detached HEAD

A detached HEAD occurs when you checkout a specific version (by commit ID) that does not point to the top of a branch

- Use the `git log` command to identify any commit ID that is not the most recent
- Checkout this version using `git checkout <id>`
- Use `git status` to notice that you have a "Detached HEAD" and the working tree corresponds to the checkout out version.

It is possible to add commits to a working tree, but you should always create a branch first. Otherwise the changes may (and will) be lost in the future, due to the fact that Git automatically cleans up such dangling commits. This cleanup can also be done manually using `git gc`.

---

## 2.2. Creating branches

- Display a list of all branches using `git branch`. Notice that there is only one branch called `master`. This is the default branch which all repositories have.
- Create a new branch called `new-soups` with `git branch new-soups`, and run `git branch` again. Notice that there are now two branches, but `master` is still the active one (it has a star).
- Switch to the new branch using `git checkout new-soups`, and run `git branch` to verify that the star has moved. You are now on the new branch
- Add two new soups (use the provided resource file `resources-02.txt`) and check them in as separate commits. They are now part of the `new-soups` branch.
- Switch back to the `master` branch and notice that your two new recipes have disappeared. Not to worry! They are still safe and sound in your other branch.
- On the `master` branch, perform the following changes (and commit them):
  - Modify the `tiramisu.md` recipe to replace ingredient "Rum" with "Amaretto", and use (much) more of it! For example change it to: "1 bottle of Amaretto (or Rum)"
  - Add another recipe `macaroni-and-cheese.md`
- Verify the alterante realities of your project using `git log --graph --all --oneline`. It should clearly show the "split" as follows:

```
* ba4cfa1 Added macaroni and cheese, and more alcohol to tiramisu
| * 2478b9f Added pumpkin soup
| * 5e8e758 Added chicken soup
|/
* 0c9f47a Recovered hamburger.md
* 949b622 Organised recipes in a folder
* 2168353 Did some changes to some recipes
```

---

## 2.3. Deleting branches

- Switch to branch `master` and from there create a new branch `temp`, and switch to it
- Modify `spaghetti-bolognese.md`, and remove the ingredient 2 `carrots`. Commit this change.
- Switch back to `master`
- Now try to delete branch `temp` using `git branch --delete temp`. Notice that Git refuses to do this:

```
error: The branch 'temp' is not fully merged.
```

- This happens because Git detects that the change on `spaghetti-bolognese.md` only exists in that branch. It would be lost forever. Let's delete it anyway. This can be done using `--force` or `-D`. The previous change is now permanently lost.

---

## 2.4. Renaming branches

- Let's rename our branch `new-soups` to `feature-1-additional-soups` using `git branch --move <old> <new>`. Verify using `git branch`.

```
feature-1-extra-soups
* master
```

## 3. Merging Exercises

These exercises will help you understand and get used to the concept of merging and resolving conflicts, including the following commands:

- `git merge`
- `git rebase`

For this exercise, begin with the starter provided by the teacher.

---

## 3.1. Displaying branch graphs

Use the `git log` command as illustrated to get an overview of the branches that are in the repository.

```
$ git log --decorate --all --graph --oneline
* 1f74681 (HEAD -> master) Added hint for combining spaghetti with tomato soup
* 66d723a Added salad and tomato to hamburger recipe
| * 3736788 (feature-3-appetizers) Added new appetizer: Crab Cakes
| * 41612b0 Added notice to spaghetti bolognese for combining with jalapeno poppers
| * efe0f78 Forgot to add last instruction. Fixed now.
| * b0c2062 Added preparation instructions for jalapeno poppers
| * 3aa1dec Added Jalapeno Poppers ingredients
| | * 5ecc16f (feature-2-cheezeburger) Added cheeze to the hamburger for a better
CHEEZE burger
| | /
| | /
* | 09ce258 Hamburger was empty
* | ba4cfa1 Added macaroni and cheese, and more alcohol to tiramisu
| | * 2478b9f (feature-1-extra-soups) Added pumpkin soup
| | * 5e8e758 Added chicken soup
| | /
| | /
* | 0c9f47a Recovered hamburger.md
* | 949b622 Organised recipes in a folder
* | 2168353 Did some changes to some recipes
* | 4949819 Removed unhealthy foods
| /
* 9e18710 New recipe for tomato soup added
* 89e63b2 Added some deserts
* ef3c213 Italian food is nice: spaghetti bolognese
* 99d73fb new asian recipe
* fc04aa4 Added the juiciest hamburger ever!
```

Examine this closely before continuing. Alternatively use `gitk --all`.

