Collaborative distributed version control

We have learned how to make a Git repository for a single person. What about sharing?

- Share the folder using email or using some file sharing service: This would lead to many back and forth emails and would be difficult keep all copies synchronized.
- One person's repository on the web: allows one person to keep track of more projects, gain visibility, feedback, and recognition.
- Common repository for a group: everyone can directly update the same repository. Good for small groups.
- Forks or copies with different owners: anyone can suggest changes, even without advance permission. Maintainers approve what they agree with.

Being able to share more easily (going down the above list) is *transformative* (easier to change something, that is you are not the sole owner) because it allows projects to scale to a new level. **This can't be done without proper tools.**

In this lesson we will learn how to keep repositories in sync and how to work with remote repositories on GitHub and other services. We will discover and exercise the centralized as well as the forking workflows, and finally look into how to automate tasks using Git hooks.

⚙ Prerequisites

1. Basic understanding of Git.
2. You need a GitHub account.

We will do this exercise on GitHub but also GitLab and Bitbucket allow similar workflows and basically everything that we will discuss is transferable. With this material and these exercises we do not endorse the company GitHub. We have chosen to demonstrate a number of concepts using examples with GitHub because it is currently the most popular web platform for hosting Git repositories and the chance is high that you will interact with GitHub-based repositories even if you choose to host your Git repository on another platform.

We also encourage course participants to use our Nordic research software repository platform, for more information see https://coderefinery.org/repository/.

Concepts around collaboration

❗ Objectives

- Be able to decide whether to divide work at the branch level or at the repository level.

Instructor note

- 15 min teaching

Motivation

- Someone has given you access to a repository online and you want to contribute?
- We will learn how to make a copy and send changes back.
- Then, we make a “pull request” that allows a review.
- Once we know how code review works, we will be able to propose changes to repositories of others and review changes submitted by external contributors.

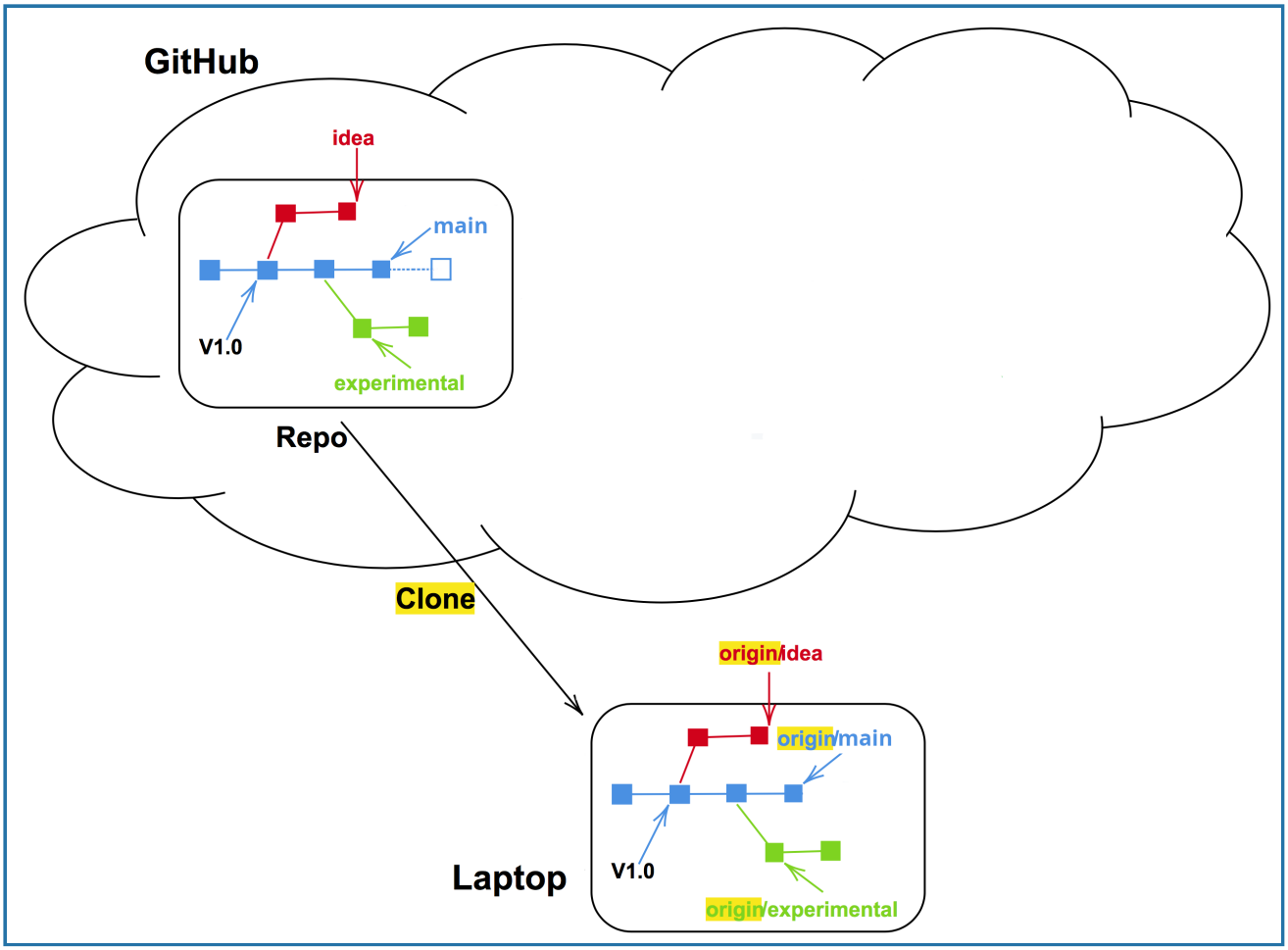
Commits, branches, repositories, forks, clones

- **repository**: The project, contains all data and history (commits, branches, tags).
- **commit**: Snapshot of the project, gets a unique identifier (e.g. `c7f0e8bfc718be04525847fc7ac237f470add76e`).
- **branch**: Independent development line. The main development line is often called `main`.
- **tag**: A pointer to one commit, to be able to refer to it later. Like a commemorative plaque that you attach to a particular commit (e.g. `phd-printed` or `paper-submitted`).
- **cloning**: Copying the whole repository to your laptop - the first time. It is not necessary to download each file one by one.
- **forking**: Taking a copy of a repository (which is typically not yours) - your copy (fork) stays on GitHub/GitLab and you can make changes to your copy.

Cloning a repository

In order to make a complete copy a whole repository, the `git clone` command can be used. When cloning, all the files, of all or selected branches, of a repository are copied in one operation. Cloning of a repository is of relevance in a few different situations:

- Working on your own, cloning is the operation that you can use to create multiple instances of a repository on, for instance, a personal computer, a server, and a supercomputer.
- The parent repository could be a repository that you or your colleague own. A common use case for cloning is when working together within a smaller team where everyone has read and write access to the same git repository.
- Alternatively, cloning can be made from a public repository of a code that you would like to use. Perhaps you have no intention to work on the code, but would like to stay in tune with the latest developments, also in-between releases of new versions of the code.

