Marwen Dallel, Mootez Saad and Mahdi Turki

GitHub Guide for the *Introduction to Software Engineering* course

September 29, 2020

MedTech - Mediterranean Institute of Technology

Contents

Chapter 1
General Overview

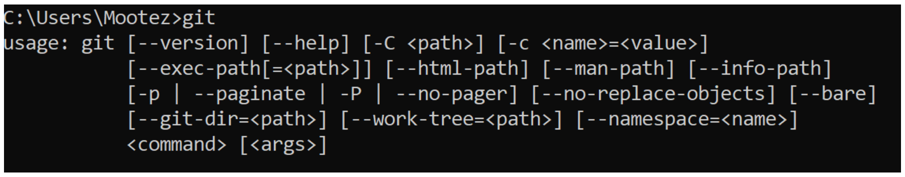
Abstract This chapter is a general overview of the available features on GitHub. We will go through the process of repository creation and familiarize ourselves with the user interface as well as understand the different concepts involved. Among others, these include branches, tags, issues and commit history (also know as commit log).

1.1 Requirements

Before starting this chapter, make sure that you have registered a GitHub account and that you have Git installed on your local machine. You can create an account by visiting this URL Join GitHub and download Git from here Git Downloads.

1.2 Git Setup

Assuming you meet all the requirements to proceed, the next thing you need to do is open your command-line interpreter (`cmd` for Windows or `Terminal` for Unix based systems) and type `git`, the following should be displayed:



```
C:\Users\Mootez>git
usage: git [--version] [--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>]
           <command> [<args>]
```

Fig. 1.1 Using the `git` command to verify if `git` is properly installed.
NB. This is the `cmd` interface used in Windows systems, yours might be different if you have another operating system.

In order for `git` to function properly, you will need to provide a username and an email address. Although you can enter any credentials, it is recommended that you use the same information you provided when creating your GitHub account as this will allow GitHub to link each commit to your account. Your username and email address will be used by `git commit` and embedded in the commits that you will create.
We will define what a commit is and go in depth in later sections.

Git Command

```
git config --global user.name "<USERNAME>"
git config --global user.email <EMAIL>
```

To check if the configuration has been successfully updated, type:

Git Command

```
git config --list --show-origin
```

Keep pressing Enter until the following lines are displayed (note that <USERNAME> and <EMAIL> will vary depending on the information that you have entered).

Git Output

```
...
file:[PATH]/gitconfig filter.lfs.required=true
file:[PATH]/gitconfig credential.helper=manager
file:[PATH]/.gitconfig user.email=<EMAIL>
file:[PATH]/.gitconfig user.name=<USERNAME>
```

Now, you need to generate an SSH key to connect to GitHub. By doing so, you avoid having to enter your credentials at each visit.
First, open `Git Bash` (if you are on Linux or Mac you can use `Terminal` directly) and type:

Command

```
ssh-keygen -t rsa -b 4096 -C "<EMAIL>"
```

This command will generate an SSH key using the email as a label. You will be prompted to enter the location of the file where the SSH key will be saved. If you press Enter without specifying a path, the key will be saved at the default location.

Command

```
> Enter a file in which to save the key
(/c/Users/you/.ssh/id_rsa):[Press enter]
```

Then, you will be asked to enter a secure passphrase (can be left empty)

Command

```
> Enter passphrase (empty for no passphrase): [Type a passphrase]
> Enter same passphrase again: [Type passphrase again]
```

Finally, we need to add the key to `ssh-agent`

Command

The first command helps you determine whether `ssh-agent` is running or not and the second adds the generated key.

```
$ eval $(ssh-agent -s)
Agent pid 1169
$ ssh-add ~/.ssh/id_rsa
```

The last step is to add the SSH key to your GitHub account. Copy the key to your clipboard as follows:

Command

```
$ clip < ~/.ssh/id_rsa.pub
```

If the command `clip` does not work, you can manually open the file and copy the SSH key.

All you have to do now is login to your `GitHub` account, go to Profile Picture > Settings > SSH and GPG keys and add a new SSH key.

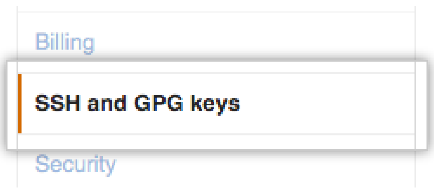
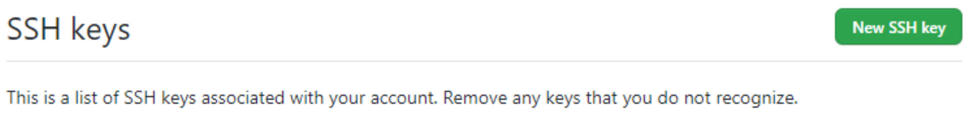


Fig. 1.2 You can manage multiple SSH and GPG keys, i.e. you can setup an SSH key for every machine you have.

