Gitting Started

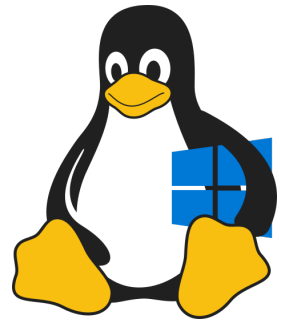
An Introduction to Git

K. Müller[1] J. Verzijden[2]

[1]MasterCLASS
E.T.S.V. Scintilla

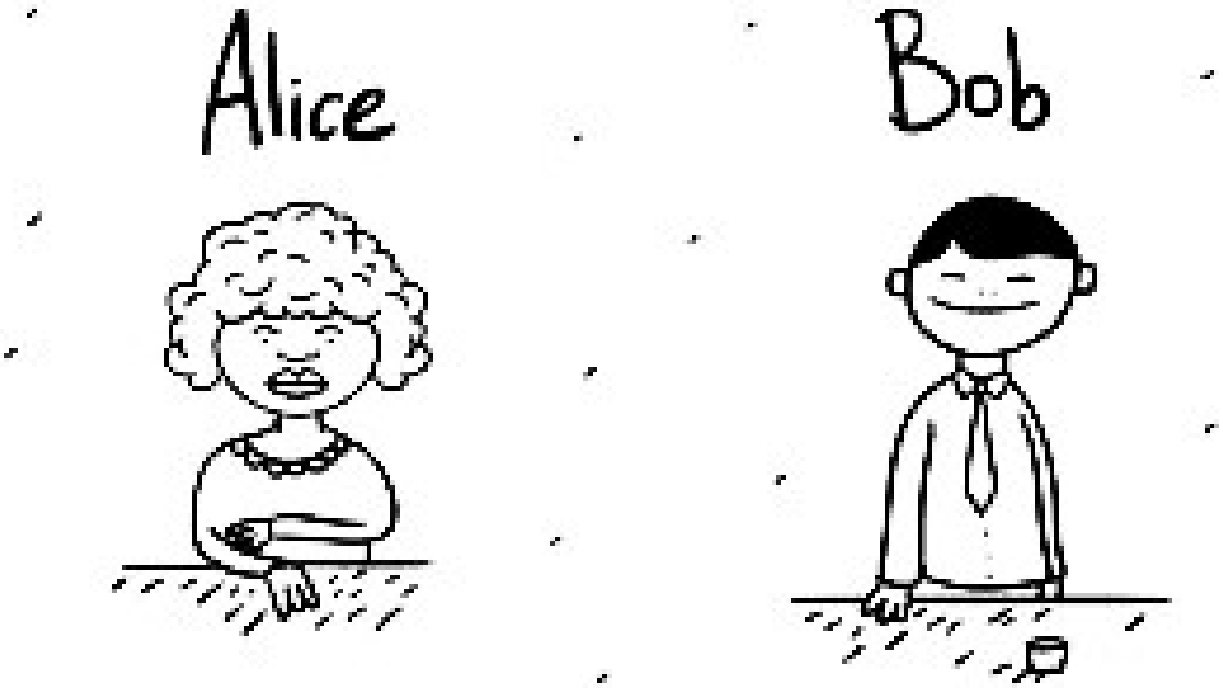
[2]Scintilla Operator Team
E.T.S.V. Scintilla

October 14, 2025



But first... Dinner!

Let's imagine...



About us

Kasper Müller
kasperm@scintilla.utwente.nl

Johan Verzijden
johanv@scintilla.utwente.nl

Contents

Contents

Why Git?

- Solves the problems from the intro.

Why Git?

- Solves the problems from the intro.
- It's efficient and fast

Why Git?

- Solves the problems from the intro.
- It's efficient and fast
- It's the most used VCS[a] in the world[b]

[a] Version Control System

[b] According to Stackoverflow, see here (2023)

Version control systems used by responding developers:

Name	2015	2017	2018	2022
Git	69.3%	69.2%	87.2%	93.9%
Subversion	36.9%	9.1%	16.1%	5.2%
TFVC	12.2%	7.3%	10.9%	[ii]
Mercurial	7.9%	1.9%	3.6%	1.1%
CVS	4.2%	[ii]	[ii]	[ii]
Perforce	3.3%	[ii]	[ii]	[ii]
VSS	[ii]	0.6%	[ii]	[ii]
ClearCase	[ii]	0.4%	[ii]	[ii]
Zip file backups	[ii]	2.0%	7.9%	[ii]
Raw network sharing	[ii]	1.7%	7.9%	[ii]
Other	5.8%	3.0%	[ii]	[ii]
None	9.3%	4.8%	4.8%	4.3%

VCS usage according to a survey by Stack Overflow[a]

[a] Source: Git on Wikipedia

The origin story

- Linus Torvalds[a]

The origin story

- Linus Torvalds[a]
- 2005

The origin story

- Linus Torvalds[a]
- 2005
- VCS for the Linux kernel

The origin story

- Linus Torvalds[a]
- 2005
- VCS for the Linux kernel
- *The stupid content tracker*

[a] Source of the photo: WikiMedia

Concepts of Git

[1] *repository* on Wiktionary

Concepts of Git

- Repository (repo)
  - A location for storage[1]
  - In this case, storage of files
  - Just a folder on your computer, e.g. `/home/johanv/my-repo`

[1] *repository* on Wiktionary

Concepts of Git

- Repository (repo)
  - A location for storage[1]
  - In this case, storage of files
  - Just a folder on your computer, e.g. `/home/johanv/my-repo`
- Commit
  - Description of changes
  - Metadata: author, timestamp, message, parent
  - Hash: Its unique identifier

[1] *repository* on Wiktionary

Concepts of Git

- Repository (repo)
  - A location for storage[1]
  - In this case, storage of files
  - Just a folder on your computer, e.g. `/home/johanv/my-repo`
- Commit
  - Description of changes
  - Metadata: author, timestamp, message, parent
  - Hash: Its unique identifier
- Branch
  - A list of commits belonging together
  - A tag points to the head of the branch (the last commit)

[1] *repository* on Wiktionary

Concepts of Git

- Repository (repo)
  - A location for storage[1]
  - In this case, storage of files
  - Just a folder on your computer, e.g. `/home/johanv/my-repo`
- Commit
  - Description of changes
  - Metadata: author, timestamp, message, parent
  - Hash: Its unique identifier
- Branch
  - A list of commits belonging together
  - A tag points to the head of the branch (the last commit)
- Remote
  - The same repository in a different location
    - Really remote (`https://github.com/instructure/canvas-lms`)
    - Locally remote (`/home/kasperm/some-repo/`)

[1] *repository* on Wiktionary

Principles of Git

Principles of Git

- Recording small sets of changes, not versions

Principles of Git

- Recording small sets of changes, not versions
- Meant for text files, not for binary files

Principles of Git

- Recording small sets of changes, not versions
- Meant for text files, not for binary files
- Meant for source files, not compiled files

Principles of Git

- Recording small sets of changes, not versions
- Meant for text files, not for binary files
- Meant for source files, not compiled files
- Make a new branch for each feature

Principles of Git

- Recording small sets of changes, not versions
- Meant for text files, not for binary files
- Meant for source files, not compiled files
- Make a new branch for each feature
- Branch names should be descriptive
  - ✔ *create-home-page*
  - ✗ *branch15*

Principles of Git

- Recording small sets of changes, not versions
- Meant for text files, not for binary files
- Meant for source files, not compiled files
- Make a new branch for each feature
- Branch names should be descriptive
  - ✔ *create-home-page*
  - ✗ *branch15*
- Commits should contain small, logical changes

Principles of Git

- Recording small sets of changes, not versions
- Meant for text files, not for binary files
- Meant for source files, not compiled files
- Make a new branch for each feature
- Branch names should be descriptive
  - ✔ *create-home-page*
  - ✗ *branch15*
- Commits should contain small, logical changes
- Commit messages should be descriptive
  - ✔ *Fix styling of logout button*
  - ✗ *fix*
  - ✗ *Meep*

Principles of Git

- Recording small sets of changes, not versions
- Meant for text files, not for binary files
- Meant for source files, not compiled files
- Make a new branch for each feature
- Branch names should be descriptive
  - ✔ *create-home-page*
  - ✗ *branch15*
- Commits should contain small, logical changes
- Commit messages should be descriptive
  - ✔ *Fix styling of logout button*
  - ✗ *fix*
  - ✗ *Meep*
- Commit messages should be in imperative tense
  - ✔ *Add page layout*
  - ✗ *Added page layout*
  - ✗ *Adding page layout*

Ignoring certain files - Gitignore

In git you only want to track source files

Ignoring certain files - Gitignore

In git you only want to track source files

- Changes in compiled or other binary files can't be tracked

Ignoring certain files - Gitignore

In git you only want to track source files

- Changes in compiled or other binary files can't be tracked
- Build artifacts clutter your commits

Ignoring certain files - Gitignore

In git you only want to track source files

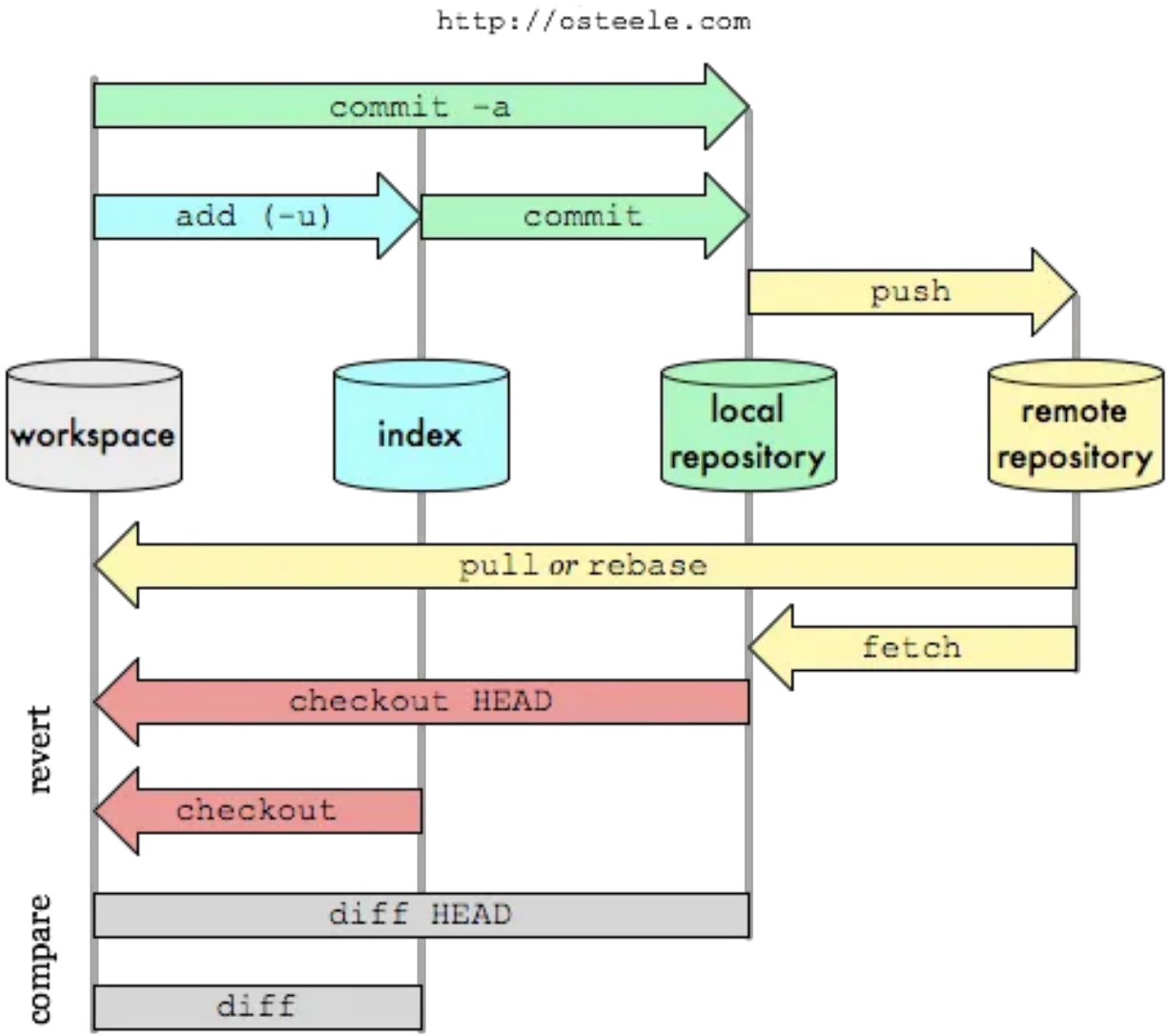
- Changes in compiled or other binary files can't be tracked
- Build artifacts clutter your commits

Solution: `.gitignore`

.gitignore

```
# IDE's project files
.idea

# windows thumbnail cache
Thumbs.db

# composer vendor dir
/vendor

# Mac DS_Store Files
.DS_Store

# Large output files
build/output/*

# machine-specific configuration files
.env
config/dev-env.conf
```

Git Workflow

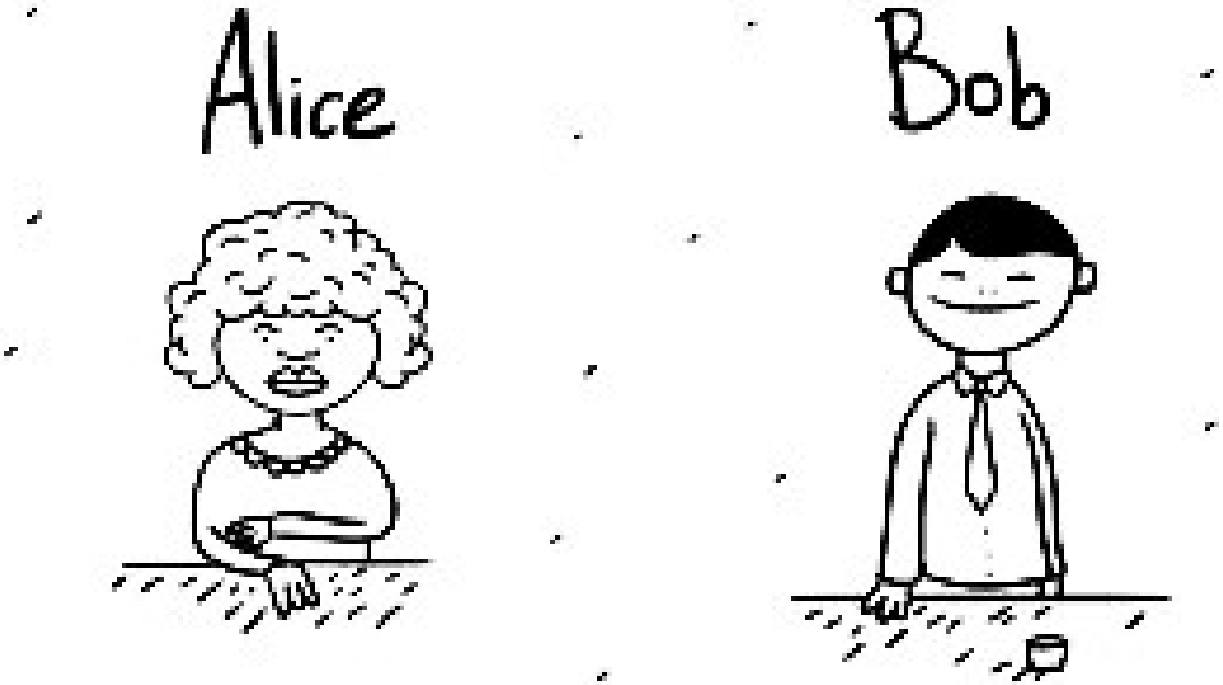


Let's take another look (at the situation)

Let's see how Git actually solves the problem in the introduction.