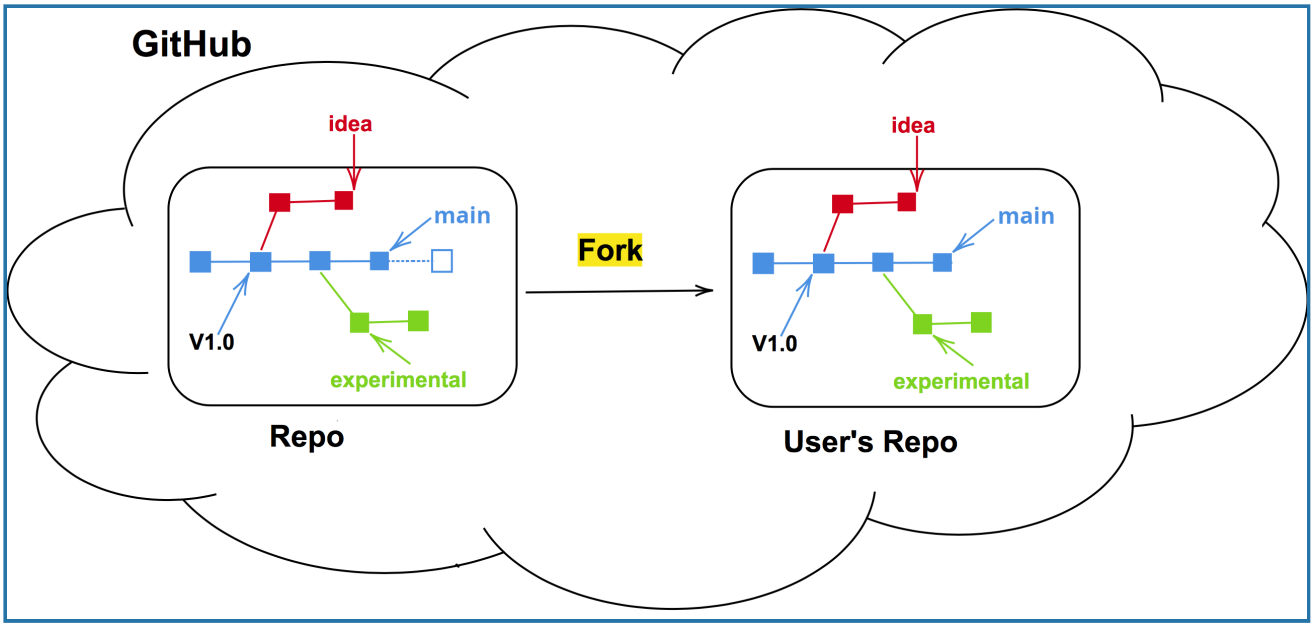


Cloning

Forking a repository

When a fork is made on GitHub/GitLab a complete copy, of all or selected branches, of the repository is made. The copy will reside under a different account on GitHub/GitLab. Forking of a repository is of high relevance when working with a git repository to which you do not have write access.

- In the fork repository commits can be made to the base branch (`main` or `master`), and to other branches.
- The commits that are made within the branches of the fork repository can be contributed back to the parent repository by means of pull or merge requests.