After clicking on SSH and GPG Keys, click on New SSH Key to paste your generated key.



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

Fig. 1.3 Here you will find the list of your keys

Add a title that describes the origin of your key (e.g. Laptop SSH Key) so you can easily manage your list of keys.

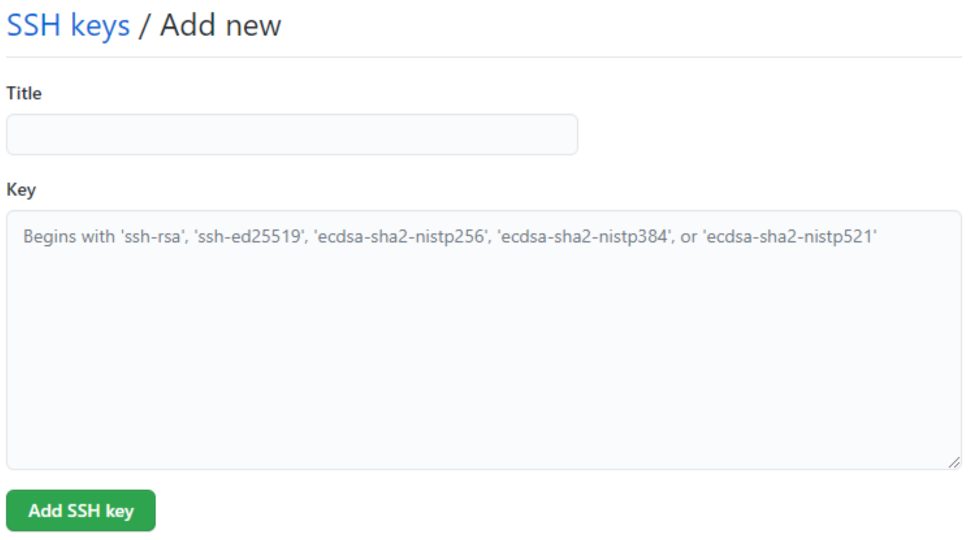

SSH keys / Add new
Title
Key
Begins with 'ssh-rsa', 'ssh-ed25519', 'ecdsa-sha2-nistp256', 'ecdsa-sha2-nistp384', or 'ecdsa-sha2-nistp521'
Add SSH key

Fig. 1.4 Adding an SSH key with a title

That is it, you have successfully configured your GitHub account on your local machine. In the following sections we will see how to create and manage repositories.

1.3 Creating a new repository

After you are done with the configuration, we will create a new repository. Go to the plus icon at the top-right, and select **New repository**

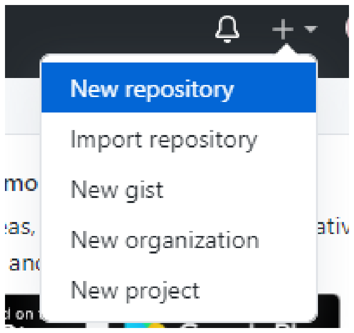
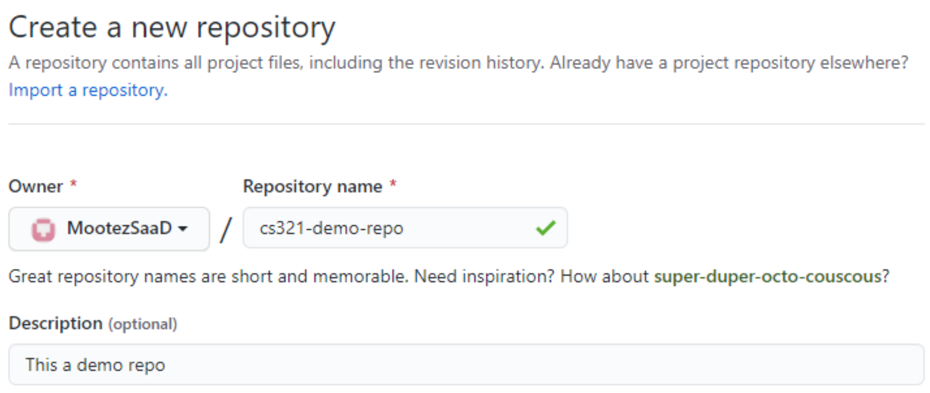


Fig. 1.5 You can create organizations, projects and gists, but for now we only care about creating a new repository



Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? Import a repository.

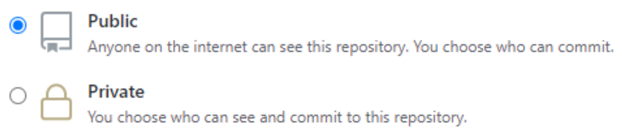
Owner * MootezSaaD / Repository name * cs321-demo-repo

Great repository names are short and memorable. Need inspiration? How about super-duper-octo-couscous?