## 3.2. Merging Branches

- Use `git branch` and locate branches `master` and `feature-1-extra-soups`
- Switch to branch `master`
- Notice there is only one soup recipe (`soups/tomato.md`)
- Now, merge `feature-1-extra-soups` as a regular three-way-merge using `git merge`

```
feature-1-extra-soups
```

- Notice that the merge was successful, and a commit is done for you immediately. Verify this with `git log --graph`. This is because the two "alternate realities" did not contain conflicting events.

Merge made by the '**recursive**' strategy.

```
soups/chicken-soup.md | 17 ++++++
soups/pumpkin-soup.md | 21 ++++++
2 files changed, 38 insertions(+)
create mode 100644 soups/chicken-soup.md
create mode 100644 soups/pumpkin-soup.md
```

- Verify that there are now actually three soups in the master branch. This means the `feature-1-extra-soups` branch is fully integrated.
- When feature branches have been integrated, they are often removed. Do this now using `git branch --delete feature-1-extra-soups`. This should work without forcing.

### 3.3. Resolving Conflicts

- Use `git branch` to locate branches `master` and `feature-2-cheezeburger`
- Switch to branch `master`
- Merge branch `feature-2-cheezeburger` into `master`. Notice there is an error! The two "alternate realities" contain conflicting events. We need to manually resolve this problem.

Auto-merging `hamburger.md`

CONFLICT (content): Merge conflict in `hamburger.md`

Automatic merge failed; fix conflicts and **then** commit the result.

- Check which files are conflicting with `git status`
- Open `hamburger.md` with a text editor and notice that Git has added markers indicating this is a conflicting file.

```
# Hamburger
```

```
## Ingredients
```

```
- 2 pounds ground beef
- 1 egg, beaten
- 3/4 cup dry bread crumbs
- 3 tablespoons evaporated milk
- 2 tablespoons Worcestershire sauce
- 1/8 teaspoon cayenne pepper
- 2 cloves garlic, minced
```

```
<<<<<<< HEAD
```

```
- tomato and salad
```

```
=====
```

```
- 2 slices of cheeze
- 1 tomato
```

```
- salad
>>>>>> feature-2-cheezeburger

## Preparation

Preheat grill for high heat.

In a large bowl, mix the ground beef, egg, bread crumbs, evaporated milk,
Worcestershire sauce, cayenne pepper, and garlic using your hands. Form the mixture
into 8 hamburger patties.

Lightly oil the grill grate. Grill patties 5 minutes per side, or until well done.

<<<<<< HEAD
Add tomato and salad to your tasting in layers.
=====
Layer slices of cheese on top of hamburger on a sandwich, and add hamburger with
tomato, salad to your taste.
>>>>>> feature-2-cheezeburger
```

- Solve the conflict by manually selecting the parts of the recipe you want to keep. You'll have to retain some additions from both. Make sure you remove all the Git markers while doing so. There should be two conflicting changes.
- When finished, the file needs to be marked as resolved. Do this by adding the file with `git add hamburger.md` and commit the results.
- Remove the freshly integrated `feature-2-cheezeburger` branch.

## 3.4. Rebasing

- The third branch `feature-3-appetizers` needs to be also integrated into master. Locate the two branches with `git branch now`.
- This time, we'll use `git rebase` to integrate the branch. To get started with this, checkout branch `feature-3-appetizers` first. Note that when rebasing, you checkout the donor branch rather than the receiving branch (unlike `git merge`).
- With `feature-3-appetizers` checked out, perform a rebase onto master using `git rebase master`. Notice that this effectively replays three commits without problems, but the fourth has an conflict.

```
First, rewinding head to replay your work on top of it...
Applying: Added Jalapeno Poppers ingredients
Applying: Added preparation instructions for jalapeno poppers
Applying: Forgot to add last instruction. Fixed now.
Applying: Added notice to spaghetti bolognese for combining with jalapeno poppers
Using index info to reconstruct a base tree...
M       spaghetti-bolognese.md
Falling back to patching base and 3-way merge...
Auto-merging spaghetti-bolognese.md
CONFLICT (content): Merge conflict in spaghetti-bolognese.md
error: Failed to merge in the changes.
Patch failed at 0004 Added notice to spaghetti bolognese for combining with jalapeno
poppers
```



The copy of the patch that failed is found in: `.git/rebase-apply/patch`

When you have resolved this problem, run `"git rebase --continue"`.