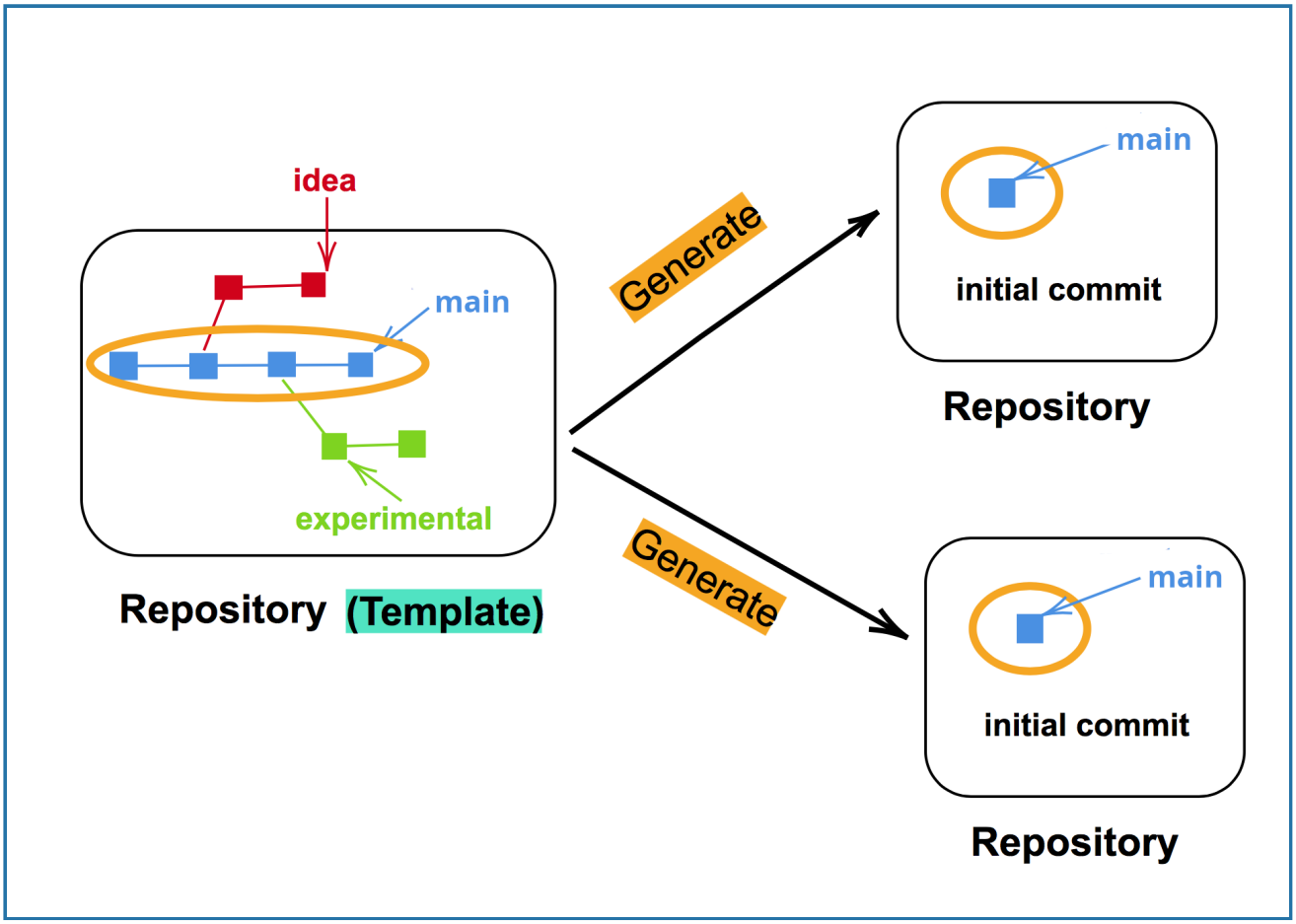


Forking

Generating from templates and importing

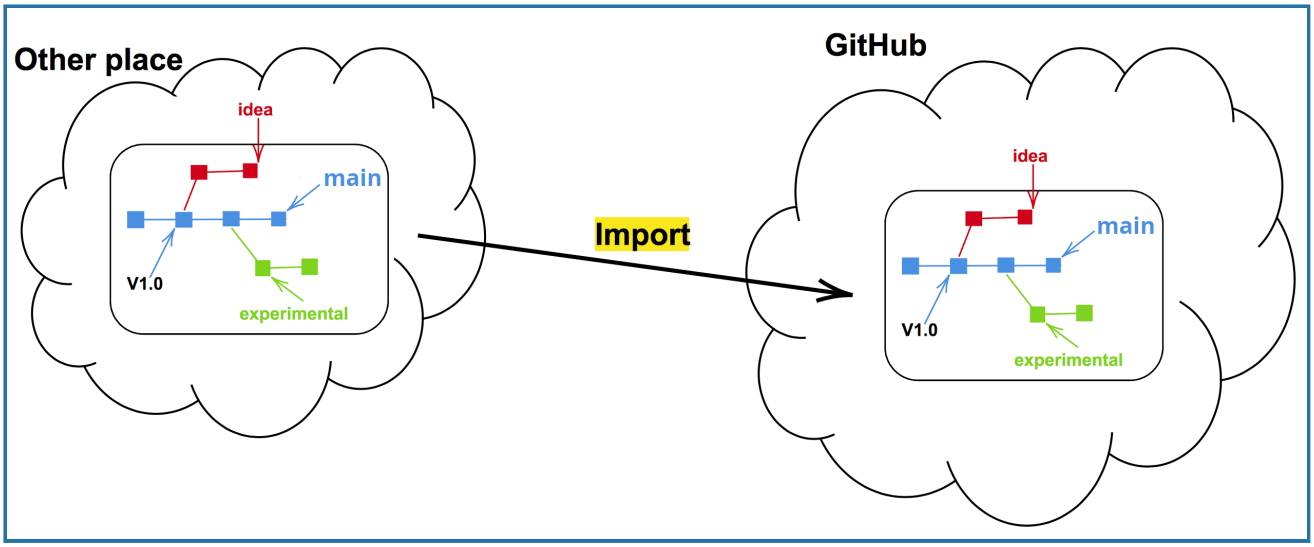
There are two more ways to create “copies” of repositories into your user space:

- A repository can be marked as **template** and new repositories can be **generated** from it, like using a cookie-cutter. The newly created repository will start with a new history, only one commit, and not inherit the history of the template.



Generating from a template.

- You can **import** a repository from another hosting service or web address. This will preserve the history of the imported project.



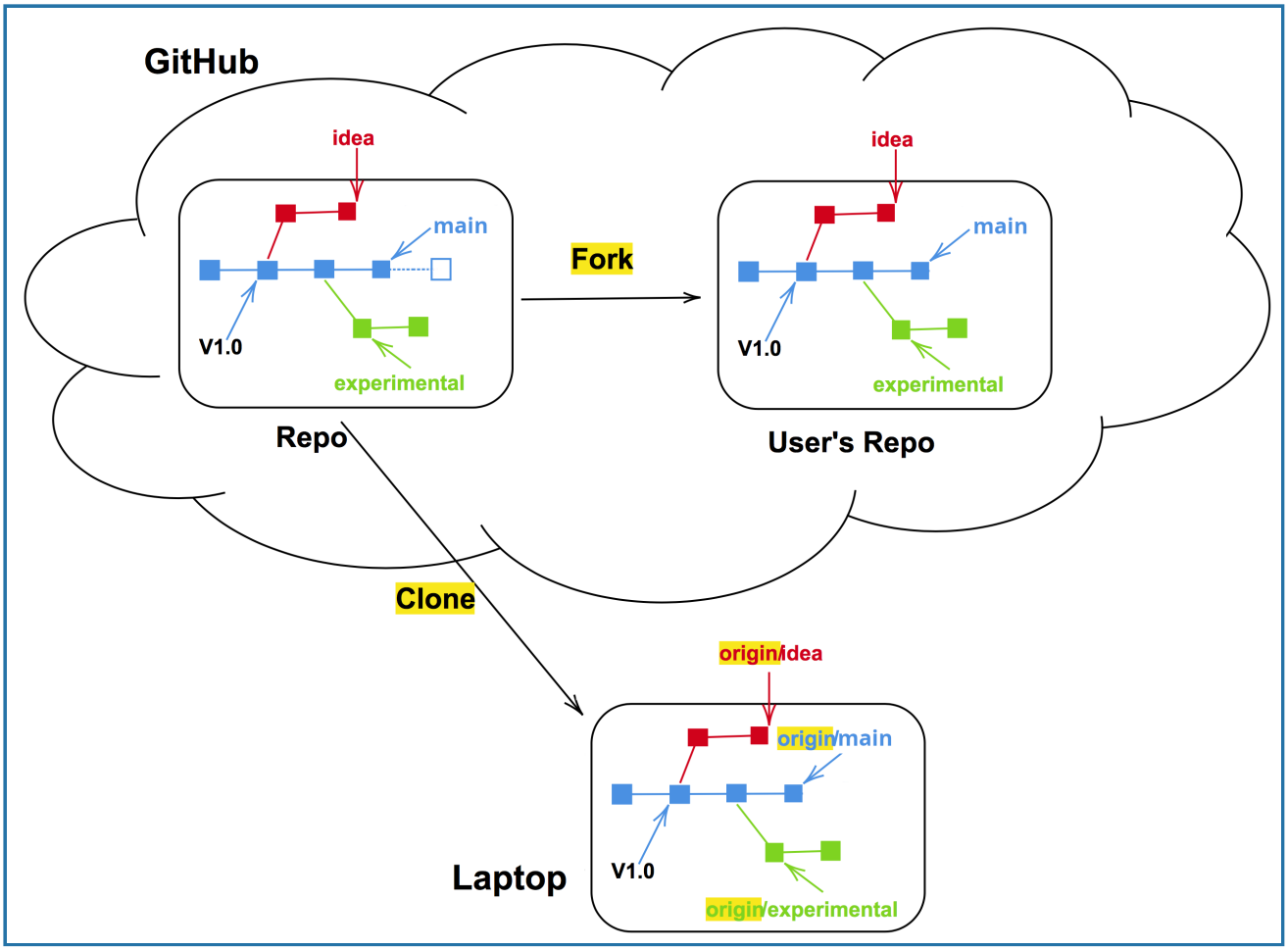
Importing a repository.

💬 Discussion

- Visit one of the repositories/projects that you have used recently and try to find out how many forks exist and where they are.
- In which situations could it be useful to start from a "template" repository by generating?

Synchronizing changes between repositories

- We need a mechanism to communicate changes between the repositories.
- We will **pull** or **fetch** updates **from** remote repositories (we will soon discuss the difference between pull and fetch).
- We will **push** updates **to** remote repositories.
- We will learn how to suggest changes within repositories on GitHub and across repositories (**pull request**).
- Repositories that are forked or cloned do not automatically synchronize themselves: We will learn how to update forks (by pulling from the "central" repository).
- A main difference between cloning a repository and forking a repository is that the former is a general operation for generating copies of a repository to different computers, whereas forking is a particular operation implemented on GitHub/GitLab.



Forking and cloning

Centralized workflow

Objectives

- Understand how to collaborate using a centralized workflow.
- Understand the difference between local branch, origin/branch, and remote branch.