Description (optional)

This a demo repo

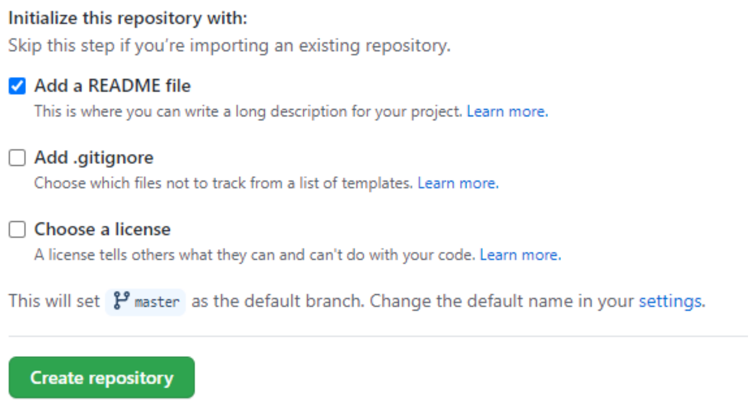
Fig. 1.6 Choose a title for repository. You can an optional description, though it is better to add one



[✓ Public] Anyone on the internet can see this repository. You choose who can commit.

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

Fig. 1.7 You can manage your repository's visiblity. Public means that any can see it, private indicates that you can choose who can view your repository



Initialize this repository with:

Skip this step if you're importing an existing repository.

[✓] Add a README file
This is where you can write a long description for your project. Learn more.

[ ] Add .gitignore
Choose which files not to track from a list of templates. Learn more.

[ ] Choose a license
A license tells others what they can and can't do with your code. Learn more.

This will set `master` as the default branch. Change the default name in your settings.

Fig. 1.8 Readme files usually include comprehensive information about the project you are hosting under your repository, such as installation guides, build and usage instructions, as well as guidelines on how one can contribute. You can always add a `README` file later. Another nice feature is the ability to choose and include a `gitignore` file upon creation. You can choose from a collection of the useful `.gitignore` templates taken from https://github.com/github/gitignore. Ultimately, the code that you will host on GitHub will be able to be forked and reused by third parties. If you are freshly starting a new repository, you can choose a license to include upon creation. Again, this is optional and you can always manually add a license file later

Ones you have completed this step, you will be redirected to your repository main page. Note that if you have chosen to add a license and a `README` file, they will be automatically added.

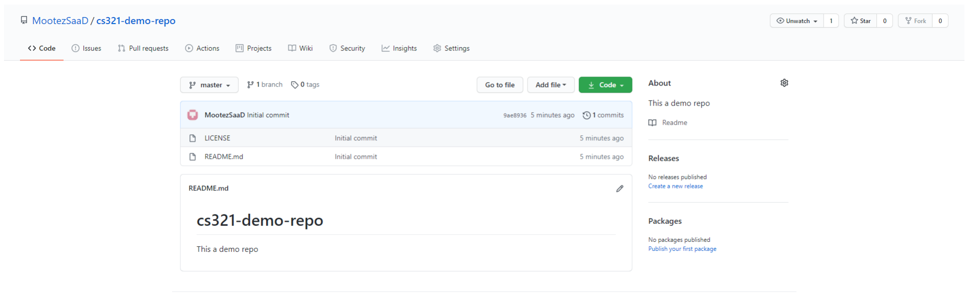
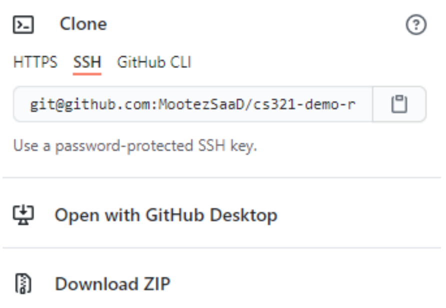


Fig. 1.9 Your repository main page

1.4 Cloning the repository and creating new commits

Now, we will clone the repository on our local machine, click on the **Code** green button and you will see the different clone methods, we will do so via SSH.



```
git@github.com:MootezSaaD/cs321-demo-r
```

Use a password-protected SSH key.

Fig. 1.10 Cloning the repository using SSH, you can use HTTPS (you are required to enter your password during each commit), or via GitHub CLI Read more

After copying the link, type the following command

Git Command

```
git clone <URL>
```

In this case <URL> is `git@github.com:MootezSaaD/cs321-demo-repo.git`, more generally repository SSH URLs follow this format:

Command

```
git@github.com:<USERNAME>/<REPO-NAME>.git
```

NB. The username is case sensitive.
If everything goes fine, you should get the following ouput.