If you prefer to skip this patch, run `"git rebase --skip"` instead.

To check out the original branch and stop rebasing, run `"git rebase --abort"`.

- You'll need to manually resolve this conflict again. Check the file with `git status`, edit as appropriate, add the change to staging using `git add`.
- The rebase can now be resumed. Do this using `git rebase --continue`. Note that this successfully rebase the conflict plus an additional commit with "Crab Cakes".

Applying: Added notice to spaghetti bolognese for combining with jalapeno poppers

Applying: Added new appetizer: Crab Cakes

- Use `git log --oneline --graph --all --decorate` and notice the `feature-3-appetizers` branch has now been "transplanted" on the top of `master`
- The result is now a fast forward capable branch, so we can easily merge this with `master`. Checkout `master` and perform a normal merge. This is guaranteed not to produce any conflicts.

Updating d38d84f..c6d4e2f

Fast-forward

```

appetizers/crab-cakes.md      | 19 ++++++
appetizers/jalapeno-poppers.md | 23 ++++++
spaghetti-bolognese.md       |  5 +++-
3 files changed, 46 insertions(+), 1 deletion(-)
create mode 100644 appetizers/crab-cakes.md
create mode 100644 appetizers/jalapeno-poppers.md

```

- Verify that you now have two appetizer recipes!
- Finally, the branch is now fully integrated and can safely be removed. Do this now.

## 4. Remotes Exercises

These exercises will make you familiar with Git's remote commands, including:

- `git fetch`
- `git push`
- `git pull`
- `git remote`

We will also explore the concept of forking. These actions will be done using GitHub, but can be performed similarly using any other remote Git hosting environment.

Note: In order to be able to do these exercises, you'll need to have internet access and register an account on your server of choice. We recommend using GitHub as a sandbox environment for this, but other servers are analogous.

---

## 4.1. Initializing a remote repository

To create a new remote repository, you'll have to use your server's web interface. The process is very similar on all servers. Here's an example using GitHub:

## Create a new repository

A repository contains all the files for your project, including the revision history.

Owner

Repository name


 kvanrobbroeck ▾ / recipes ✓

Great repository names are short and memorable. Need inspiration? How about **sturdy-chainsaw**.

Description (optional)

World's best recipes

☒  **Public**  
Anyone can see this repository. You choose who can commit.

☐  **Private**  
You choose who can see and commit to this repository.

☒ **Initialize this repository with a README**

This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: **None** ▾ | Add a license: **GNU General Public License v3.0** ▾ ⓘ

Create repository

Once the (empty) repository is created you can clone it to your local hard drive using `git clone`. This allows you to start working on a new project that's linked up to the server automatically. This is the advised way when starting a new project and you already know you're going to use the server.

```
$ git clone https://github.com/jimi/recipes.git
$ cd recipes      # Now inside empty repository
```

Alternatively, you can upload an existing project from your local hard drive into the repository. Let's do that with our project:

- Create a new repository using your preferred server. We advice to use GitHub
- Add a new remote to your existing project

```
# First cd into your existing project
$ git remote add origin https://github.com/jimi/recipes.git # Now linked up to the
server
$ git pull origin master                                     # Make sure you're up to
date (should be the case)
```

```
$ git push origin master
branch to the remote!
```

```
# Upload your master
```

- Now check your changes are indeed uploaded by using the server's web interface:

kvanrobbroeck Merge branch 'master' of github.com:kvanrobbroeck/git-course-recipes		Latest commit 5eee349 37 seconds ago
appetizers	Added new appetizer: Crab Cakes	4 days ago
desserts	Added macaroni and cheese, and more alcohol to tiramisu	4 days ago
soups	Added pumpkin soup	4 days ago
LICENSE	Initial commit	2 minutes ago
README.md	Initial commit	2 minutes ago
hamburger.md	Integrated branch feature-2-cheeze-burger	4 days ago
macaroni-and-cheese.md	Added macaroni and cheese, and more alcohol to tiramisu	4 days ago
nasi-goreng.md	Did some changes to some recipes	4 days ago
spaghetti-bolognese.md	Added notice to spaghetti bolognese for combining with jalapeno poppers	4 days ago

- Make sure to solve any issues you may encounter!