Instructor note

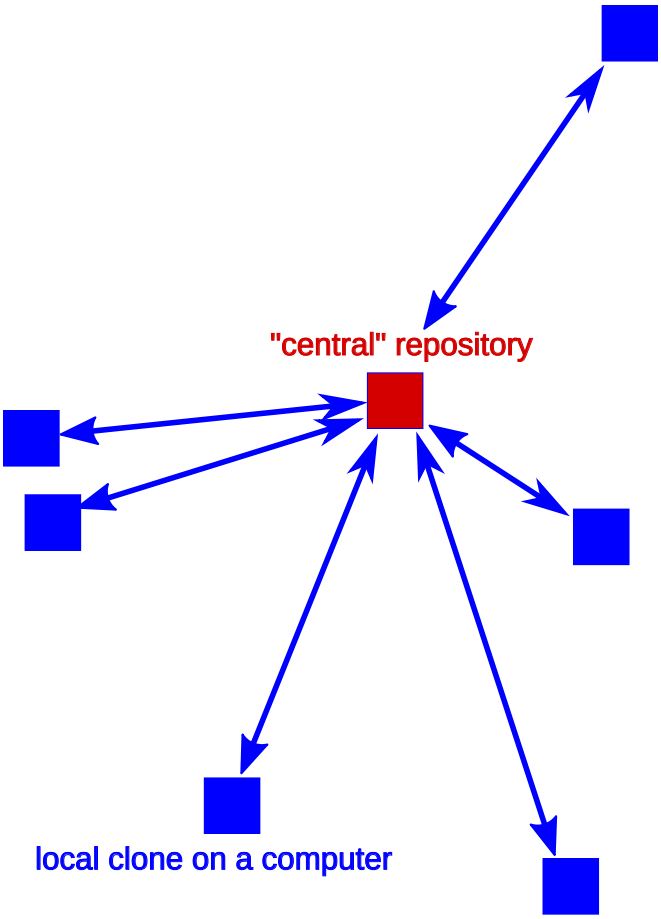
- 30 min teaching
- 30 min exercises

Meaning of "central" in a distributed version control

In this episode, we will explore the usage of a **centralized workflow** for collaborating online on a project **within one repository on GitHub**. This means that everyone has access to that **central repository** - convenient (but doesn't scale to a huge project).

In the next section, we will see that Git is **distributed** version control. This means that any type of repository links that you can think of can be implemented - not just "everything connects to one central server".

Centralized layout



*Centralized layout. **Red** is the repository on GitHub. **Blue** is where all contributors work on their own computers.*

Features:

- Typically all developers have both read and write permissions (double-headed arrows).
- Suited for cases where **all developers are in the same group or organization or project**.
- **Everybody who wants to contribute needs write access**.
- Good idea to write-protect the main branch (typically `main` or `master`).

Real life examples:

- Within the CodeRefinery team we mostly use this approach: https://github.com/coderefinery
- https://github.com/ropensci/plotly

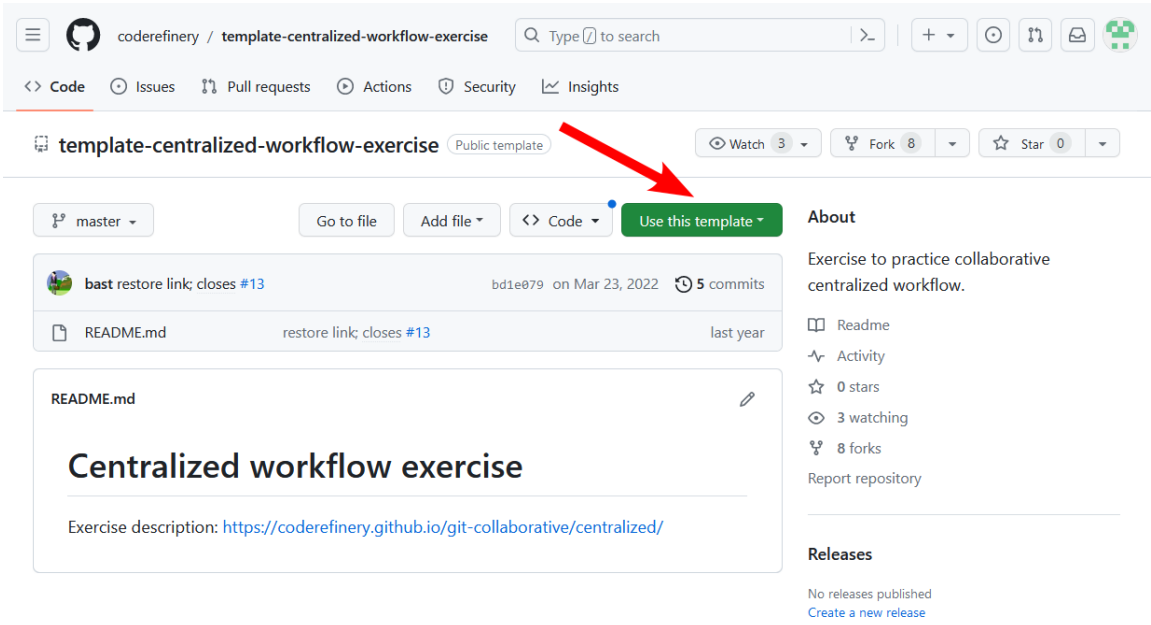
Exercise preparation

In this exercise we will practice collaborative centralized workflow in groups (but you can also collaborate with us as individual). One person (**maintainer**) will create the exercise repository, and **collaborators** will contribute to it. We'll discuss how this leads to code review and discuss a number of typical pitfalls.

⚙ Exercise preparation

Part of team/exercise room | Following on your own

- Form not too large groups (4-5 persons).
- Each group needs to appoint someone who will host the shared GitHub repository: the *maintainer*. This is typically the exercise lead (if available). Everyone else is a *collaborator*.
- The **maintainer** (one person per group) generates a new repository from the template https://github.com/coderefinery/template-centralized-workflow-exercise called `centralized-workflow-exercise` (There is no need to tick "*Include all branches*" for this exercise):

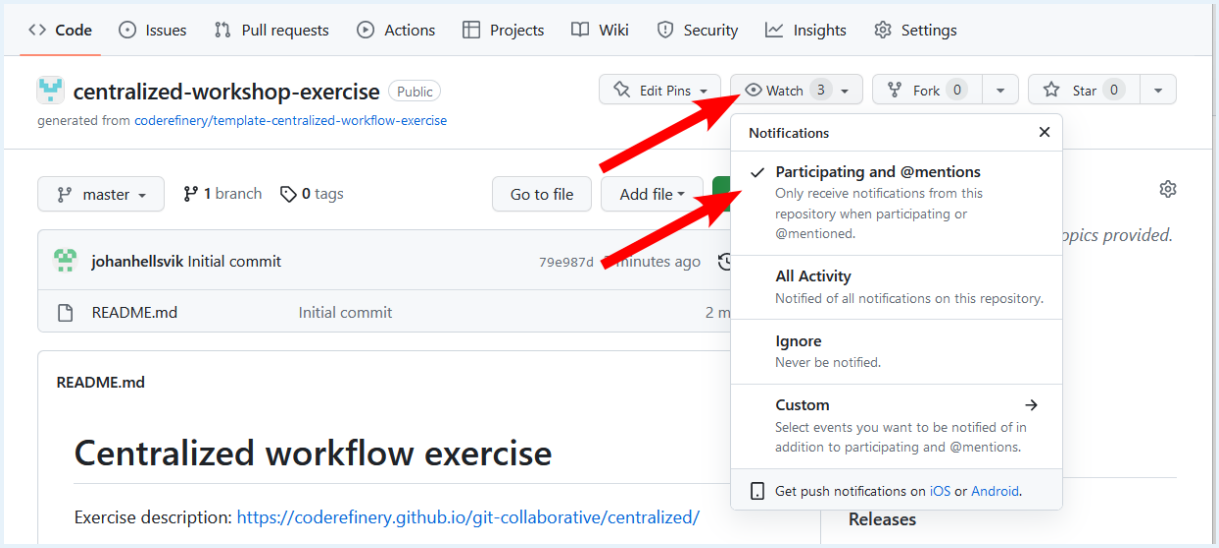


- Then **everyone in your group** needs their GitHub account to be added as collaborator to the exercise repository:
  - Collaborators give their GitHub usernames to their chosen maintainer.
  - Maintainer gives the other group members the newly created GitHub repository URL.
  - Maintainer adds participants as collaborators to their project (Settings → Manage Access → Invite a collaborator).