Git Output

```
$ git clone git@github.com:MootezSaaD/cs321-demo-repo.git
Cloning into 'cs321-demo-repo'...
...
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (3/3), done.
Receiving objects: 100% (4/4), done.
remote: Total 4 (delta 0), reused 0 (delta 0), pack-reused 0
```

A folder with your repository's title should be created. If you navigate to the folder, using Git Bash, you will get a result similar to this:

Command

```
$ cd cs321-demo-repo
Mootez@DESKTOP ~/cs321-demo-repo (master)
```

`master` is the branch that we are currently working on, it is the main branch, and oftentimes it contains stable and bug-free code. We will create a new JavaScript file that contains some trivial code. In our case, we will be sorting arrays.

Command

```
  $ touch sort_array.js
```

We will edit it using `vim` since will only write a couple line of code (obviously you can use any code editor of your liking).

Command

```
  $ vim sort_array.js
  const array = [5, -2, 84, 0, 21];
  console.log(array);
  array.sort();
  console.log(array);
  ...
  ~
  sort_array.js[+] [unix] (15:13 21/09/20)
  -- INSERTION --
```

We can now track the changed files using

Command

```
$ git status
On branch master
Your branch is up to date with 'origin/master'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        sort_array.js
nothing added to commit but untracked files present
(use "git add" to track)
```

Here we have an untracked file, a file that `git` "does not know about", meaning that it is newly created and not present in the last snapshot. We will now add the file and create a commit.

Git Command

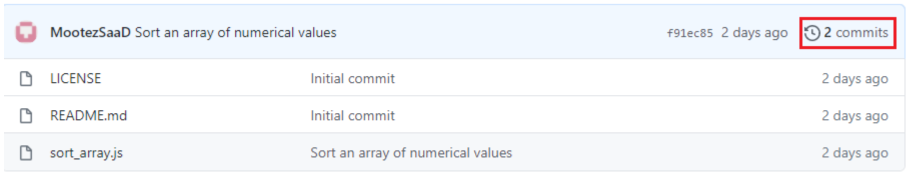
```
# We can add all edited files at once
$ git add .
# Or we can add each file at time, the reason behind this is
# that you create multiple commits, it is a good practice to
# split your modifications into small chunks on many commits, rather
# than creating one commit containing all of these modifications.
$ git add sort_array.js
$ git commit -m "Sort an array of numerical values"
[master f91ec85] Sort an array of numerical values
 1 file changed, 4 insertions(+)
 create mode 100644 sort_array.js
```

You can find here some conventions for writing commit messages. Finally, we push our commit.

Git Command

```
$ git push
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 8 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 391 bytes | 391.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To github.com:MootezSaaD/cs321-demo-repo.git
   9ae8936..f91ec85  master -> master
```

If we go back to the repository's page, we will see that the file has been added and we have a total of two commits (for the first one was created by GitHub since we predefined a `README` and a license file) To see the log of commit, `git log` and



MootezSaaD Sort an array of numerical values	f91ec85 2 days ago	2 commits
LICENSE	Initial commit	2 days ago
README.md	Initial commit	2 days ago
sort_array.js	Sort an array of numerical values	2 days ago

Fig. 1.11 Our "first" commit to the repository

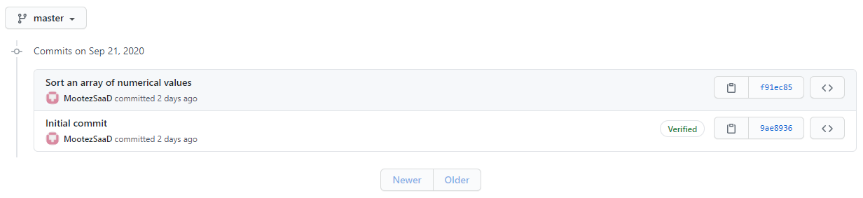
it will display the list of all commits

Git Command

```
$ git log
commit f91ec859e929d9770e0b870f5c0605b44f813605
(HEAD -> master, origin/master, origin/HEAD)
Author: Mootez <mootez.saad@medtech.tn>
Date:   Mon Sep 21 16:00:56 2020 +0100
    Sort an array of numerical values

commit 9ae8936755c3cbedf89e32ae744bae87a887affe
Author: Mootez Saad <34676841+MootezSaaD@users.noreply.github.com>
Date:   Mon Sep 21 12:52:43 2020 +0200
    Initial commit
```