Let's take another look (at the situation)

Let's see how Git actually solves the problem in the introduction.



Contents

Installation - GUI

[2] Graphical User Interface

Installation - GUI

Some options:

- Pure Git GUI[2] clients:

[2] Graphical User Interface

Installation - GUI

Some options:

- Pure Git GUI[2] clients:
  - git-gui & gitk (comes with the Git CLI)

[2] Graphical User Interface

Installation - GUI

Some options:

- Pure Git GUI[2] clients:
  - git-gui & gitk (comes with the Git CLI)
  - Github Desktop (only for Mac & Windows)

[2] Graphical User Interface

Installation - GUI

Some options:

- Pure Git GUI[2] clients:
  - git-gui & gitk (comes with the Git CLI)
  - Github Desktop (only for Mac & Windows)
  - gitg (only for Linux & Windows)

[2] Graphical User Interface

Installation - GUI

Some options:

- Pure Git GUI[2] clients:
  - git-gui & gitk (comes with the Git CLI)
  - Github Desktop (only for Mac & Windows)
  - gitg (only for Linux & Windows)
  - See https://git-scm.com/downloads/guis for more

[2] Graphical User Interface

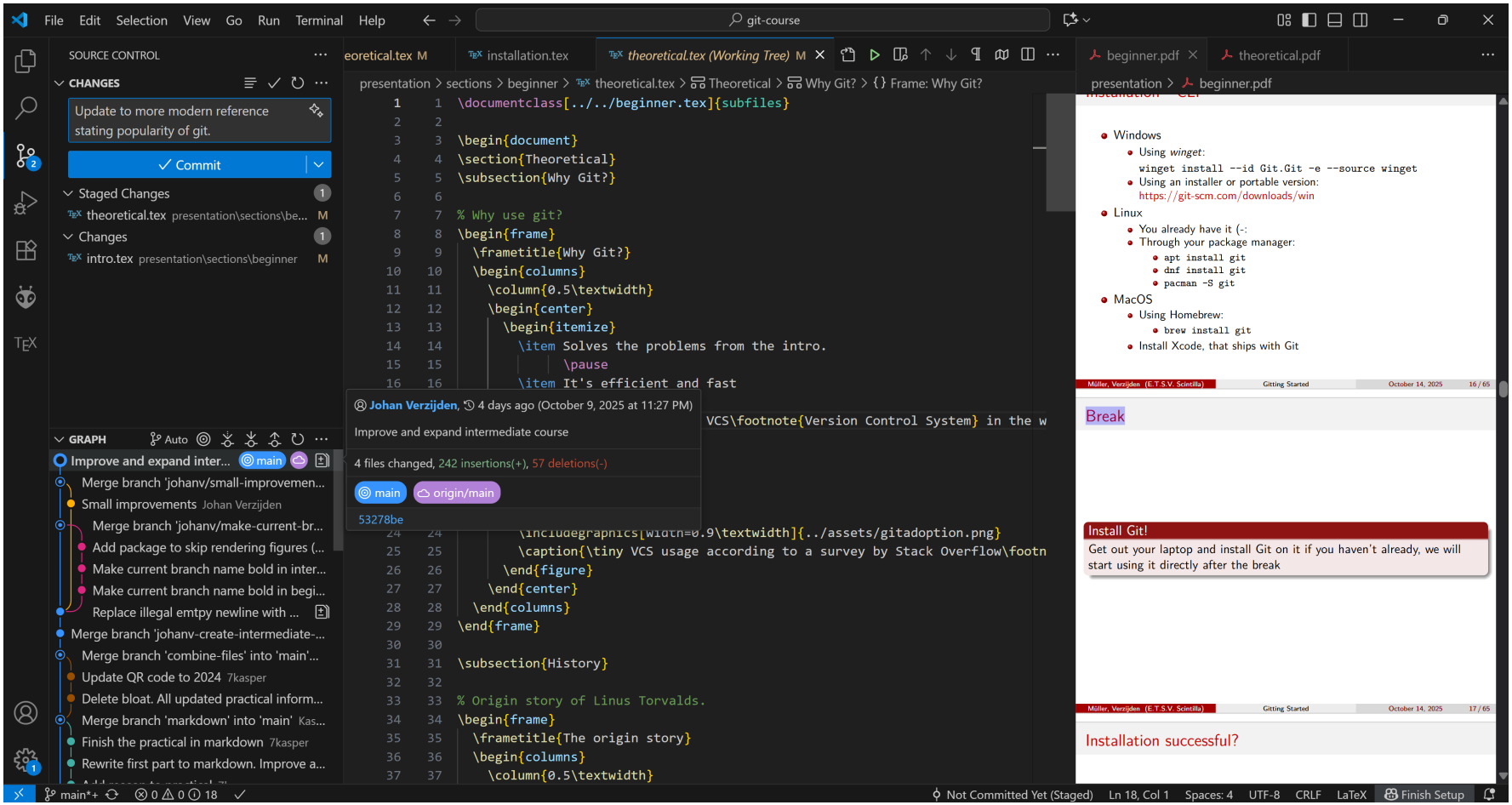
Installation - GUI

Some options:

- Pure Git GUI[2] clients:
  - git-gui & gitk (comes with the Git CLI)
  - Github Desktop (only for Mac & Windows)
  - gitg (only for Linux & Windows)
  - See https://git-scm.com/downloads/guis for more
- Most IDEs have a Git GUI built-in

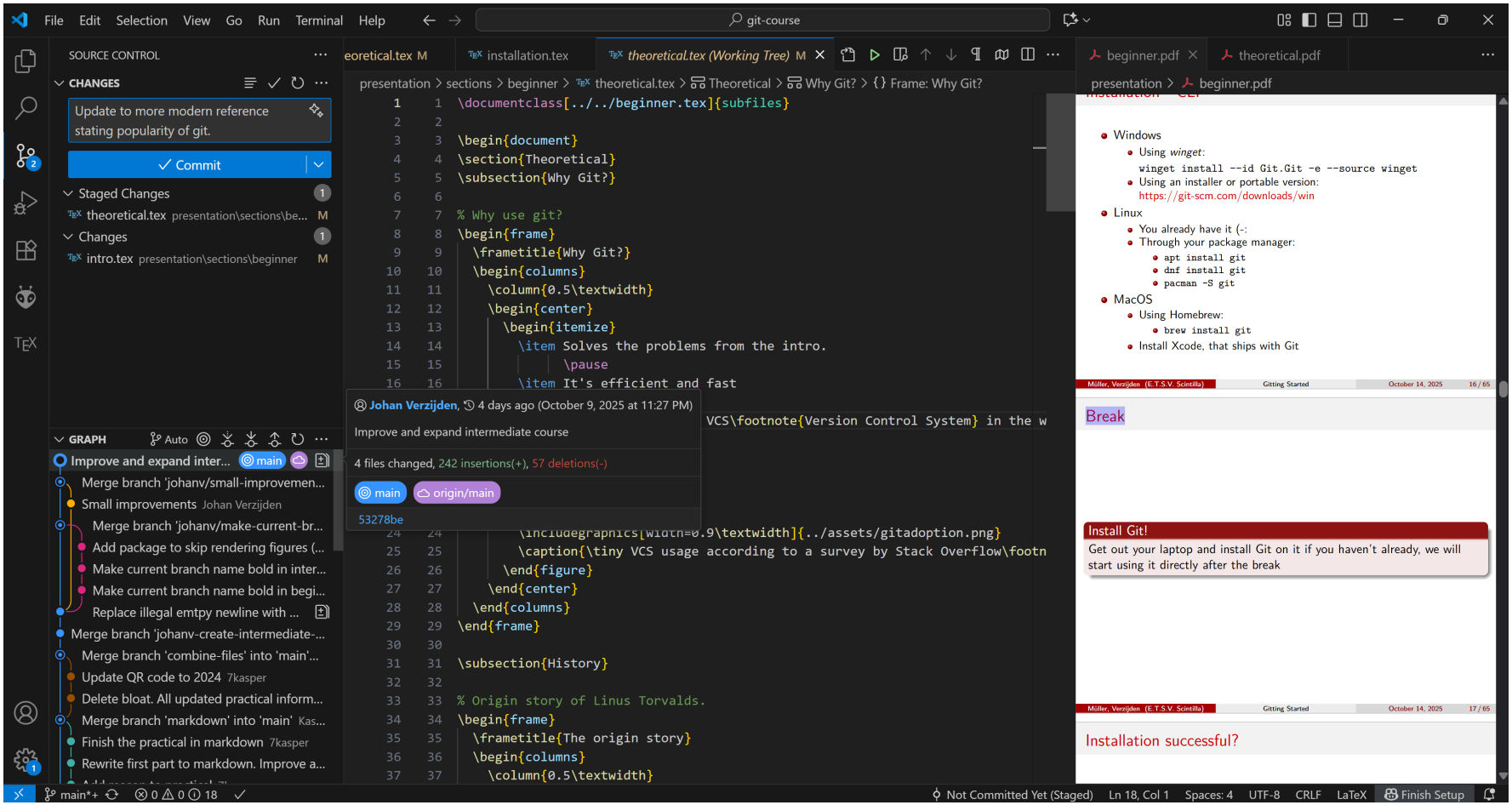
[2] Graphical User Interface

Installation - GUI Showcase



[3] Command Line Interface

Installation - GUI Showcase



Nice and all, but we will use the CLI[3].

[3] Command Line Interface

Installation - CLI

Installation - CLI

- Windows
  - Using *winget*:

    ```
    winget install --id Git.Git -e --source winget
    ```
  - Using an installer or portable version:
    https://git-scm.com/downloads/win

Installation - CLI

- Windows
  - Using *winget*:

    ```
    winget install --id Git.Git -e --source winget
    ```
  - Using an installer or portable version:
    https://git-scm.com/downloads/win
- Linux
  - You already have it (-:
  - Through your package manager:
    - `apt install git`
    - `dnf install git`
    - `pacman -S git`

Installation - CLI

- Windows
  - Using *winget*:

    ```
    winget install --id Git.Git -e --source winget
    ```
  - Using an installer or portable version:
    https://git-scm.com/downloads/win
- Linux
  - You already have it (-:
  - Through your package manager:
    - `apt install git`
    - `dnf install git`
    - `pacman -S git`
- MacOS
  - Using Homebrew:
    - `brew install git`
  - Install Xcode, that ships with Git

5 minutes

Install Git!

Get out your laptop and install Git on it if you haven't already, we will start using it after 5 minutes

Installation successful?

Installation successful?

```
johanv@my-machine: ~$ git
usage: git [-v | --version] [-h | --help] [-C <path>] [-c <name>=<value>]
           [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
           [-p | --paginate | -P | --no-pager] [--no-replace-objects] [--bare]
           [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
           [--super-prefix=<path>] [--config-env=<name>=<envvar>]
           <command> [<args>]

These are common Git commands used in various situations:

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

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

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

Contents

Configuration

Why:

- Let git know who's editing.
- Let git know how to do some things.
- Usually only change after setup.

How:

- Using commands:

```
git config --global core.editor 'nano'
git config --global <setting> <value>
```

What:

Setting	Advised value
core.editor	nano / <preferred editor>
user.name	<your name>
user.email	<your email address>
user.useConfigOnly	true
init.defaultBranch	main
pull.ff	true

Configuration - File structure

Read `git help config` or `man git config` for all configuration options.

You can also edit using an editor:
`git config --global --edit`:

```
[core]
  editor = nano
[init]
  defaultBranch = main
[user]
  useConfigOnly = true
  name = Alice
  email = alice@example.com
[pull]
  ff = true
```

Contents

What else is possible?

- Working securely with SSH and GPG keys
- Working with different identities (e.g. work and personal)
- Quickly stashing changes to switch where you are working on.
- Finding bugs with a binary search (bisect)
- Getting specific features from branches (cherry-pick)
- Neatly organising history (rebase)
- Including other repositories as dependancies (submodules)

What else is possible?

- Working securely with SSH and GPG keys
- Working with different identities (e.g. work and personal)
- Quickly stashing changes to switch where you are working on.
- Finding bugs with a binary search (bisect)
- Getting specific features from branches (cherry-pick)
- Neatly organising history (rebase)
- Including other repositories as dependancies (submodules)

Come back next year for the intermediate course! :-)

Common Git Commands - Cheat sheet

`git ...`

- `init`
- `add [<filenames>]`
- `commit [-m '<message>']`
- `switch [-c] <branch name>`
- `log [ --graph --oneline]`
- `push`
- `pull`
- `fetch`
- `clone <path/URL>`
- `blame <file> [--color-by-age]`
- `revert <commit>`
- `reset [--hard] <commit>`

Sources for future reference

- `git help <command>`
- `man gittutorial`
- gitimmersion.com

Contact details

Think of a question later on? Feel free to reach out to us!

MasterCLASS
masterclass@scintilla.utwente.nl

Kasper Müller
kasperm@scintilla.utwente.nl

Johan Verzijden
johanv@scintilla.utwente.nl

Contents

Practical - Goals

- Show the typical flow of working with git.
- Show you most important commands and their uses in a short time
- Show a bit of markdown and a summary for optimal learning result ;-)

Practical - Documentation

We are taking a break now. Feel free to continue installing and configuring now. The information is also included with the QR below. After the short break we will both help with questions regarding anything :-)

Feel free to work on your own or in pairs. If there are many questions on a certain topic we might do a class-wide explanation.

docs.scintilla.utwente.nl/masterclass/GitCourse2025

Tutorial

Starting a new project

Using Git Bash or other terminal we start by creating a regular folder to hold our project. Then setup the repository by doing an init.