- **Don't forget to accept the invitation**
  - Check https://github.com/settings/organizations/
  - Alternatively check the inbox for the email account you registered with GitHub. GitHub emails you an invitation link, but if you don't receive it you can go to your GitHub notifications in the top right corner. The maintainer can also "copy invite link" and share it within the group.

- **Watching and unwatching repositories**

- Now that you are a collaborator, you get notified about new issues and pull requests via email.
- If you do not wish this, you can "unwatch" a repository (top of the project page).
- However, we recommend watching repositories you are interested in. You can learn things from experts just by watching the activity that come through a popular project.



Unwatch a repository by clicking "Unwatch" in the repository view, then "Participating and @mentions" - this way, you will get notifications about your own interactions.

Exercise: Part 1 - creating a pull request

✍ Centralized-1: Clone a repository, add a file, push changes as a branch, and create a pull request

- Before we start with the exercise, instructor points to the preparation (above).
- Then work on steps A-H.
- There are also optional exercises.
- Before and after each action you take, run the following informational commands:
  - `git graph` - almost every time
  - `git status` - when you modify files

❗ Hint for breakout rooms

If the helper in the room is the one who sets up the central repository, they cannot easily demostrate the steps via screen-sharing as the repository's maintainer. A good alternative is to have one of the learners screen-share and get advice on the steps from other learners and helpers!

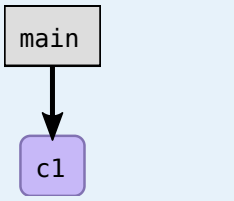
Step A. Clone your maintainer's group repository

```
$ git clone REPOSITORY-URL centralized-workflow-exercise
```

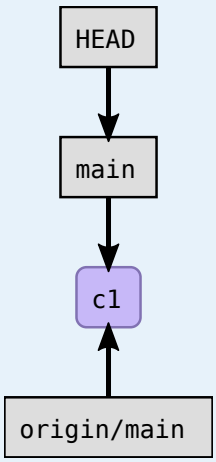
Where `REPOSITORY-URL` is the repository created by the exercise maintainer.

Clone using the SSH path you get from the webpage (the one that starts with `git@github.com:`), not the one that starts with `https://github.com`. Otherwise, you won't be able to push later.

Representation of what happens when you clone



remote or central



local

Here and in what follows, "c1" is a commit, "b1" etc. are commits on side branches and "m1" is a merge commit.

- We clone the entire history, all branches, all commits. In our case, we have one branch (we did not include *all branches* when creating our repository from template) and we have only one commit (*initial commit*).
- `git clone` creates pointers `origin/main` so you can see the branches of the origin.
- `origin` refers to where we cloned from.
- `origin` is a shortcut for the full URL.
- `origin/main` is a read-only pointer.
- The branches starting with `origin/` only move during `git pull` or `git fetch` or `git push`.
- Only `git pull` or `git fetch` or `git push` require network.
- All other operations are local operations.

Step B. Change directory into the newly created directory

```
$ cd centralized-workflow-exercise
```

Try to find out where this repository was cloned from using `git remote -v`.

Step C. Create a branch `yourname-somefeature` pointing at your commit

Create a branch from the current `main`. Also adapt "yourname-somefeature" to a better name:

```
$ git branch yourname-somefeature main
$ git switch yourname-somefeature
```

The `yourname-` prefix has no special meaning here (not like `origin/`): it is just part of a branch name to indicate who made it.

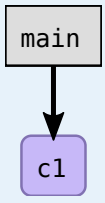
Step D. Create a file with a unique name, e.g.: `yourusername.txt`

In this file share your favourite cooking recipe or haiku or Git trick or whatever (we will push soon to a public repository so don't share something you don't want to become public for the duration of the exercise).

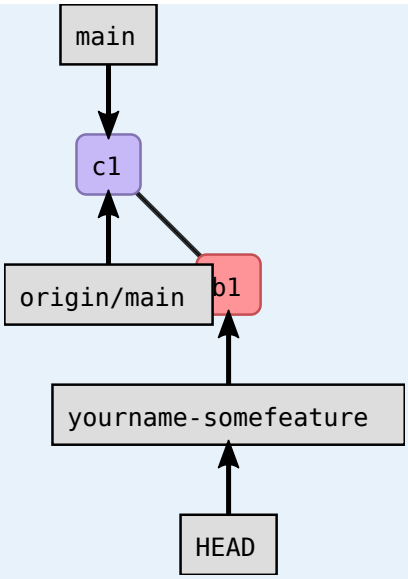
Step E. Stage and commit the change

```
$ git add yourusername.txt
$ git commit
```

The commit only exists locally



remote or central



local (read figure left to right)

Step F. Push your change as a new branch

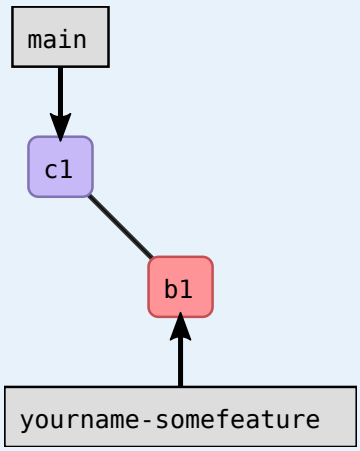
```
$ git push origin -u yourname-somefeature
```

If you get a password request for https://github.com when you try to push

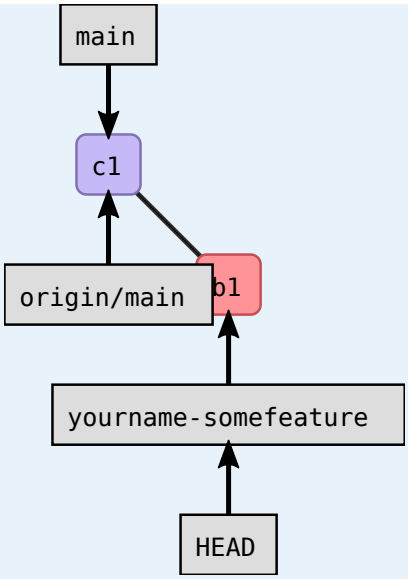
Probably you cloned with the HTTPS URL (see `git remote -v` to confirm). You can change this to SSH by going to the repository page, clicking "Code", copying the SSH URL (starts with `git@github.com:`), and then updating the URL with:

```
$ git remote set-url origin SSH-REPOSITORY-URL
```

Now the commit also exists on the remote repository



remote or central (read figure left to right)



local (read figure left to right)

Meaning of -u

The `-u` or `--set-upstream` will connect the local branch with the newly created upstream/remote branch and track it. This has the following advantages:

- If you from here on only type `git push` or `git pull` without branchname, Git will know what branch you refer to (depending also on your Git configuration). However, we still recommend to explicitly type where you want to push/pull to/from and which branch explicitly.
- When you type `git status`, Git will inform you whether your local branch is ahead or behind the upstream branch that it tracks.

However, also without the `-u` this step and the rest of the exercise will work. The fact that the local and remote branch are not connected is not a problem if you explicitly type out the remote and branch name every time.

Step G. Browse the network of branches and commits

After you have pushed your branch and other participants have too, browse the network of branches and commits (on GitHub click on Insights -> Network) and discuss what you see.

Step H. Submit a pull request

Submit a pull request from your branch towards the `main` branch. Do this through the web interface.