Though GitHub offers a nicer graphical interface,



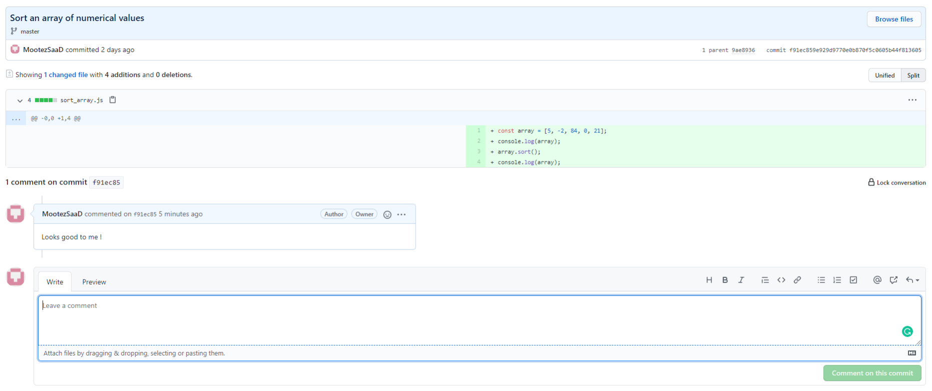
Commits on Sep 21, 2020

Sort an array of numerical values
MootezSaaD committed 2 days ago f91ec85

Initial commit
MootezSaaD committed 2 days ago Verified 9ae8936

Fig. 1.12 You can navigate to this page by clicking on the number of commits (highlighted in Figure 1.11). This interface is more intuitive and easier to use than the command line.

You can check the content of the commit if you click on it, at the top is where the commit's title written in bold with the author's name below it. On the left-hand side is the state of the file before this commit (in this case it did not exist so it is empty) and on the right-hand side the new state of the file, lines highlighted in green show the additions that have been made (if a part of it was deleted that part would be highlighted in red). You can also leave a comment, this can be useful when collaborating within a team.



Sort an array of numerical values
master
MootezSaaD committed 2 days ago 1 parent 9ae8936 commit f91ec859e929d9770e0b870f5c0605b44f813605

Showing 1 changed file with 4 additions and 0 deletions.

4 sort_array.js

```
@@ -0,0 +1,4 @@
+ const array = [5, -2, 84, 0, 21];
+ console.log(array);
+ array.sort();
+ console.log(array);
```

1 comment on commit f91ec85

MootezSaaD commented on f91ec85 5 minutes ago
Looks good to me !

Fig. 1.13 Commit page

1.5 Dealing with branches

In a more concrete setting, we might find it better to create multiple branches instead of dealing with the `master` branch only, for easier and modular development. For instance, in a group of four, two members can work on Feature1 and the two others can work on a Feature2 with each feature having its own branch. This model is called

Git Feature Branch Workflow, it offers many benefits for continuous development and not disturbing the main code base. Some projects adopt this model with a slight modification, where each branch can also represent a patch or a fix for an issue/bug. It is also important to keep in mind that all commits, branches or any type of changes made are local by default. This means that unless the modifications are pushed to the remote branch, the one hosted on GitHub, the changes would only be available on the local machine. Having said that, let us create a new local branch called `new_branch`.

Git Command

```
$ git checkout -b new_branch
Switched to a new branch 'new_branch'
```

Let us create a JavaScript function that takes an array as input and returns an array composed of two elements, where the first element is an array containing all even values and the second contains all odd values.

Command

```
$ vim split_parity.js
function splitParity(x) {
 return [[...x.filter(e => e%2 == 0)],
        [...x.filter(e => e%2 == 1)]];
}
```

Same procedure for creating a commit, however there is a slight difference when pushing to the repository. Instead of simply typing `git push` we need to add the following:

Command

```
$ git push origin my_branch
```

If you want to change the local branch you are working on, you can do so by

Git Command

```
$ git checkout <BRANCH-NAME>
```

If you want to merge the content of `branch1` into `branch2`

Command

```
# Checkout to the destination branch
$ git checkout branch2
# Merge with source branch
$ git merge branch1
```

1.6 Forking and creating Pull Requests

A fork is a copy of a repository. Forking a repository would allow to perform changes on the code without affecting the original copy. They are usually used to make small changes (e.g. fixing a bug) or as base for a new project idea.
Suppose you are browsing the issues of a large open source project with thousands lines of code. One issue in particular piqued your interest and you want to contribute by fixing it. Since you are neither the owner of the repository nor part of the team managing the project, you do not have sufficient permissions to push your changes directly to the code base. In this case, you need first to fork the repository.
To do so, go the repository's main page and click on the **fork** button at the top-right. Wait for few seconds and your fork should be ready.