## 4.2. Forking & Pull Requests

You can collaborate on an existing project owned by someone else by making a fork. This is a copy of the repository on which you have write access.

- Fork the following repository: <https://github.com/kvanrobbroeck/git-course-recipes>. This is done using the server's web interface:



- Explore the project. Notice in `macaroni-and-cheese.md`, the preparation instructions mentions Preheat oven to 350 degrees F. Since we're living in Europe, we prefer to have this in °C. Let's change this to 175 degrees C.
  - First create a new branch `ftoc` to do your changes on. This will make it easier to create a pull request later!
  - Do the modification, and commit the file
  - Push the change to the server using `git push origin ftoc`
- The change is now pushed to your forked repository. However, it is not part of the source repository, since you are not the owner of this repository you can't just push into it directly. You'll need to create a pull request for this, which is done using the server's web interface. Here's what that looks like using GitHub:

## Comparing changes

Choose two branches to see what's changed or to start a new pull request. If you need to, you can also [compare across forks](#).

base fork: [kvanrobbroeck/git-course-recipes](#) base: [master](#) ... head fork: [Realdolmen-Academics-2016...](#) compare: [master](#)

✓ **Able to merge.** These branches can be automatically merged.

[Create pull request](#) Discuss and review the changes in this comparison with others.

1 commit 1 file changed 0 commit comments 1 contributor

Commits on Nov 22, 2016



[kvanrobbroeck](#) Update macaroni-and-cheese.md 89a4323

Showing 1 changed file with 1 addition and 1 deletion. Unified Split

2 macaroni-and-cheese.md

@@ -13,7 +13,7 @@			
13		13	
14	## Preparation	14	## Preparation
15		15	
16	-Preheat oven to 350 degrees F.	16	+Preheat oven to 175 degrees C.
17		17	
18	Place bacon in a large, deep skillet. Cook over medium high heat until evenly brown. Drain, crumble and set aside.	18	Place bacon in a large, deep skillet. Cook over medium high heat until evenly brown. Drain, crumble and set aside.
19		19	

- On the receiving end, the pull request can then be accepted (or rejected).

  **This branch has no conflicts with the base branch**  
Merging can be performed automatically.

[Merge pull request](#) You can also [open this in GitHub Desktop](#) or view [command line instructions](#).

Note: it is a good idea to add some comments to the pull request, so that a discussion can be started around it. Otherwise the other party is not likely to accept it.

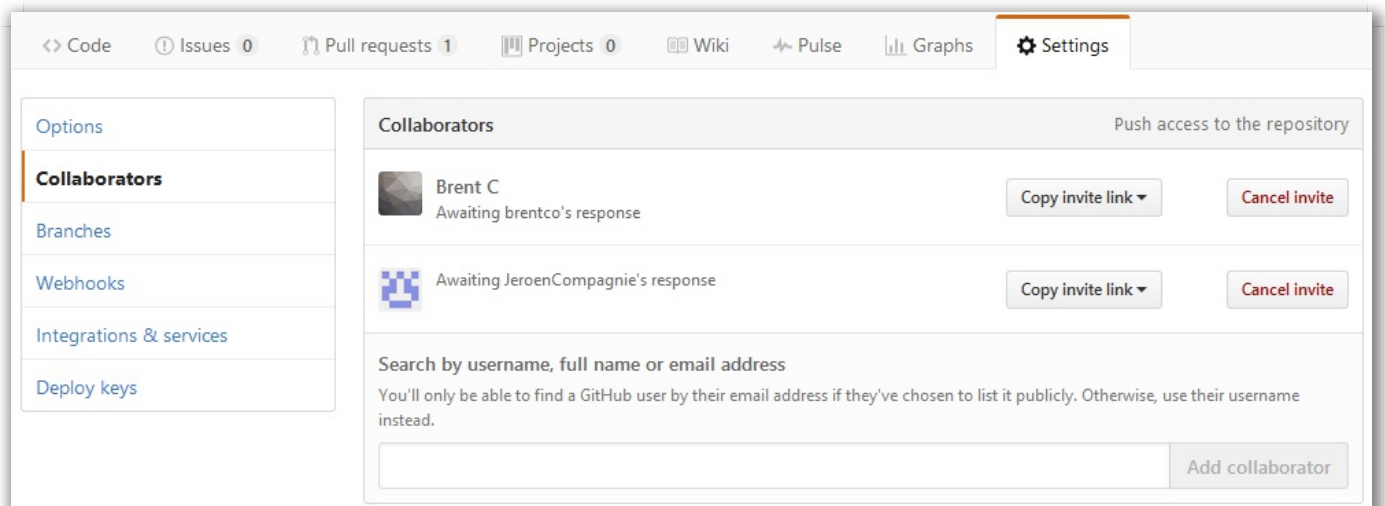
## 4.3. Team Exercise: Pushing and Pulling