Meaning of a pull request: think of it as change proposal. In a popular project, it means that anyone can contribute with *almost no work* on the maintainer's side - a big win.

There are **several options to open a pull request**:

- Follow the link printed to terminal output when git-pushing a branch to GitHub/GitLab
- Visit the GitHub repository in the browser after pushing the branch and click on the green button “Compare & pull request”
- Click on “Pull requests” on top of the GitHub repository and either “Compare & pull request” or “New pull request”
- Click on “Branches” and then “New pull request” from the respective branch

Exercise: Part 2 - code review and merging changes

✍ Centralized-2: Merge the pull requests (together)

- **We do step 2A and 2B together** (instructor demonstrates, and everybody follows along in their repositories).

Instructor note

At this stage it might be good to show how to submit and how to review a pull request.

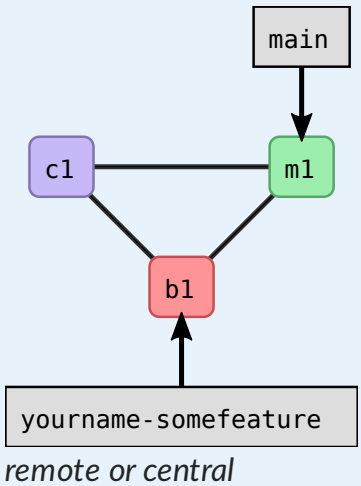
- When co-teaching change roles and switch screenshares also.
- Discuss what you look at when submitting.
- Discuss what you look at when reviewing.

Step 2A. Discuss and accept pull requests

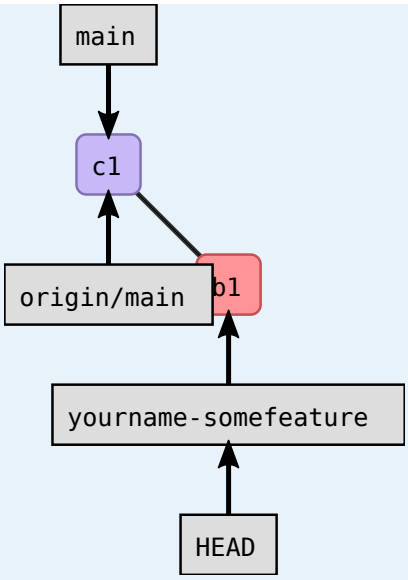
We do this step together on the main screen (in the main room) or on stream. The instructor shows a submitted pull request, discusses what features to look at, and how to discuss and review.

At the same time, helpers can review open pull requests from their exercises groups.

❗ Once the pull-request is accepted, the change is merged



remote or central



local

Finally also discuss the “network” on GitHub.

Instructor note

At this stage demonstrate how to suggest small changes to pull/merge requests:

- GitHub
- GitLab

💬 Discussion

Naming

- In GitLab or BitBucket these are named **merge requests**, not **pull requests**.
- Which one do you feel is more appropriate and in which context? (The name **pull request** may make more sense in the forking workflow: next episode).
- It can be useful to think of them as **change proposals**.

Pull requests can be used for code review

- We recommend that pull requests are reviewed by someone else in your group.
- Collaborative learning
- OK if students and junior researchers review senior researchers
- In our example everyone has write access to the “central” repository.

Pull requests are from branch to branch

- They originate from a source branch and are directed towards a branch.
- Not from commit to branch.
- Pull requests create new commits on the target branch.
- They do not create new branches.

Protected branches

- A good setting for large projects is to make the `main` branch **protected** and all changes to it have to go through code review.
- Centralized workflow with protected branches is a good setup for many projects.

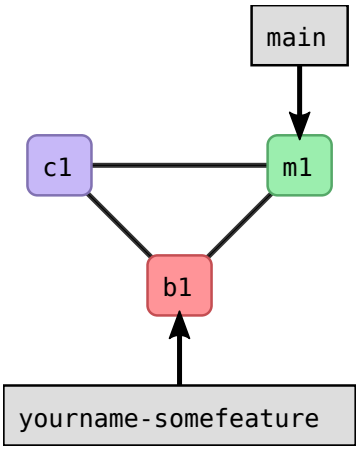
Read more

- This is a great resource: Practical git PRs for small teams

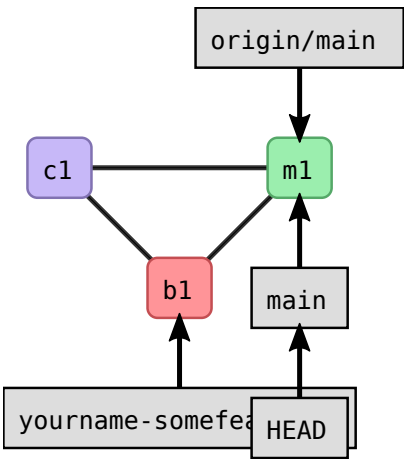
Step 2B. Update your local copy

Your branch `yourname-somefeature` is not needed anymore but more importantly, you need to sync your local copy: **Everybody needs to do this step in their exercise repository** but we do this together in the main room so that we can discuss this step and ask questions.

```
$ git switch main
$ git pull origin main
```



remote or central



local

Optional exercises

✍ (optional) Centralized-3: Cross-referencing issues using "#N"

We will submit another change by a pull request but this time we will **first create an issue**.

1. Open an issue on GitHub and describe your idea for a change. This gives others the chance to give feedback/suggestions. **Note the issue number**, you will need it in step 3.
2. Create a new branch and switch to it.
3. On the new branch create a commit and in the commit message write what you did, but also add that this "closes #N" (replace N by the actual issue number from step 1).
4. Push the branch and open a new pull request. If you forgot to refer to the issue number in step 3, you can still refer to it in the pull request form (add a "closes #N" to the title or description).
5. Note how now commits, pull requests, and issues can be cross-referenced by including `#N`.
6. Notice how after the pull request is merged, the issue gets automatically closed. This only happens for certain keywords like `closes` or `fix`.
7. Discuss the value of cross-referencing them and of auto-closing issues with commits or pull requests.

See also the GitHub documentation for more examples.

✍ (optional) Centralized-4: Why did we create a feature branch "yourname-somefeature"? (exercise/discussion)

Pushing directly to the main branch is perfectly fine for simple personal projects - the pull-request workflows covered here are for larger projects or for collaborative development. Guidelines for simpler workflows are given in the how much Git is necessary? episode of the git-intro lesson.

In collaborative development, whenever we update our repository we create a new branch and create a pull-request. Let's now imagine that everyone in your group (or one person on two different clones) makes a new change (create a new file) but without creating a new branch.

1. You all create a new file in the main branch, stage and commit your change locally.
2. Try to push the change to the upstream repository:

```
$ git push origin main
```

   You probably see something like this:

```
$ git push

To https://github.com/user/repo.git
 ! [rejected]        main -> main (non-fast-forward)
error: failed to push some refs to 'https://github.com/user/repo.git'
To prevent you from losing history, non-fast-forward updates were rejected
Merge the remote changes (e.g. 'git pull') before pushing again.  See the
'Note about fast-forwards' section of 'git push --help' for details.
```

- The push only worked for one participant (one clone).
- Discuss why push for everybody else in this group was rejected?

✔ Solution

The push for everyone except one person fails because they are missing one commit in their local repository that exists on the remote. They will first need to pull the remote changes before pushing their own, which will usually result in a merge commit.