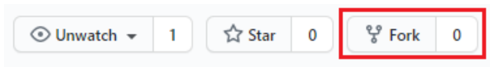


Fig. 1.14 Forking a repository

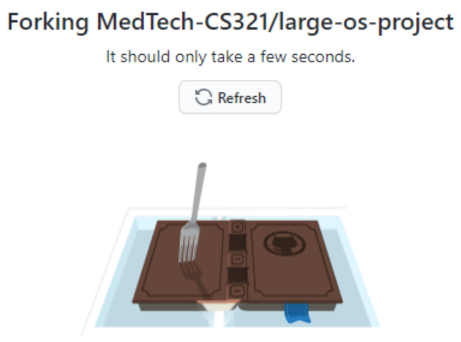

Forking MedTech-CS321/large-os-project
It should only take a few seconds.
Refresh

Fig. 1.15 Waiting for the repository to be forked

Now, if you want to modify the code base you need to follow the contribution guidelines imposed by the maintainers. For the purposes of this demonstration, we assume the following guidelines:

- In order to fix an issue, you need to create a new branch called `fix-for-<ISSUE-NUMBER>`
- All modifications must be first pushed to your forked repository on the aforementioned branch

Git Command

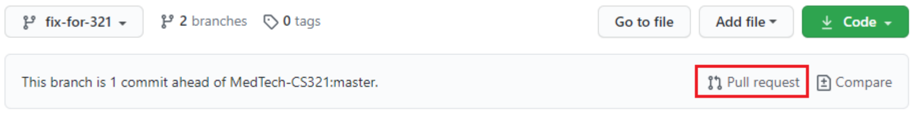
```
# Suppose the issue's number (id) is 321
$ git checkout -b fix-for-321
```

Let us say we have a function that takes two numbers "a" and "b" and returns a/b, however there is no check if "b" equals to zero, so we want to handle that by returning a message.

Git Command

```
$ git diff divide_number.js
diff --git a/divide_number.js b/divide_number.js
index 8732fdc..eaadd2e 100644
--- a/divide_number.js
+++ b/divide_number.js
@@ -1,3 +1,6 @@
 function divide(a,b) {
+       if (b == 0) {
+                return "Cannot divide by zero !";
+        }
        return a/b;
 }
```

After fixing the issue, we push the commit. If we go back to our repository, we can notice that the branch has been added. So the question becomes: How can we merge our modification with the original code base?
For that we need to introduce the concept of pull requests (also known as merge requests). “Pull requests let you tell others about changes you’ve pushed to a branch in a repository on GitHub. Once a pull request is opened, you can discuss and review the potential changes with collaborators and add follow-up commits before your changes are merged into the base branch.”
NB. From now on, we will refer to pull requests by their acronym "PR".



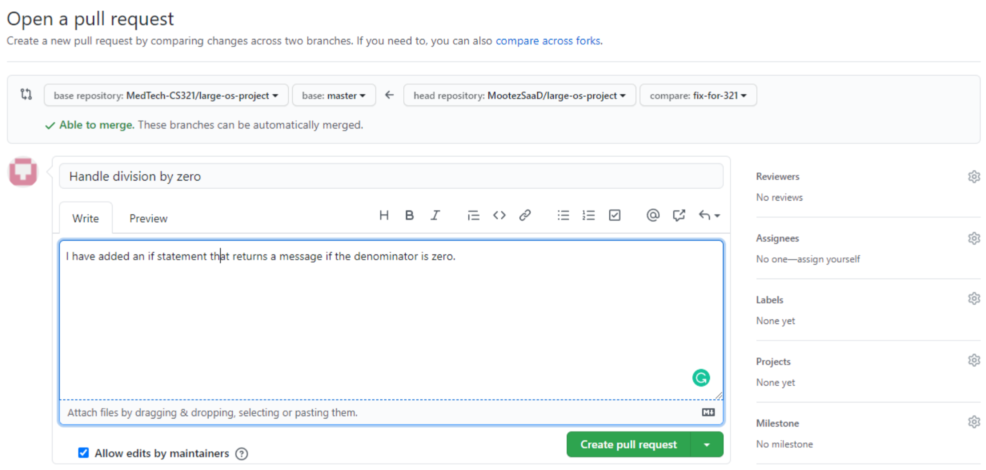
This branch is 1 commit ahead of MedTech-CS321:master.

Fig. 1.16 Click on Pull request to create a new one.

You will be asked to fill the PR’s title and description, in some projects you will need to follow a template that provide a structure for your PR’s description body and title.

At this stage, the maintainers will review your code and perform some tests, if your code passes then your PR will be merged otherwise you would be asked to create a new commit (note that you can create multiple commits per PR, all you have to do is to push your commit as usual and it will be automatically added to the pull request).
Before ending this section I would like to mention what are the "issues" that we have been talking about. Simply, issues are used to track bugs, enhancement, ideas, tasks etc. So if someone discovers a bug in a software or think that they have an idea that can enhance the software, they would need to open an issue at the repository, by going the **Issues** tab next to **Code** and create a new one. Often in large projects, you will need to also follow a template before opening the issue , e.g. stating what is the version that you have tested, the OS that you are operation on etc. It can also be used within development teams to report (and mainly document) issues of many kinds.