This exercise will show the process of collaboration with different team members on a remote repository.

- Team up with the person next to you! You'll be sharing the same repository, so decide on either one of your repositories to use. If you don't have a repository yet from a previous exercise, one of the two team members can start by forking

<https://github.com/kvanrobbroeck/git-course-recipes>

- Person 1 (owner):
  - To be able to share the same repository, the repository's owner needs to add access for the other team member. This is done using the server's web interface. Here's how that's done using GitHub:



- Person 2 (collaborator):
  - Accept the collaboration request sent to you (by email) by your team member.
- You can now collaborate on the same project. Clone (don't fork) your shared repository to your local hard drive
- We will now start a little race to be the first to push a change:
- Person 1
  - In `tiramisu.md`, change the line `1 pound mascarpone cheese` into `500g mascarpone cheese`
  - Commit your changes
  - Push your changes to the server (origin)
- Person 2
  - In `tiramisu.md`, change the line `3/4 cup white sugar` into `85g white sugar`
  - Commit your changes
  - Push your changes to the server (origin)
- You will notice that the winner (the first one to push the change) will have no issues, but the second person does! He/she will have to first pull the changes from the server, integrate (merge) that locally, and then push again.

## 5. Security Exercises



In these exercises, we follow the instructions provided in the presentation to configure Git to work with SSH connections and GPG signing keys. We use GitHub as the example. Other available cloud services or privately hosted servers work in very similar fashion.

This includes the following commands:

- `gpg`
- `ssh-keygen`
- `git commit --gpg-sign`
- `git clone`

---

## 5.1. Secure access using SSH

Follow the instructions laid out in the presentation to configure Git to work with SSH.

- Generate a new SSH key using `ssh-keygen`. Provide a strong password for your private key
- Locate the newly generated public and private keyfile (defaults to `~/.ssh/id_rsa` and `~/.ssh/id_rsa.pub`)
- Dump the contents of the public key file using `cat ~/.ssh/id_rsa.pub` and copy this in your clipboard
- Log on to your GitHub profile. (It is wise to configure GitHub to use 2-factor-authentication!)
- Go to your account settings page, and locate the option "SSH and GPG keys"
- Add the SSH key and associate this with a label (this is just a mnemonic)
- Now clone a GitHub repository using the SSH protocol (`git@github.com:<username>/<repository>.git`) rather than HTTPS

---

## 5.2. Signing commits using GPG

Follow the instructions laid out in the presentation to configure Git to sign commits with GPG keys.

- Generate a new GPG key using `gpg --gen-key`. Use RSA and provide the information asked for
- Configure Git to use your generated signing key with `git config --global user.signingkey <key_id>`
- Dump the contents of your public GPG key using `gpg --export --armor <your_email>` and copy this to your clipboard
- Log on to your GitHub profile. (It is wise to configure GitHub to use 2-factor-authentication!)
- Go to your account settings page, and locate the option "SSH and GPG keys" and paste

the GPG key there

- You can now start signing commits using `git commit --gpg-sign ...`
- Verify that your commit is signed using `git log --show-signature`
- Push some changes to GitHub and check that GitHub now recognizes the commit as a "verified commit"
- Distribute your public key to friends and colleagues. Public keys are safe to show to anyone (including your worst enemies!)