💬 Discussion: How to make changes to remote branches

If there is a remote branch `somefeature`, we can create a local branch and start tracking `origin/somefeature` like this:

```
$ git switch somefeature
```

Once we track a remote branch, we can pull from it and push to it:

```
$ git pull origin somefeature
$ git push origin somefeature
```

We can also delete remote branches:

```
$ git push origin --delete somefeature
```

❗ Creating pull requests from the command line

There are several possibilties:

- https://cli.github.com/
- https://hub.github.com/

- https://github.com/NordicHPC/git-pr

How you can find out in which repositories you are a collaborator

Visit https://github.com/settings/repositories where you will see an overview of all repositories you have write access to.

GitHub/GitLab organizations

- Projects often start under a personal namespace.
- If you want the project to live beyond the interest or work time of one person, one can share projects under an "organization".
- You can then invite collaborators to an organization.
- This is what we do in the CodeRefinery project: https://github.com/coderefinery

Keypoints

- Centralized workflow is often used for remote collaborative work.
- `origin` refers to where you cloned from (but you can relocate it).
- `origin/mybranch` is a read-only pointer to branch `mybranch` on `origin`.
- These read-only pointers only move when you `git fetch` / `git pull` or `git push`.

Distributed version control and forking workflow

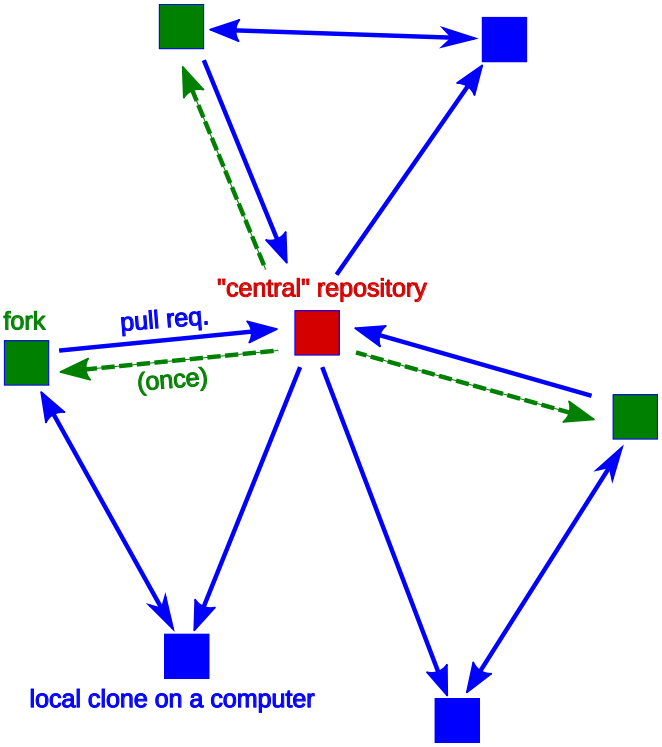
Objectives

- Get a mental representation of what is happening on GitHub/GitLab.
- Get comfortable with the forking workflow.

Instructor note

- 30 min teaching
- 30 min exercises

Forking layout



*Forking workflow. **Red** is the central repository, where only owners have access. **Green** are forks on GitHub (copy for a single user to work on). **Blue** are local copies where contributors work on their own computer.*

In the **forking layout**, again we call one repository the "central" repository but people push to **forks** (their own copies of the repository on GitHub/GitLab/Bitbucket).

Features:

- **Anybody can propose contributions without asking for advance permission** (to public projects).
- Maintainer still has **full control over what is merged**.
- Contributors now have **more than one remote** to work with.

This is used by almost all large (and small) open-source projects these days. Real life examples:

- NumPy
- JupyterLab

💬 Discussion: Why do we create a fork and not only a clone?

- It is not easy to show my changes on my computer to somebody else.
- The maintainer cannot pull from my laptop.
- The cloud provider is a trusted place to pull from and allows review before pulling.
- Backup.

Working with multiple remotes

In the forking layout described above we work with **multiple remotes**, in this case **two remotes**: One remote refers to the "central" repository, and the other remote refers to the fork.

- There is nothing special about the name `origin`. The `origin` is an alias/placeholder (think of "sticky note" referring to an URL).
- We can name these aliases as we like.
- We can add and remove remotes:

```
$ git remote add upstream https://github.com/project/project.git
$ git remote rm upstream

$ git remote add group-repo https://example.com/exciting-project.git
$ git remote rm group-repo

$ git remote add upstream https://github.com/project/project.git
$ git remote add downstream https://github.com/userX/project.git
```

- To see all remotes:

```
$ git remote --verbose
```

Warning

We will work with a new repository for this exercise!

For this exercise we will fork a different repository compared to earlier today. Please step out of the repository and check that you fork the **forking**-workflow-exercise.

Exercise preparation

Exercise preparation

Part of team/exercise room | Following on your own

Maintainer (team lead):

- Create an exercise repository by generating from a template using this template: https://github.com/coderefinery/template-forking-workflow-exercise called `forking-workflow-exercise`
- In this case we **do not add collaborators** to the repository (this is the point of this example).
- Share the link to the newly created repository with your group.

Learners in exercise team: Fork the newly created repository (not the "coderefinery" one) and then **clone your fork**.

Exercise: Part 1 - creating a pull request

✍ Distributed-1: Fork a repository and create a pull request

As an example we will collaboratively develop a cookbook for taco recipes, inspired by tacofancy.

Objectives:

- Learn how to fork, modify the fork, and open a pull request towards the central repository.
- Learn how to update your fork with changes that others have already made to the forked repository.

Exercise:

- Maintainer prepares an exercise repository (see above; this will take 5-10 minutes).
- **Learners work on steps A-F** (15-20 minutes).
- There are two optional steps after step E for those who want more.
- After step E you take a break or help others. Please ask questions both during group work and in the collaborative document.
- **We will review the pull requests together** and then update forks.

Before and after each action you take, run the following informational commands. Carefully observe what happens, especially in `git graph`:

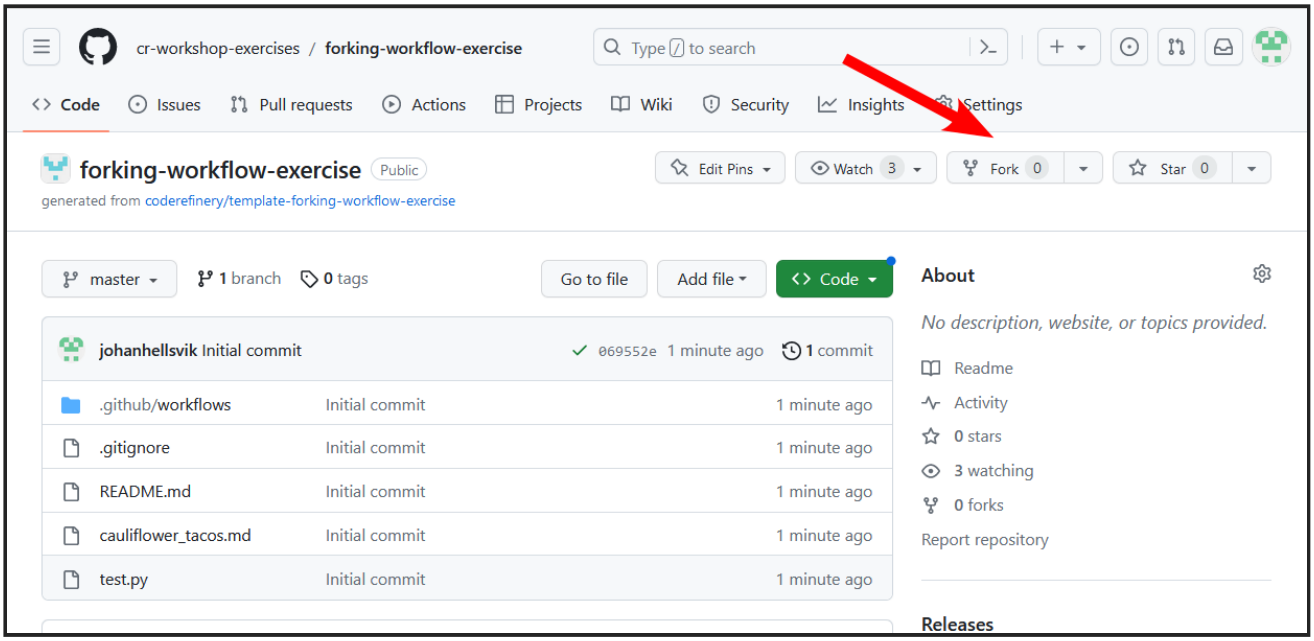
- `git graph` - almost every time. As a reminder, to define `git graph` use:

```
$ git config --global alias.graph "log --all --graph --decorate --oneline"
```