In this course, you will be required to work on a project (not only by coding, but applying the ideas that you will encounter), so we hope that you find this section useful since it encapsulates many of the concepts that have been stated in this guide.


Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also compare across forks.

base repository: MedTech-CS321/large-os-project · base: master · head repository: MootezSaaD/large-os-project · compare: fix-for-321

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

Handle division by zero

I have added an if statement that returns a message if the denominator is zero.

Attach files by dragging & dropping, selecting or pasting them.

☑ Allow edits by maintainers

Reviewers: No reviews

Assignees: No one—assign yourself

Labels: None yet

Projects: None yet

Milestone: No milestone

Fig. 1.17 Opening a new Pull request

1.7 Resolving merge conflicts

In order to resolve a merge conflict, we first need to understand what `merging` means. Unlike other version control systems, Git has the ability merge branches together. It will try to incorporate commits from a branch A to a branch B automatically.

Assume a repository with two branches: `master` and `feature-1`. After commit C2, the development team decided to work on a new feature (`feature-1`). The changes—commits C3 and C4—related to this feature were pushed to a new branch, then merged with `master`. A new commit C5 is created indicating that both branches have been merged. Typically, the name of the new commit is in the form: `Merge branch 'feature-1' into 'master'`.

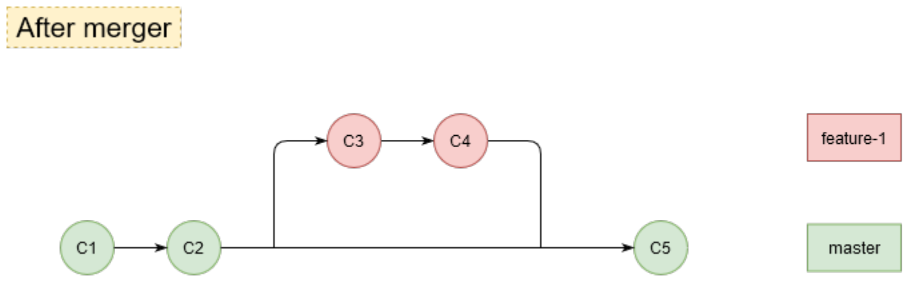


Fig. 1.18 Git network graph after merging branches

As there have been no commits in the branch `master` between C2 and C5, Git had no issues merging the commits of `feature-1` into `master`.

However, there are situations where merging a branch into the current branch might fail. A merge conflict occurs and manual intervention is required in order to proceed.
Merge conflicts can occur when:

- Merging branches
- Cherry picking a commit from another branch
- Rebasing a branch

In this section we will only focus on the first and second type of merge conflicts, those who occur when merging branches. The following diagram shows a different scenario. This time, a commit (C5) has been pushed to the master branch before `feature-1` was merged.

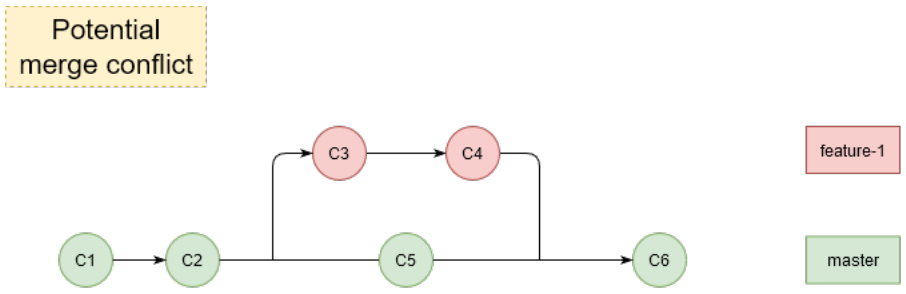


Fig. 1.19 Git network graph with a potential merge conflict

Assume commits C3, C4 and C5 changed the same parts of the same files. If we try to merge `feature-1` and `master`, a merge conflict would arise. Git is not able to automatically create a new merge commit as it previously did since it cannot know which version of the code—the one coming from `feature-1` starting at C4 or that of the `master` branch starting at C5—is the correct one. The process of merging is then paused until a decision is made. We will discuss the possible decisions one can make later on.

So far, we only discussed scenarios where commits were pushed to a new branch then merged back to `master`, but what happens when changes are made to the remote branch that the local branch does not know about yet? To answer this question, we need to introduce two new concepts: **local branches** and **remote branches**.

A **local branch** is a branch that only you, the local user, can see and work on. It exists only on your local machine.

A **remote branch** is a branch on a remote repository. In our case, it is hosted on a server owned by GitHub.

In reality, Git keeps track of all changes made on both the remote and local branch. If a team was only to work on a single remote branch (e.g. `master`), they would each have their own version of that remote branch locally with the same name. Then, whenever a push operation is made Git will identify all the new commits of the local branch then push them to the remote branch for everyone else to see and use.