```
$ mkdir my-summary
$ cd my-summary
$ git init
```

README.md

Let's start our project with a file. We will create the file *README.md*. This is a markdown type file. Markdown is a simple filetype that allows for formatted text. It is likely you already know quite a bit of markdown as it is also used in for instance Discord and (to a lesser extend) in WhatsApp.

The *README.md* file is special. For developers this is often an entry point to understand what the repository is about. On many websites such as Gitlab the *README.md* file will also be shown (formatted) on the front page of your repository website.

Your first file

Let's add some content to the *README.md* file:

```
# Git Course Summary
This _README.md_ file will contain some markdown
    text detailing what I learned during the Git
    Course.
```

We can now save the file. Note that at this point the file is only changed in your *workspace*.

Your first commit

- We move the created file to the index:
  `$ git add README.md`
- And we do our first commit!
  `$ git commit -m 'Initial commit'`

Your first commit

- We move the created file to the index:
  `$ git add README.md`
- And we do our first commit!
  `$ git commit -m 'Initial commit'`
- On the left you see the commit tree. Currently we are on the first commit inside the main branch.
- You can also see your first commit:
  `$ git log`
  `$ git log --oneline`

Changing something

Let's add some text to the *README.md* file:

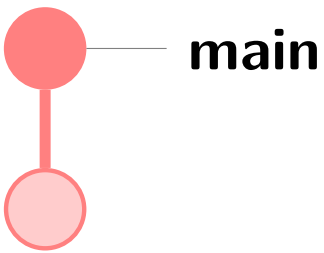
```
[...]
## How to commit
1. Add, delete or change files
2. You can stage your files by doing `git add
   filename` or stage everything with `git add -A
   `
3. Commit using `git commit -m 'message'`, be
   sure to provide a good description of the
   changes.
```

We can now save the file. Note that at this point the file is again only changed in your *workspace*. Try to add it to the index and then make a second commit.

Your second commit

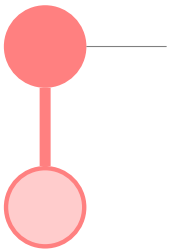
- We can move all files onto the index by using:
  `$ git add -A`
- And peform the second commit:
  `$ git commit -m 'Add the section: how to commit'`

Your second commit



- We can move all files onto the index by using:
  `$ git add -A`
- And peform the second commit:
  `$ git commit -m 'Add the section: how to commit'`
- If everything went well we now have the tree on the left. The reference HEAD is now pointing to the second commit.

Branching out

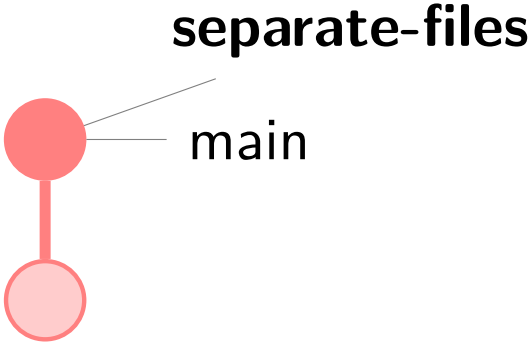


It is useful to develop larger features in a seperate branch of your repository. This helps working together as well as giving the ability to switch between versions of your codebase.

Let's say we want to move our 'How to commit' section to a different file. We start by switching to a new branch:
(the -c argument means create)

```
$ git switch -c separate-files
```

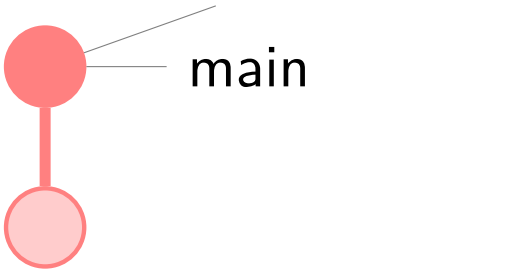
A new branch



The create command bases the new branch on the branch you were in. So at this moment the *separate-files* branch and the *main* branch are identical. We did switch to the *separate-files* branch, this is also visible in the terminal. Any new commits will be pushed to the current branch.

Let's give that a try!

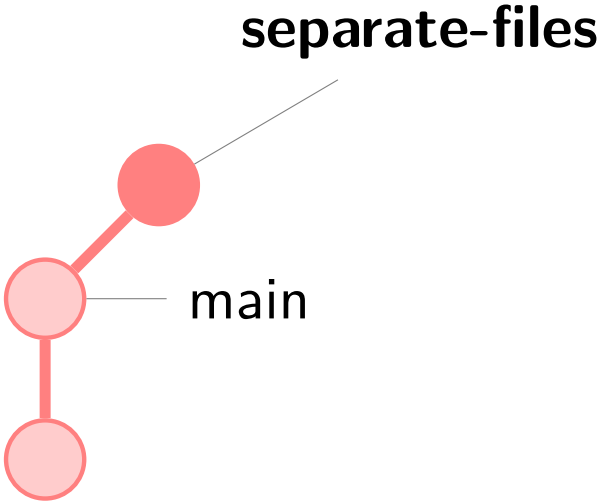
Separating the files



- Make a new file *TUTORIAL.md* and copy the *How to commit* section over from *README.md*.
- Delete that part in the *README.md* file.
- Commit all changes:

```
$ git add -A'
$ git commit -m 'Move "How to commit"
 to TUTORIAL.md'
```

Separating the files

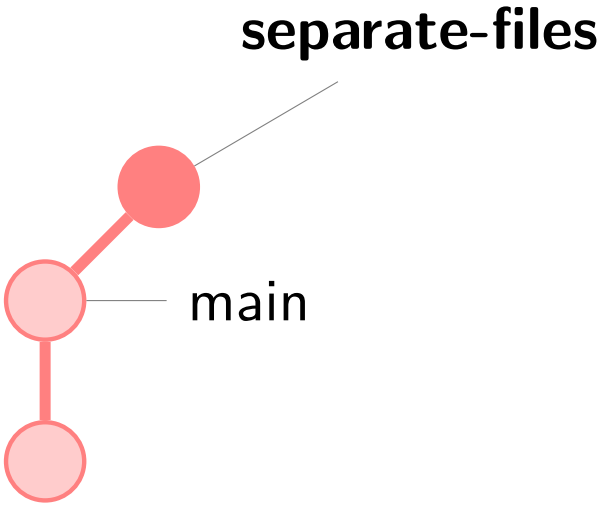


- Make a new file *TUTORIAL.md* and copy the *How to commit* section over from *README.md*.
- Delete that part in the *README.md* file.
- Commit all changes:

```
$ git add -A'
$ git commit -m 'Move "How to commit"
 to TUTORIAL.md'
```

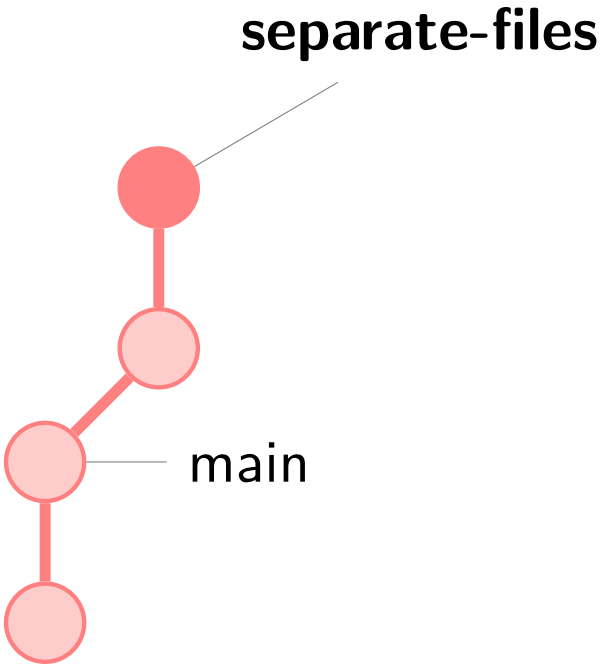
- The *separate-files* branch now has an extra commit on it and is no longer the same as the *main* branch.

More commits



- Add a small header in *TUTORIAL.md* explaining what is in the file.
- Commit this change:
  `$ git commit -am 'Add file info for TUTORIAL.md'`
  Note: the -a option on git commit automatically adds all changed files to the index, but it does ***not*** *track new files!*

More commits



- Add a small header in *TUTORIAL.md* explaining what is in the file.
- Commit this change:
  `$ git commit -am 'Add file info for TUTORIAL.md'`
  *Note: the -a option on git commit automatically adds all changed files to the index, but it does **not** track new files!*
- The branch now has 2 extra commits.
- Change and save something in the *README.md* file.
- Now we want to edit something unrelated to the separate files. We try switching back to the main branch, can you?
  `$ git switch main`

Switching branches

You cannot switch branches when your workspace is not 'clean'. You basically have a few options at this point.

1. Commit the last change of your workspace to the branch.
2. Clean all files in your workspace to the last commit (i.e. delete all changes since the last commit):
   `$ git reset --hard`
3. Stash your changes (explained in intermediate course)

Switching branches

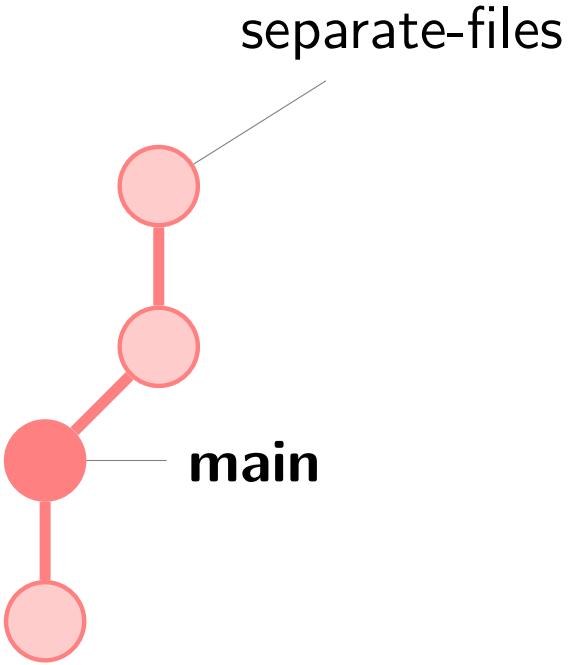
You cannot switch branches when your workspace is not 'clean'. You basically have a few options at this point.

1. Commit the last change of your workspace to the branch.
2. Clean all files in your workspace to the last commit (i.e. delete all changes since the last commit):
   `$ git reset --hard`
3. Stash your changes (explained in intermediate course)

Try the second option and switch again using:

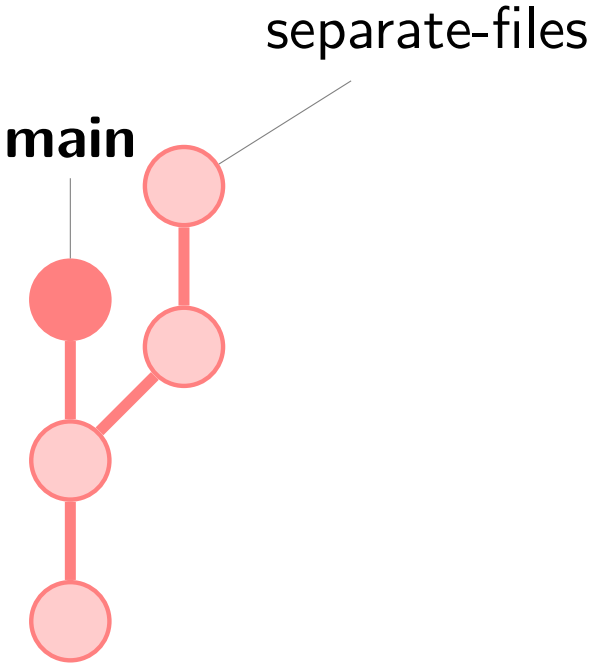
```
$ git switch main
```

Pushing to main



- Your reference is now back at the *main* branch. Notice how your files have also reverted back. *Note: Do not worry, you can freely switch between the branches. As long as you don't reset or clean everything no code is ever lost when using git.*
- Let's add a commit on the main branch. Go into *README.md* and add:
  `**TUTORIAL.md** - Basic tutorial on how to commit.`
- Commit changes:
  `$ git commit -am 'Add file description for TUTORIAL.md'`

Pushing to main



- Your reference is now back at the *main* branch. Notice how your files have also reverted back. *Note: Do not worry, you can freely switch between the branches. As long as you don't reset or clean everything no code is ever lost when using git.*
- Let's add a commit on the main branch. Go into *README.md* and add:
  `**TUTORIAL.md** - Basic tutorial on how to commit.`
- Commit changes:
  `$ git commit -am 'Add file description for TUTORIAL.md'`
- The *main* branch now has another commit.

Merging

Git allows us to do non-linear code editing. We can keep pushing different features to the separate branches. We can switch, compare and do all sorts of different things with this.

Eventually there comes a point where we might want to get all changes of a branch (for instance separating the *TUTORIAL.md* file) into another branch (for instance the *main* branch). We do this with merging.

Merging

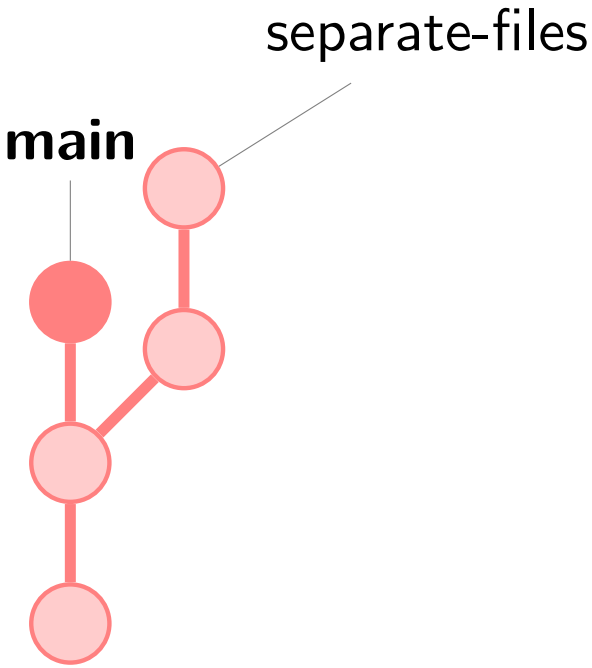
Git allows us to do non-linear code editing. We can keep pushing different features to the separate branches. We can switch, compare and do all sorts of different things with this.

Eventually there comes a point where we might want to get all changes of a branch (for instance separating the *TUTORIAL.md* file) into another branch (for instance the *main* branch). We do this with merging.

To merge the *separate-files* branch onto the current (*main*) branch perform:

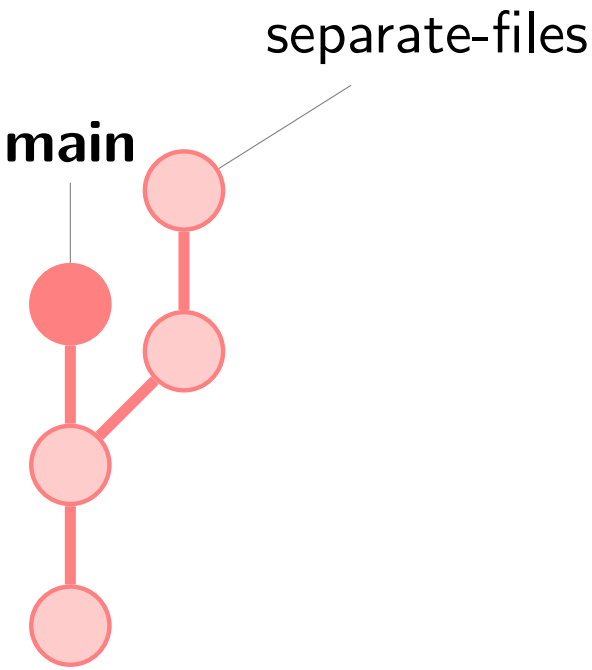
```
$ git merge separate-files
```

Merging - Conflict



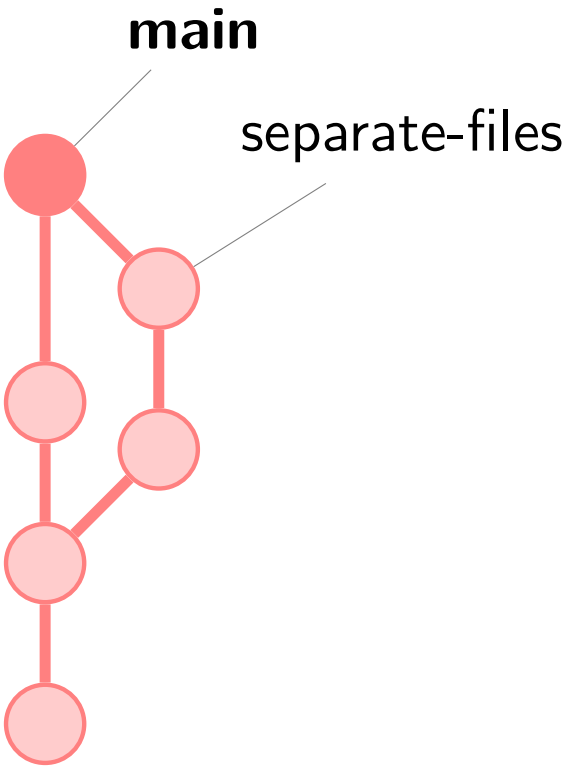
- Merging a branch can go automagically. But not now. Since both *separate-files* and *main* have more recent changes to *README.md* we have a conflict that needs to be resolved.
- Go through *README.md* and find the conflict. With your IDE or by deleting the GIT messages you can fix your code to get the wanted result from both branches.
- When you are done fixing the *README.md* file add it to the index:
  `$ git add -A`

Merging - Conflict



- Now continue the merge:
  `$ git merge --continue`
  Note: When all conflicts are resolved you will be asked to provide a message. Providing this message (by closing the file) will make a new merge commit.

Merging - Conflict



- Now continue the merge:
  `$ git merge --continue`
  Note: When all conflicts are resolved you will be asked to provide a message. Providing this message (by closing the file) will make a new merge commit.
- We now see a new commit on *main* and all the history of *seperate-files* is also pulled in.

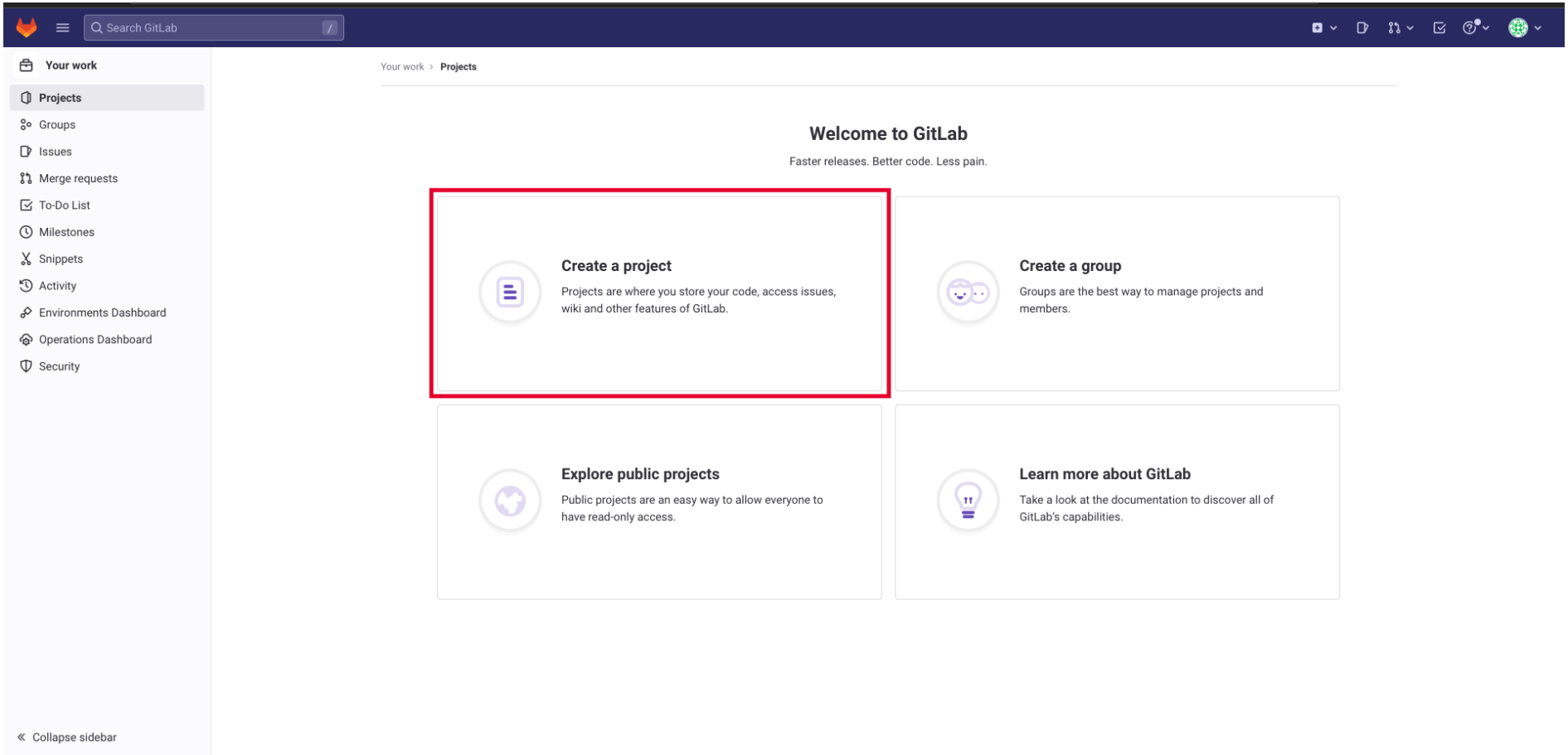
Working together

Now we have the basics of comitting and branches down. A real power of Git is working together with other people. Git allows us to clone a remote repository and then push our commits and branches to that. Any repository can be setup as remote if there is Git server running or SSH access to it. However we almost always see a dedicated git server in use.

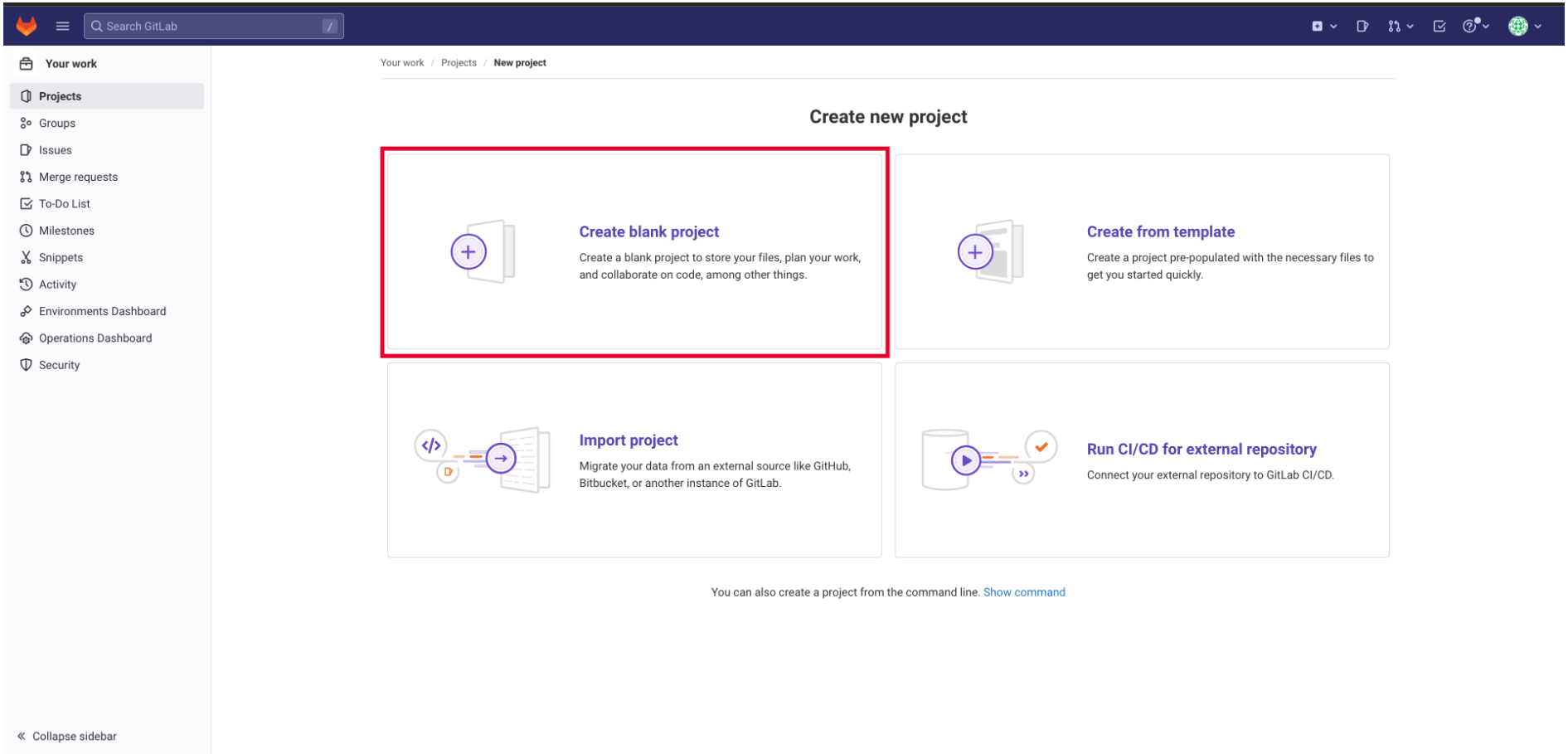
For open source projects we usually see the remote repository hosted on a public website such as `https://github.com`. Closed source (such as company projects) are very often self-hosted with something like `https://gitlab.com`. We will now use utwente's gitlab server to host our source code because you already have an account there and you can share with other students.

Go to `https://gitlab.utwente.nl` and sign in with your student number (e.g. s2037335) and password.

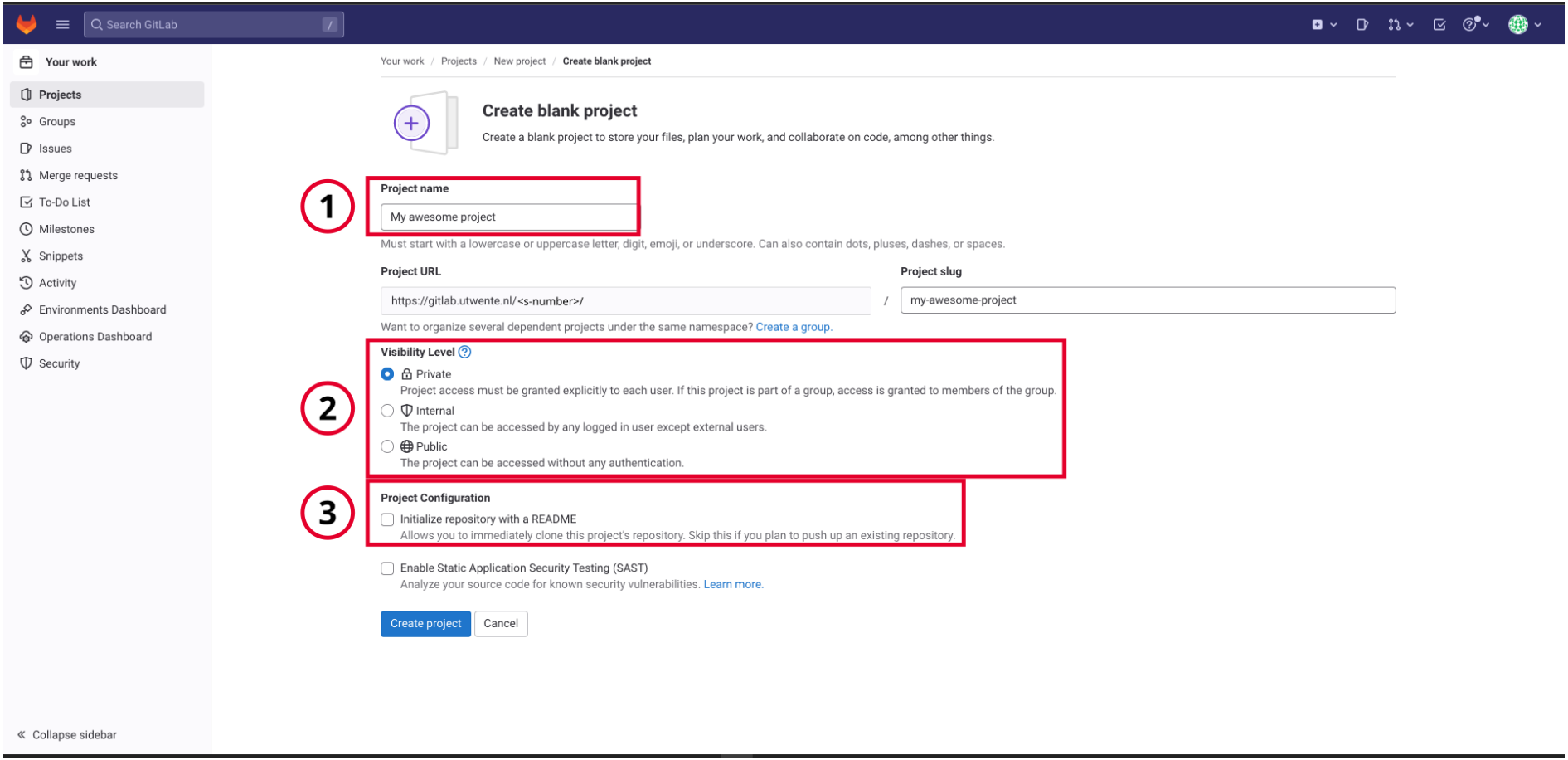
Create repository on Gitlab



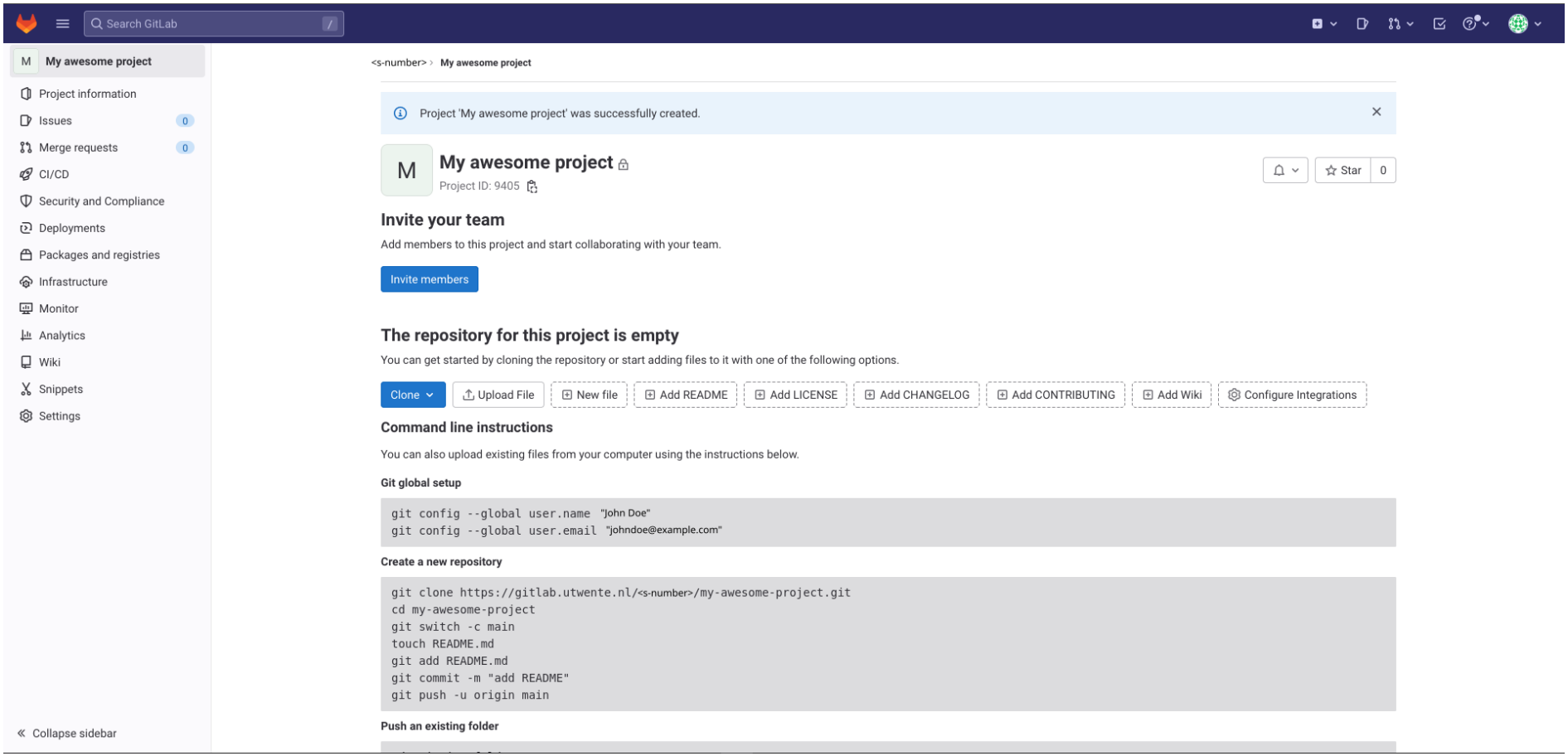
Create repository on Gitlab



Create repository on Gitlab



Create repository on Gitlab



Set up remote tracking locally

On our local repository we will add a reference to the remote repository we just created:

```
$ git remote add origin https://gitlab.utwente.nl/<s-number
>/<project-name>.git
```

Push to remote

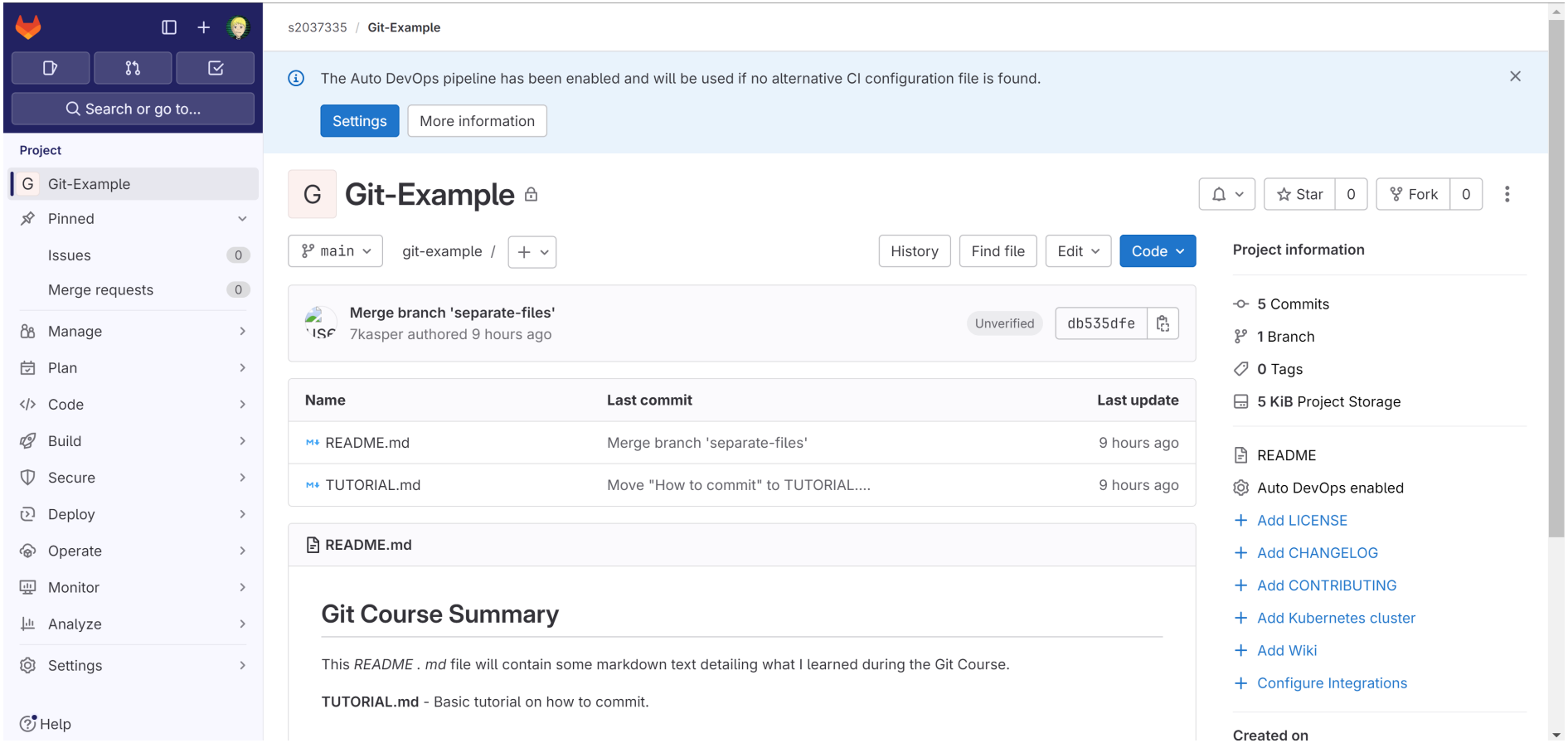
If we look at the Gitlab page we will not see any of the files we made. That is because we have not yet pushed our commits to the remote. Let's synchronise these repositories:

```
$ git push -u origin main
```

Note: you may need to provide login information. On a public repository everyone can download but only members can push changes to it.

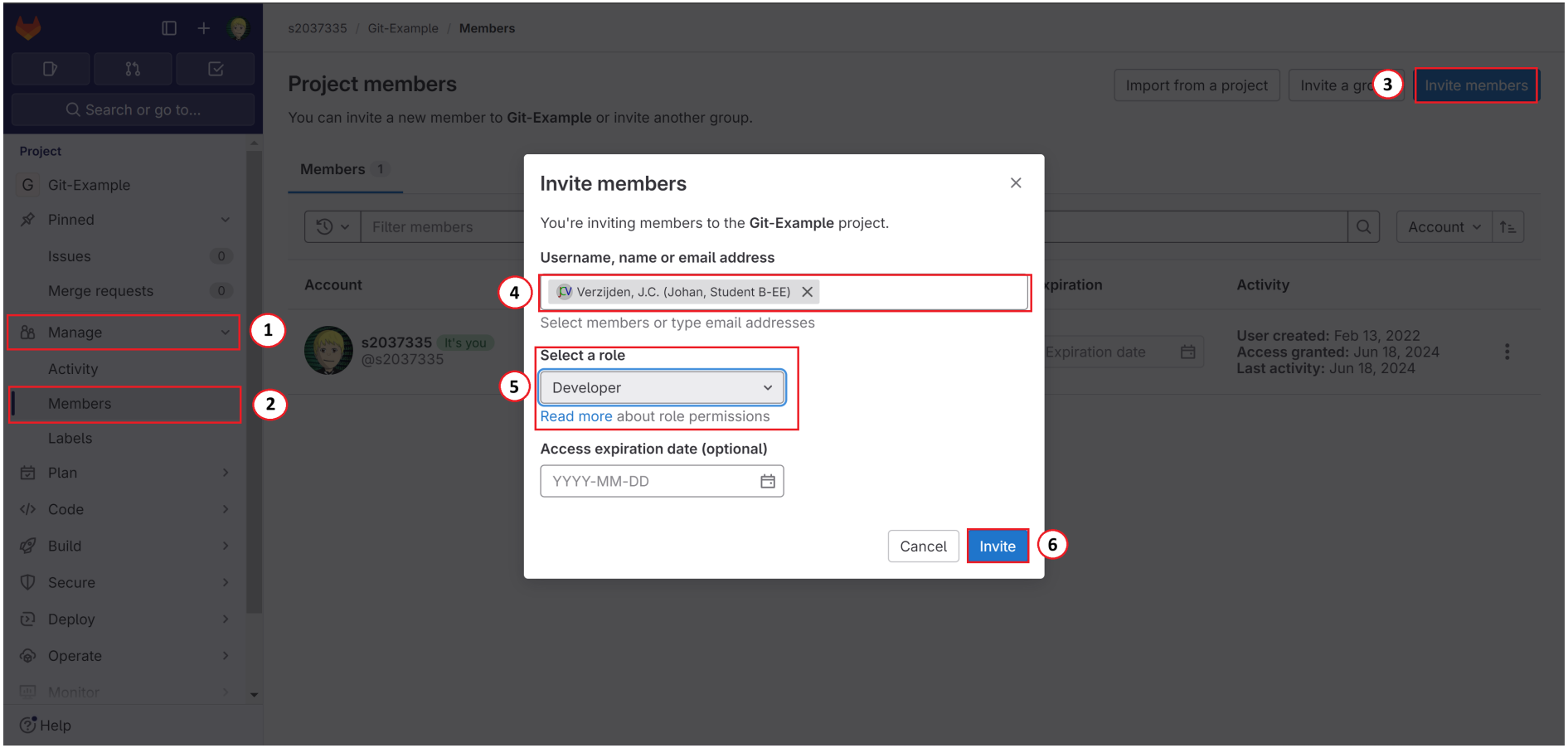
Look at repository on Github/Gitlab

You will now see your files and README on the remote repository!



Inviting other developers

As previously said; since the repository is public everyone on utwente can see and copy the code. To allow people on private repositories or to allow them to also push changes to the remote you can invite them as members:



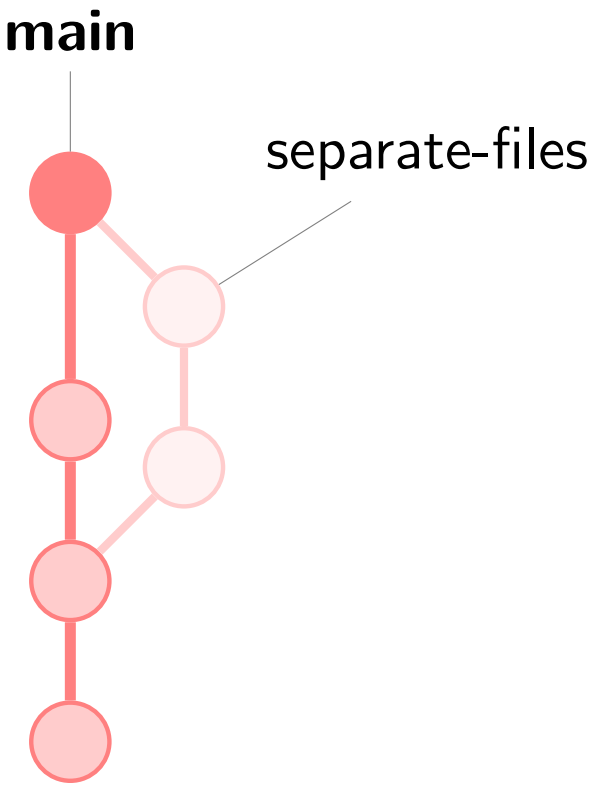
Cloning

The remote is where people base their own local repositories on. If we are invited to someone's repository we can clone this repository onto our local machine. We can then edit code, do commits and eventually push back to the remote.

Clone someone else's repo (after being added as member) or simulate working together by cloning your own repository in a different folder:

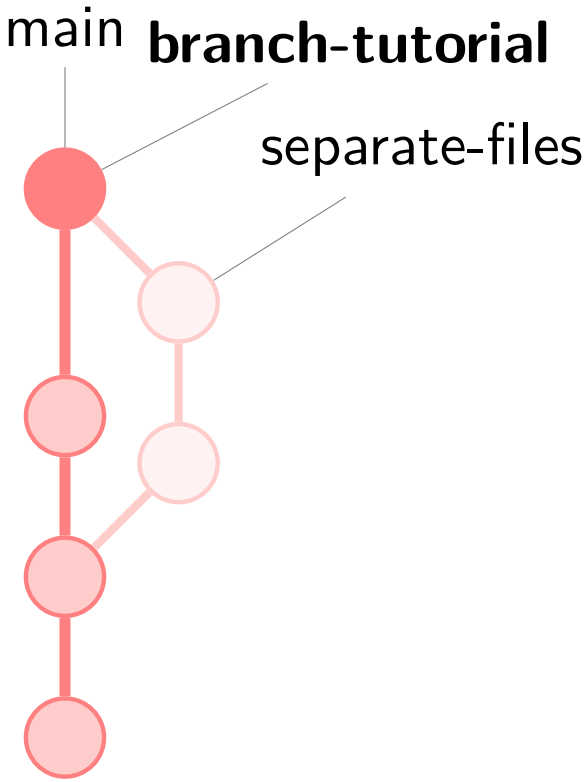
```
$ cd ../
$ mkdir clone
$ cd clone
$ git clone https://gitlab.utwente.nl/<s-number>/<project-name>
$ cd <project-name>
```

Comitting on the clone



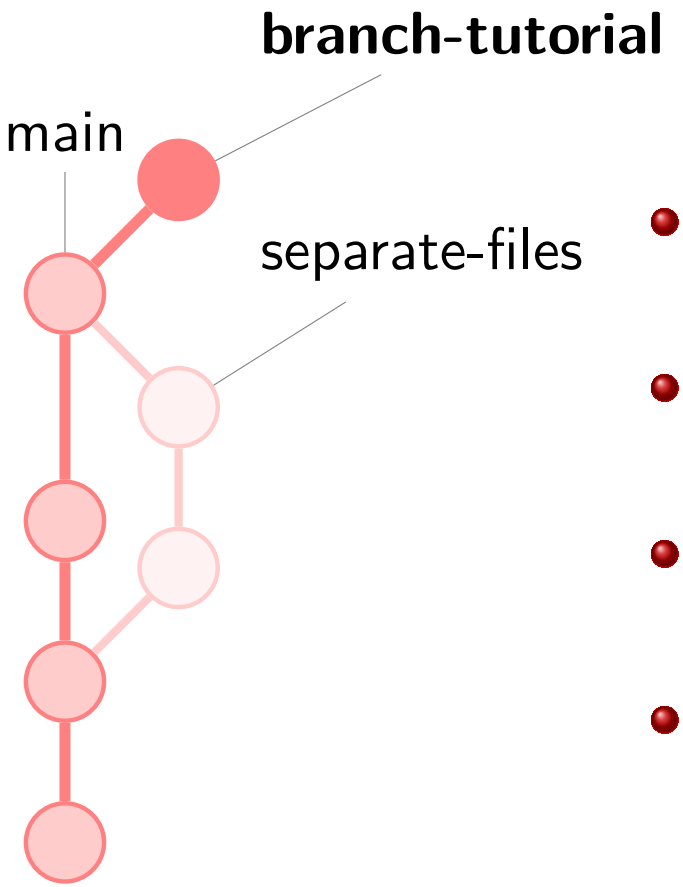
- We will do a commit on the new branch *branch-tutorial* branch on the clone:
  `$ git switch -c branch-tutorial`

Comitting on the clone



- We will do a commit on the new branch *branch-tutorial* branch on the clone:
  `$ git switch -c branch-tutorial`
- Now add a summary to the *TUTORIAL.md* file of how to make a branch and switch to it.
- Once again commit your changes: `$ git commit -am 'Add branching tutorial'`

Comitting on the clone



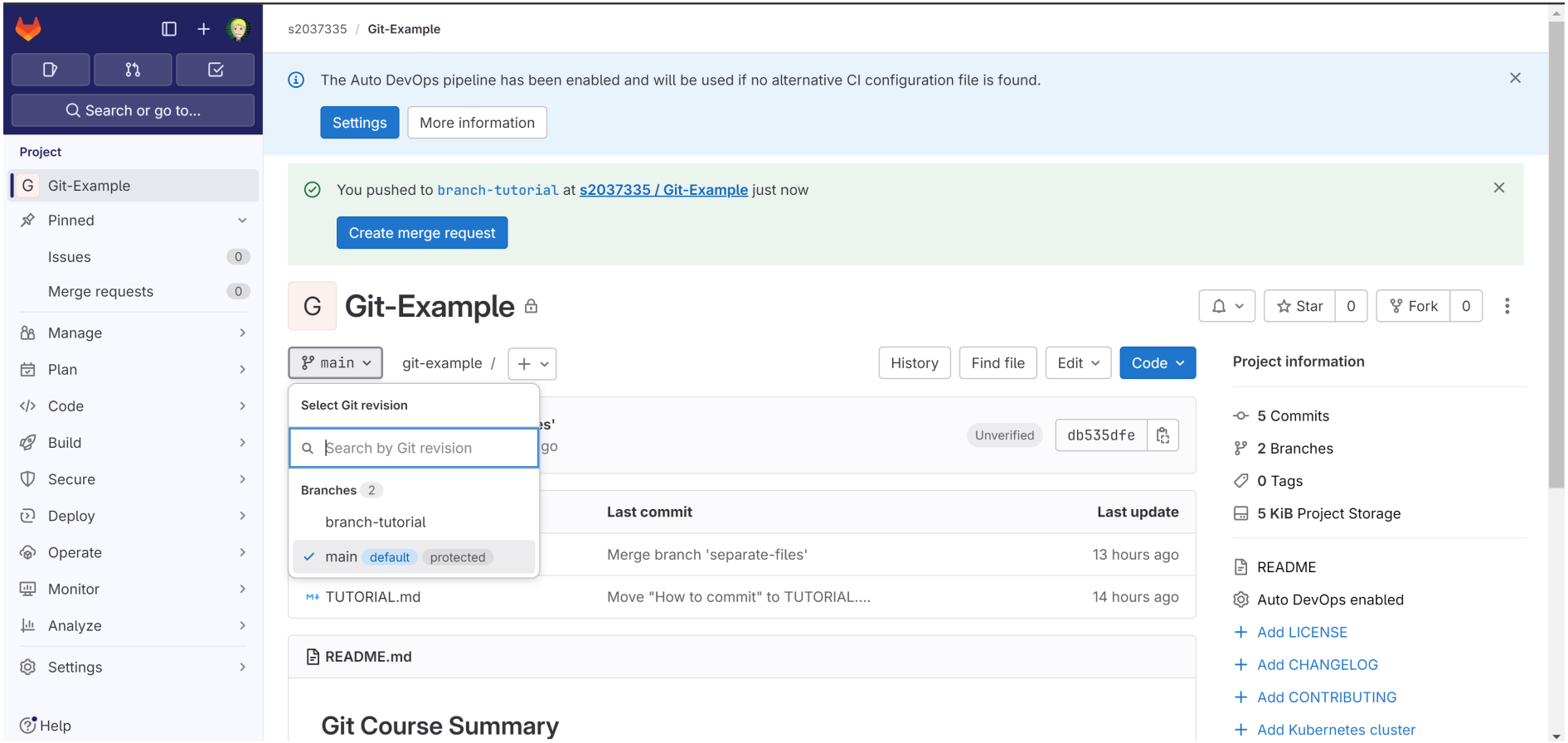
- We will do a commit on the new branch *branch-tutorial* branch on the clone:
  `$ git switch -c branch-tutorial`
- Now add a summary to the *TUTORIAL.md* file of how to make a branch and switch to it.
- Once again commit your changes: `$ git commit -am 'Add branching tutorial'`
- We now see a new commit on *branch-tutorial* in the clone repository.
- *Note: Although the separate-files history is embedded in the main after the merge commit the branch itself is not known as it was never pushed to or pulled from the remote.*

Pushing from the clone

We commit our new branch from the clone to the remote repository:

```
$ git push origin branch-tutorial
```

After this we can see that on the website we can switch between the pushed branches:



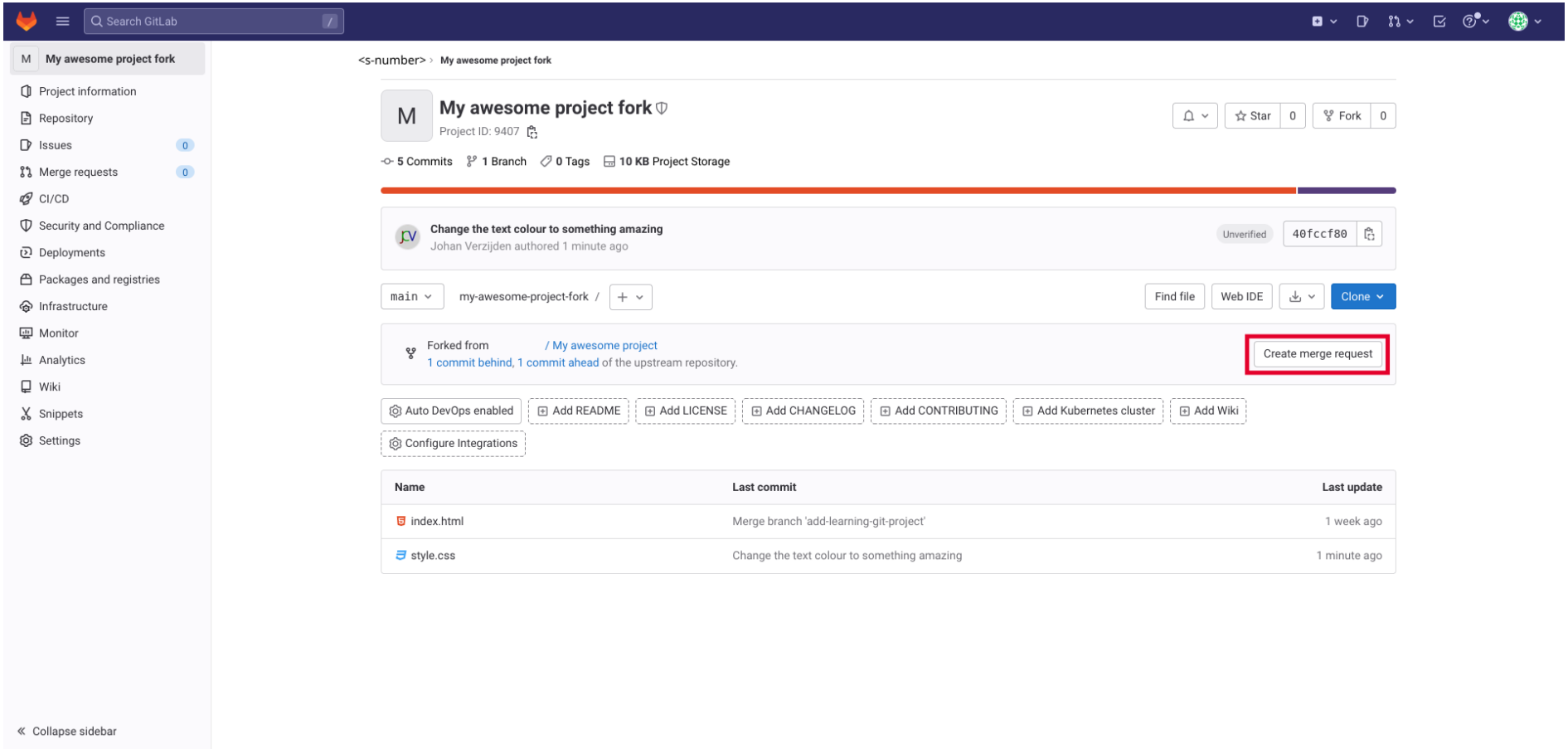
Merge request

Instead of merging on the command line we can create a merge request (often also called pull request). This is very useful for working together. Before actually bringing the *branch-tutorial* branch changes into the *main* version of our summary project we first create the request for this. This allows us to write some text explaining, discussing this with comments and also have other people review the changes you suggest.

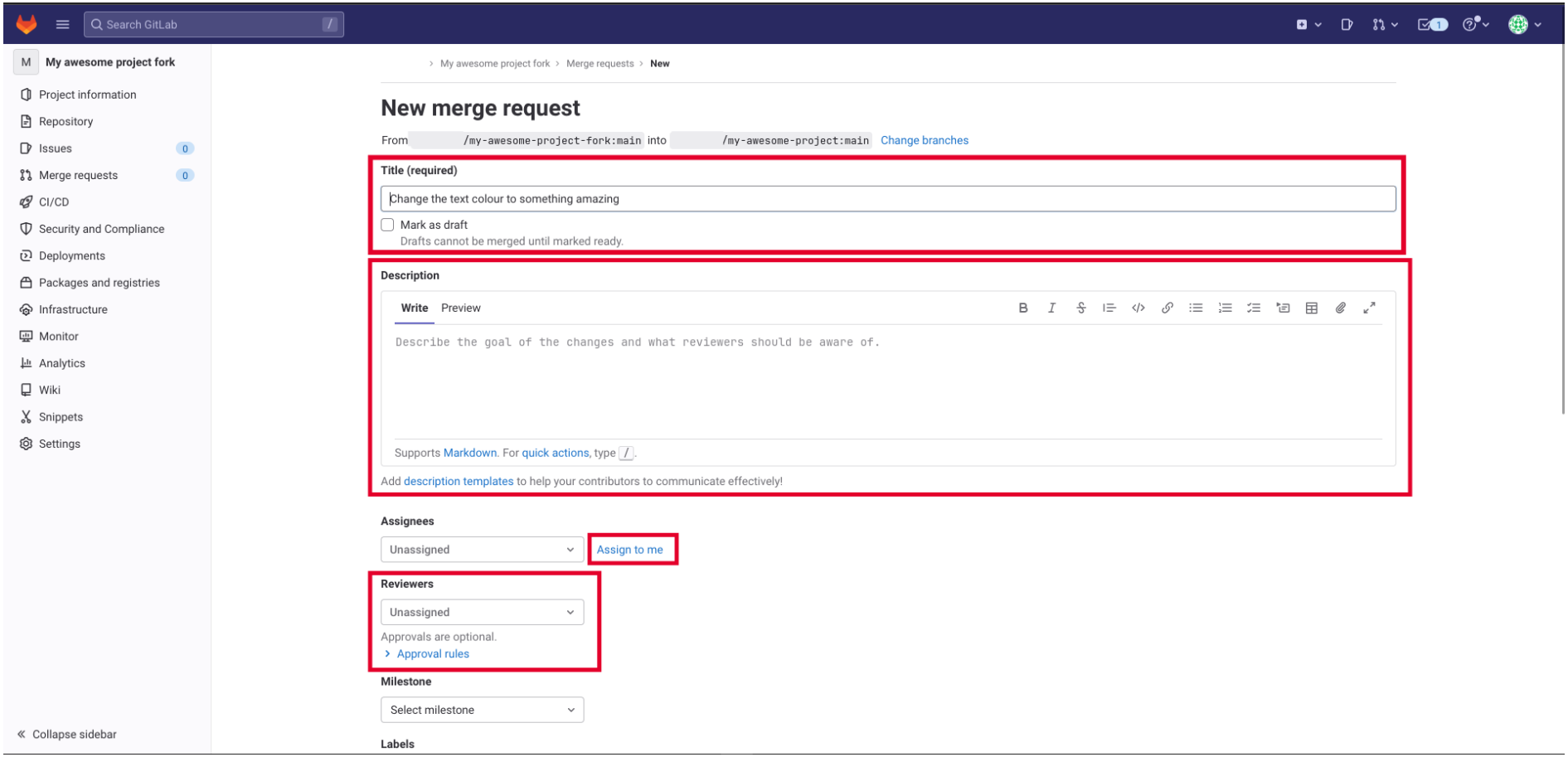
Note: On open source projects you are often not a member so you cannot push to these repositories straight away. You can however create a so-called fork of the project. This is your own copy where you have all the access rights. After pushing to your fork you can create a pull request to push the changes 'upstream' to the actual repository.

Let's try it!

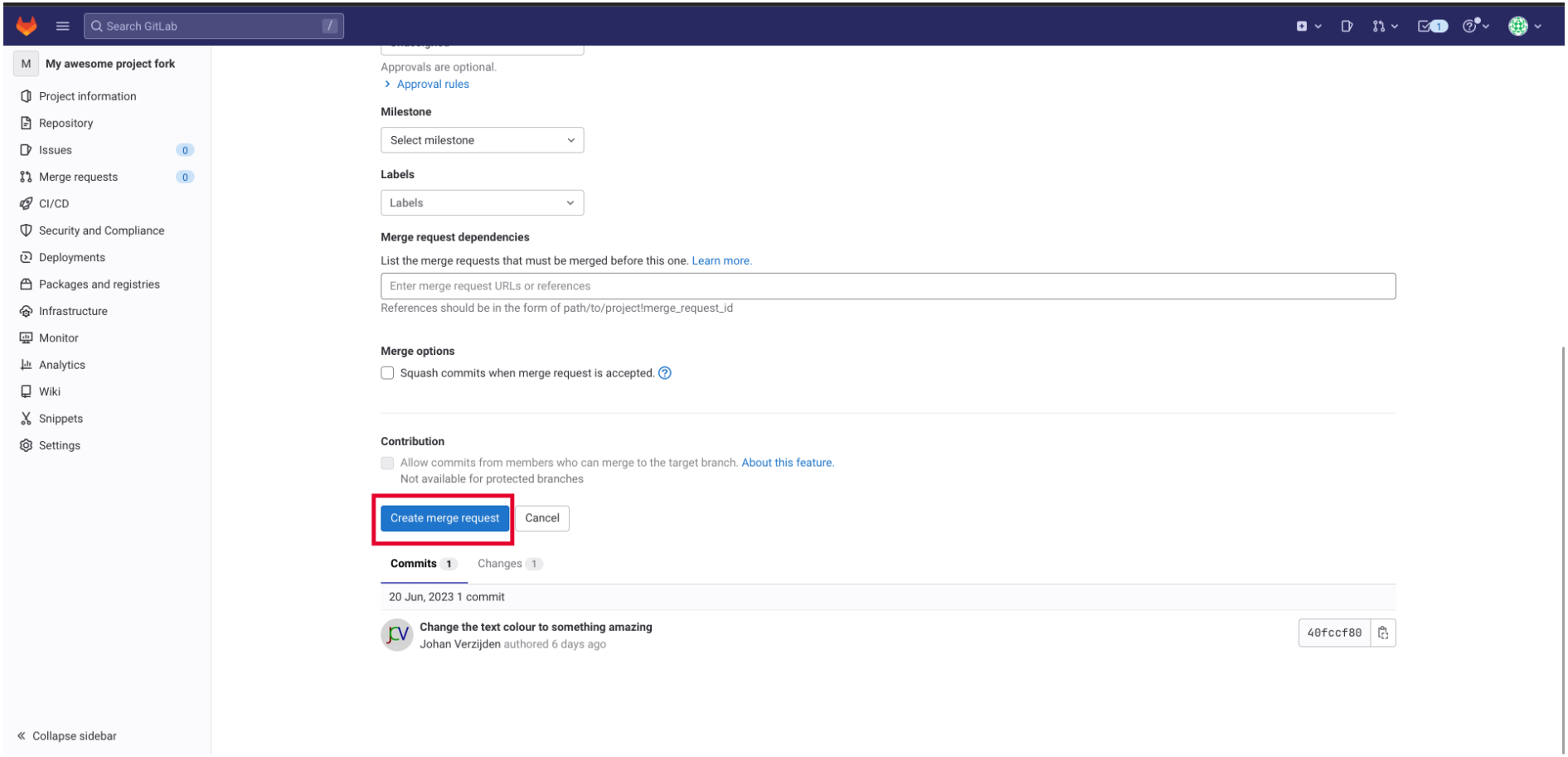
Open merge request



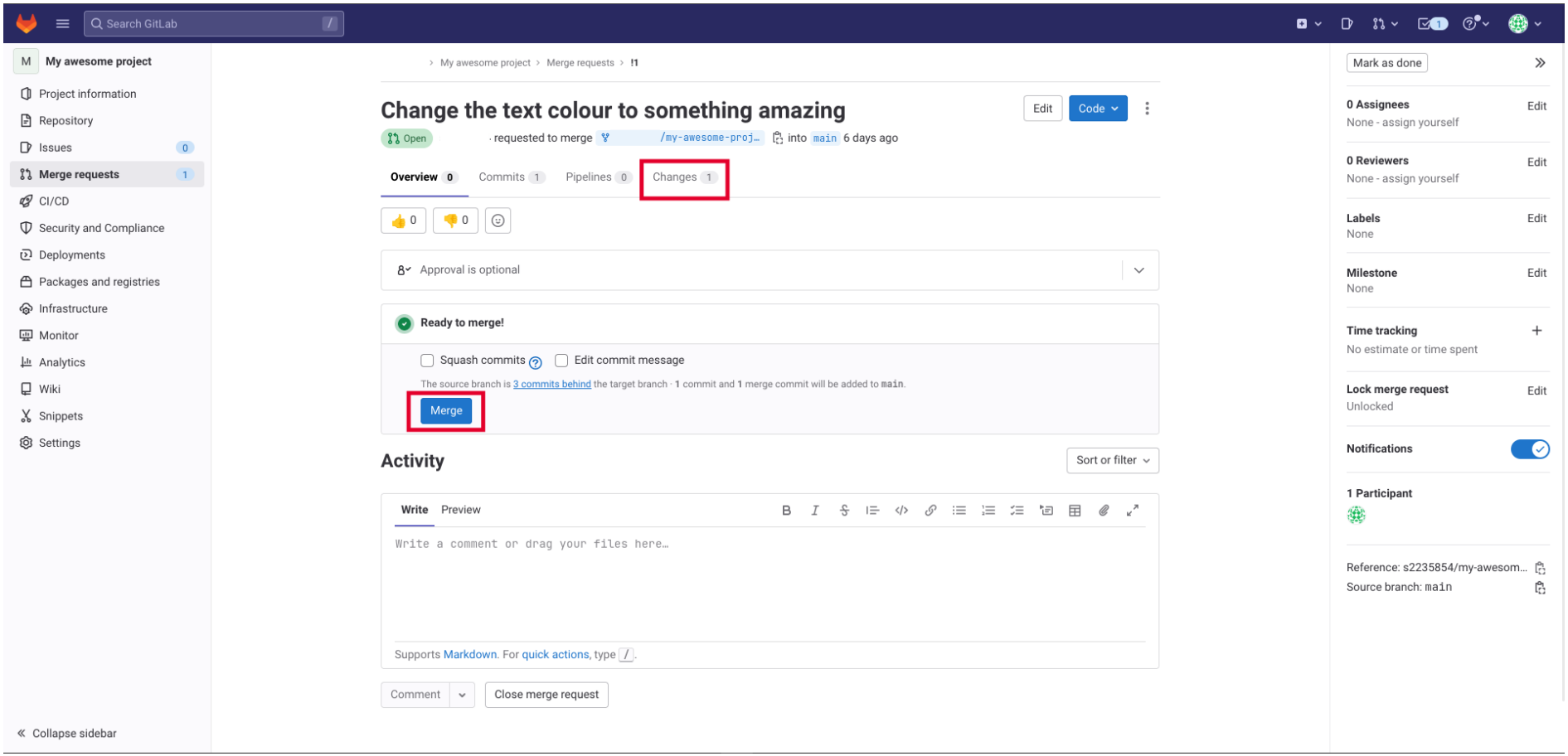
Open merge request



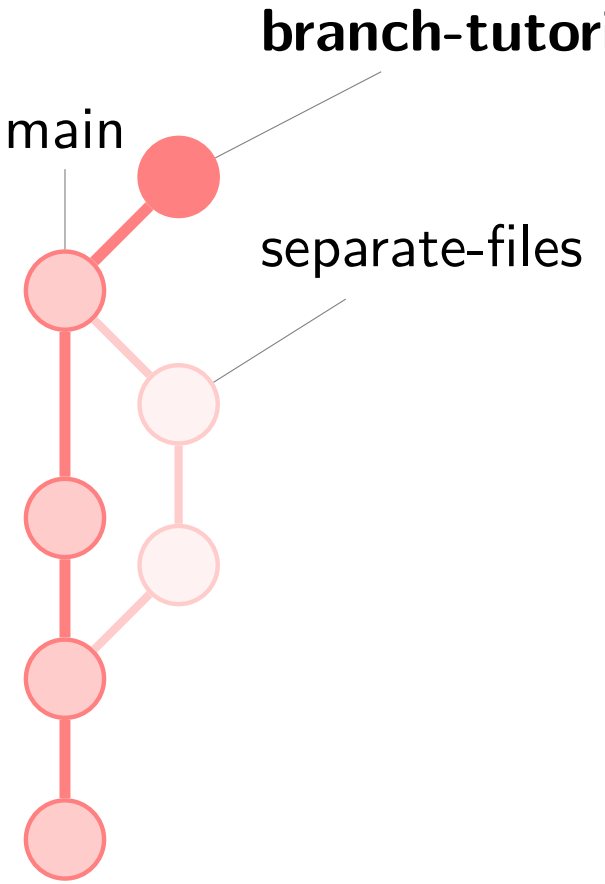
Open merge request



Review and accept merge request

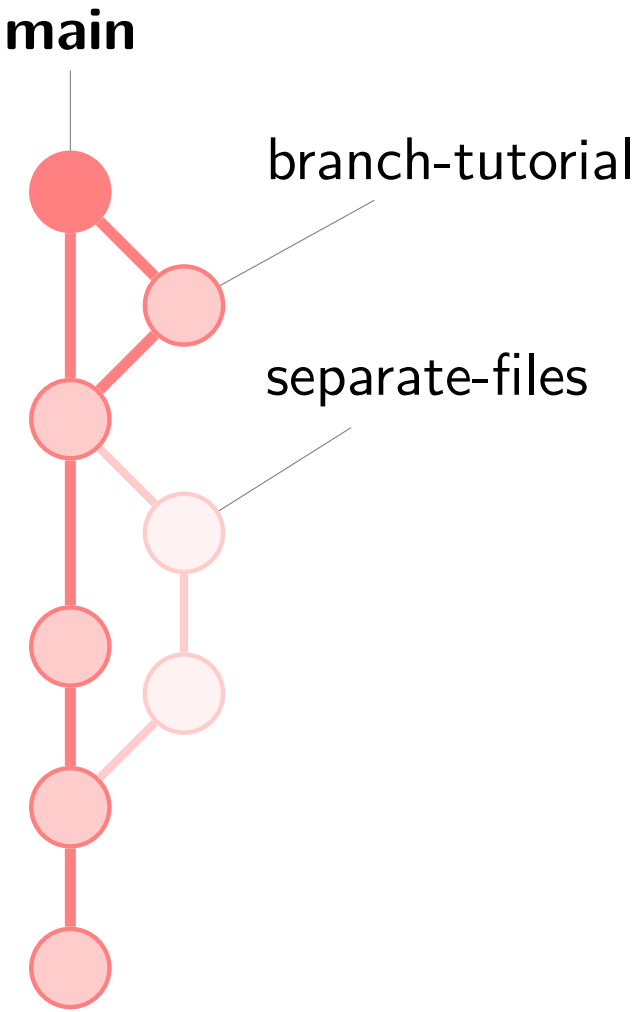


Performing the merge



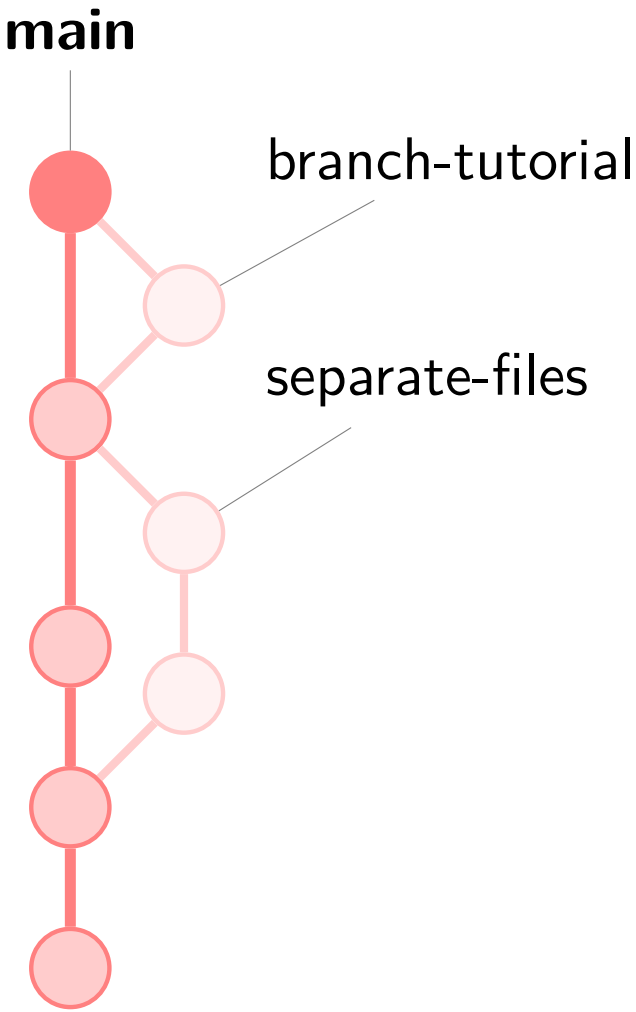
- On the website create a merge request. Merge *branch-tutorial* into *main*.
- Review and merge the request.

Performing the merge



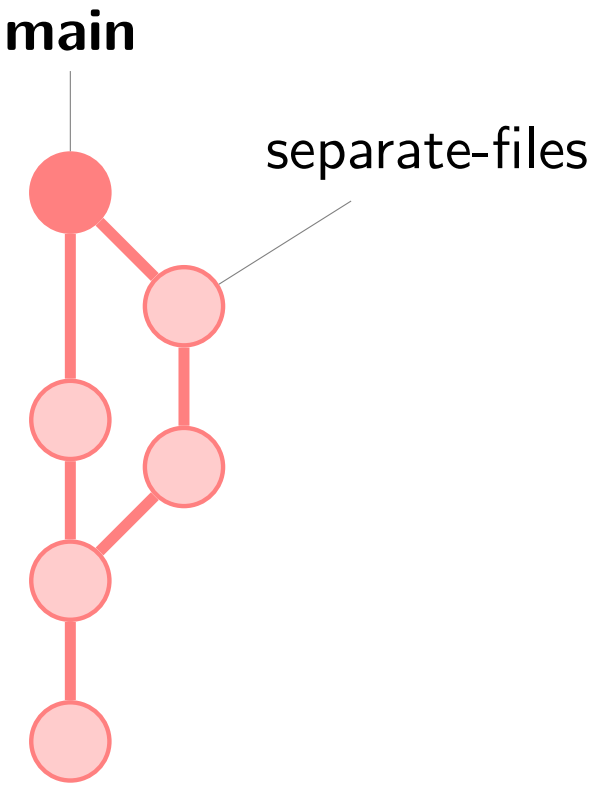
- On the website create a merge request. Merge *branch-tutorial* into *main*.
- Review and merge the request.
- Now again a merge commit is created on the *main* branch.
- If we know we don't need the *branch-tutorial* branch anymore we can click 'Delete source branch' on the Merge Request.

Performing the merge



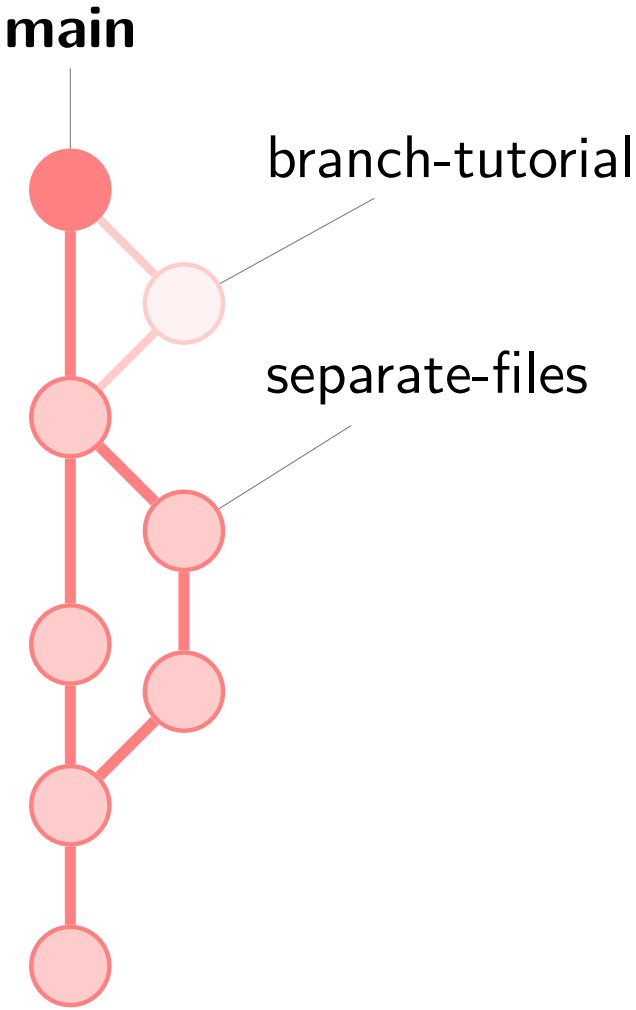
- On the website create a merge request. Merge *branch-tutorial* into *main*.
- Review and merge the request.
- Now again a merge commit is created on the *main* branch.
- If we know we don't need the *branch-tutorial* branch anymore we can click 'Delete source branch' on the Merge Request.
- We see the remote no longer holds the source branch but all changes and history are embedded from the merge.

Pulling from the origin



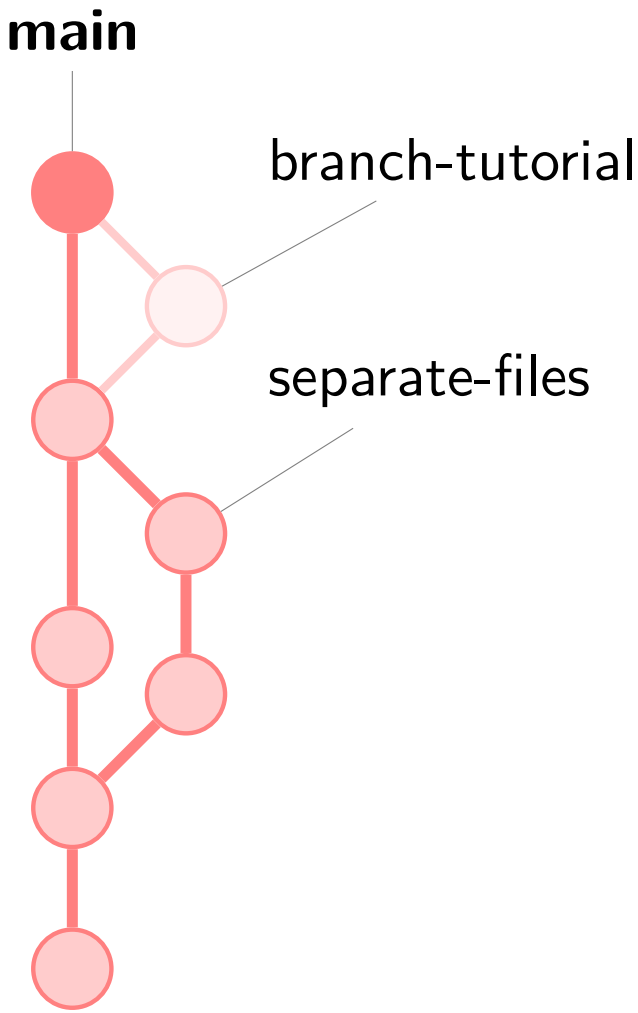
- Let's switch back to our first local repository:
  `$ cd ../../<project-name>`
- Now of course we don't know about the changes of the remote yet. Let's reel them in:
  `$ git pull origin main`

Pulling from the origin



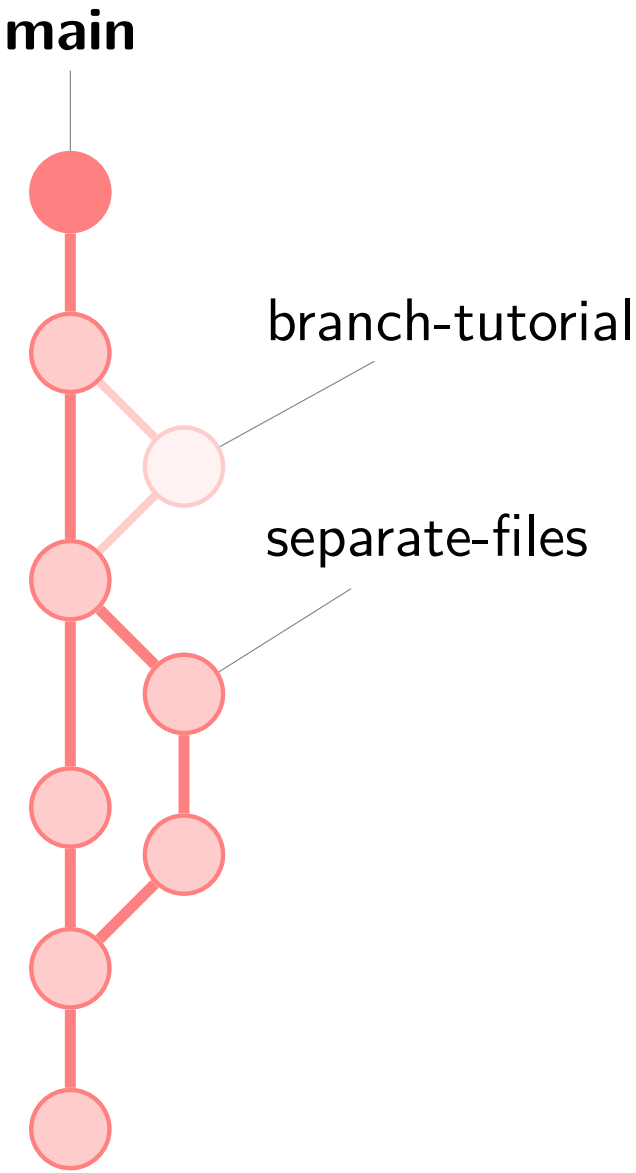