This means that, since the remote branch is not automatically affected by changes made to the local branch, the local and remote branch can become out of sync. When that happens, a few challenges become apparent:

- How do make sure we have the latest commits on our local branch?
- What happens if the commits pushed to the remote branch modify the same piece of code as the commits made in the local branch?

To visualize these issues, let us resume our work on the repository of section 1.5. Suppose two persons are tasked with improving the code of `divide_number.js`. One decides to add a print statement showing the result of the division, while the other decides to use the conditional (ternary) operator to evaluate for 0 and also comment his code.
As a reminder, here is the current code:

Current code

```
function divide(a,b) {
    if (b == 0) {
        return "Cannot divide by zero!";
    }
    return a/b;
}
```

Both person 1 and person 2 clone the repository at the same time. First, person 1 adds a comment explaining what the function does.

Code

```
// Divides a by b.
function divide(a,b) {
    if (b == 0) {
        return "Cannot divide by zero!";
    }
    return a/b;
}
```

Git Command

```
$ git add .
$ git commit -m "Commented divide function"
[master 63ba811] Commented divide function
 1 file changed, 1 insertions(+), 0 deletions(-)
$ git push
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 8 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 374 bytes | 374.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/MedTech-CS321/large-os-project.git
   4de113e..63ba811  master -> master
```

Notice here that the commit reference (hash) for this change is `63ba811` and that the remote branch (`master`) moved from commit `4de113e` to `63ba811`. However, since both developers cloned at the same time, person 2 is behind the remote branch by one commit. They are at commit `4de113e`.
Person 2 now makes changes to the code:

Code

```
// Divides a by b using a ternary operator
function divide(a,b) {
    return b == 0 ? "Cannot divide by zero!" : a/b;
}
```

They also decide to perform a `git fetch` in order to check whether or not a commit has been made in the meantime.

Git Command

```
$ git add .
$ git commit -m "Updated script to use ternary operators"
[master c8d66ce] Updated script to use ternary operators
 1 file changed, 1 insertion(+), 5 deletions(-)
```

```
$ git fetch
$ git status
On branch master
Your branch and 'origin/master' have diverged,
and have 1 and 1 different commits each, respectively.
  (use "git pull" to merge the remote branch into yours)

nothing to commit, working tree clean
```

Indeed, person 2 has now the confirmation that they are 1 commit behind the remote branch (`origin/master`). In this situation, Git cannot automatically merge the changes. Person 2 needs to manually fix the conflicts before merging otherwise they will not be able to push. This conflict happened because the local branch `master` is out of sync with the remote branch.

Git Command

```
$ git merge origin/master
Auto-merging divide_number.js
CONFLICT (content): Merge conflict in divide_number.js
Automatic merge failed; fix conflicts and then commit the result.
```

At this stage, if person 2 checks their code, they would find the following markers added.

Code

```
<<<<<<< HEAD
// Divides a by b using a ternary operator
function divide(a,b) {
    return b === 0 ? "Cannot divide by zero!": a/b;
}
=======
// Divides a by b
function divide(a,b) {
    if (b == 0) {
        return "Cannot divide by zero!";
    }
    return a/b;
}
```

```
>>>>>>> origin/master
```

In fact, Git adds conflict-resolution markers to the files that have conflicts. This reads as follow: the local version is the top part of that block (everything between "<<<<<<< HEAD" and "======="), while the version fetched from the remote branch is represented by everything below that block. In order to resolve the conflict, you have to either choose one side or the other or merge the contents yourself. It is also important to delete the conflict-resolution markers.

In this case, person 2 and person 1 agree that the second code version (the local version of person 2) is better. The file is then modified to look like this:

Code

```
// Divides a by b using a ternary operator
function divide(a,b) {
    return b === 0 ? "Cannot divide by zero!": a/b;
}
```

Code

```
$ git add .
$ git commit -m "Fixed merge conflicts"
[master 4c0f038] Fixed merge conflicts
$ git push
Enumerating objects: 8, done.
Counting objects: 100% (8/8), done.
Delta compression using up to 8 threads
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 538 bytes | 538.00 KiB/s, done.
Total 4 (delta 2), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (2/2), completed with 1 local object.
To https://github.com/MedTech-CS321/large-os-project.git
   63ba811..4c0f038  master -> master
```

The conflict has been fixed and the push operation went through successfully.

Chapter 2
Further Reading

This chapter has been compiled from various sources, here is a list that can be helpful if you want to learn more about `git`.

- GitHub Essentials - Second Edition - Packt
- GitHub Documentation
- Atlassian Git Tutorials
- Some open source projects where you can find examples of issues, PRs etc: JabRef, elasticsearch

If you face any problems, or have any questions, you can email us at smupor-talta@msb.tn.
Good luck!