- Let's switch back to our first local repository:
  `$ cd ../../<project-name>`
- Now of course we don't know about the changes of the remote yet. Let's reel them in:
  `$ git pull origin main`
- Yes! Now our changes are also in the other local branch!
- *Note: In this repository we still of course have the separate-files branch as we never deleted it locally. We do not have the branch-tutorial branch as we have never pulled it from the remote.*

Reverting a commit



- Imagine that the last change screwed up our program and we want to delete it. Git helps a lot by recording all changes so we can just revert the change.
- First find out the hash of the commit that introduced the mistake. You can do this on the website or perhaps with:
  `$ git log`
- In my case it seemed the 'Add branching tutorial' was wrong. Its hash starts with 'd7f775'. We can revert it using:
  `$ git revert d7f775`

Reverting a commit



- Imagine that the last change screwed up our program and we want to delete it. Git helps a lot by recording all changes so we can just revert the change.
- First find out the hash of the commit that introduced the mistake. You can do this on the website or perhaps with:
  `$ git log`
- In my case it seemed the 'Add branching tutorial' was wrong. Its hash starts with 'd7f775'. We can revert it using:
  `$ git revert d7f775`
- This creates a new commit on *main* that does the opposite changes of the commit specified. It essentially negates it.

Never forget

Reverting creates a new commit. History never gets deleted this way and we can even revert the revert. Of course to update to our other developers we have to push to the origin and then pull on the other machines again.

Note: While it is not advised to muck about in the history of git commits, you can for instance use `git reset`. A use case might be that you accidentally commited a password to a shared repository and reverting the commit does not remove this from history. As these actions are more uncommon we will discuss them only in the intermediate course.

Done!

This concludes the practical for the basic git course. You have now used all important and most essential git commands and concepts, congratulations!

There is a lot more that Git can do. It can help you find bugs by doing a binary search on your code, there are commands to easily help you find changes in history (so you know who to blame). You can work together with multiple upstream branches, secure your code with signed GPG keys, etc. If you want to learn more about this search the web or come to our intermediate course! :-)