- `git status` - when you modify files.

Step A: Fork and clone

First fork the exercise repository (please carefully check with your exercise group which repository you should fork).

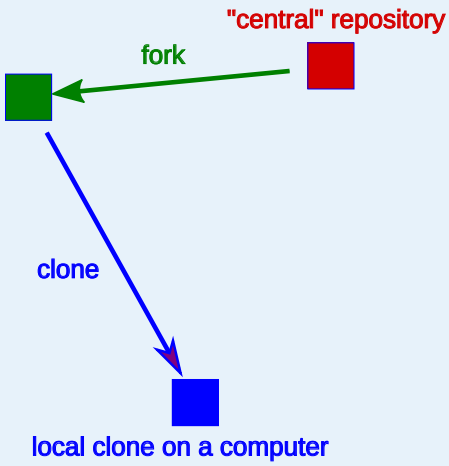


How to fork.

Then **clone your fork** to your computer. The repository URL should include your username. **Clone using the SSH path you get from the webpage, not the https URL** from the web browser. The URL should start with `git@github.com:`. Otherwise, you won't be able to push later.

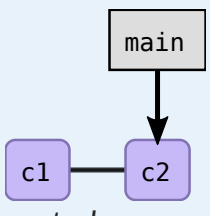
Pictorial representation of this step

Here is a pictorial representation of this part:

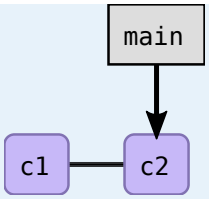


Forking followed by cloning.

This is how it looks after we fork:



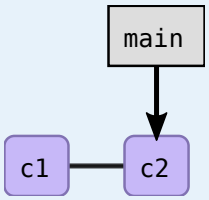
central



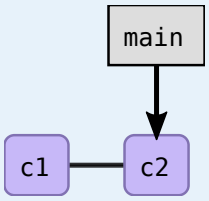
fork

- A fork is basically a (bare) clone.
- The forked repo and the fork are in principle independent repositories.
- When forking we copy all commits, all branches.

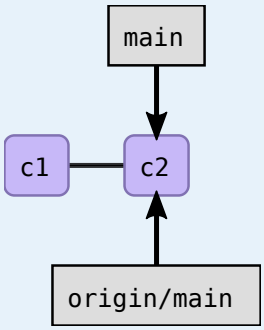
After we clone the fork we have three in principle independent repositories:



central



fork



local

Step B: Open an “issue” as a change proposal

Before we start any coding, open a new “Issue” on the central repository as a “proposal” where you describe your idea for a recipe with the possibility to collect feedback from others. After creating this issue note the issue number. We will later refer to this issue number.

Discuss why it can be useful to open an issue before starting the actual coding.

Step C: Modify and commit

Before we do any modification, we create a new branch and switch to it: this is a good reflex and a good practice. Choose a branch name which is descriptive of its content. For example:

```
$ git branch myname-feature    # describes both who it belongs to and the purpose
$ git switch myname-feature
```

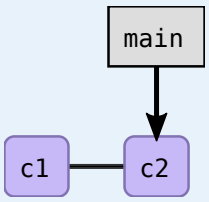
On the new branch create a new file which will hold your recipe, for instance `traditional_coderefinery_tacos.md` (but change the name). You can get inspired here. Hopefully we all use different file names, otherwise we will experience conflicts later (which is also interesting!).

There is also a file called `test.py` which will automatically verify whether your recipe contains the string "taco" (case insensitive). This is there to slowly introduce us to automated testing.

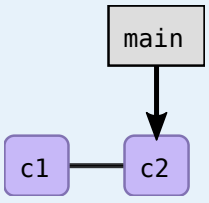
Once you are happy with your recipe, commit the change and in your commit message **reference the issue** which you have opened earlier with "this is my commit message; closes #N" (use a more descriptive message and replace N by the actual issue number from step B).

Pictorial representation of this step

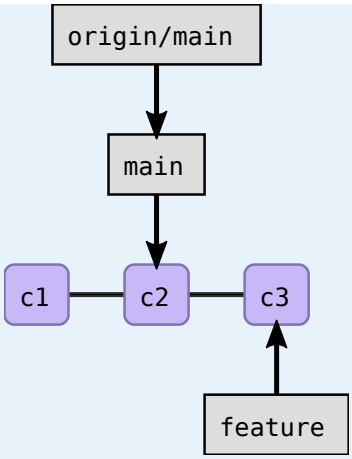
And here is a picture of what just happened:



central



fork



local

Step D: Push your changes to the fork

Now push your new branch to your fork. Your branch is probably called something else than "myname-feature". Also verify where "origin" points to.

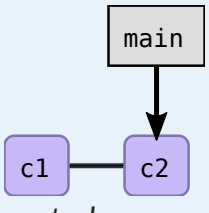
```
$ git push origin myname-feature
```

If you get a password request for https://github.com when you try to push

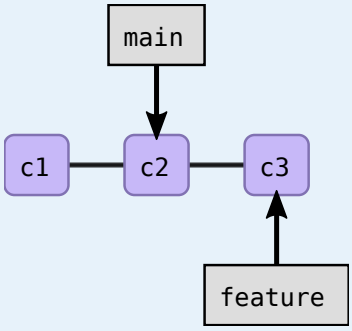
Probably you cloned with the HTTPS URL (see `git remote -v` to confirm). You can change this to SSH by going to the repository page, clicking "Code", copying the SSH URL (starts with `git@github.com:`), and then updating the URL with:

```
$ git remote set-url origin REPOSITORY-URL
```

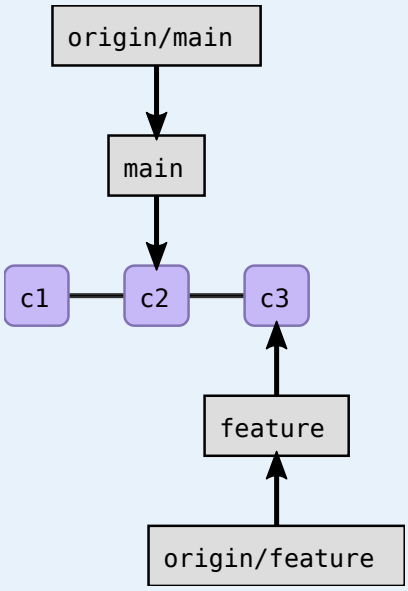
Pictorial representation of this step



central



fork

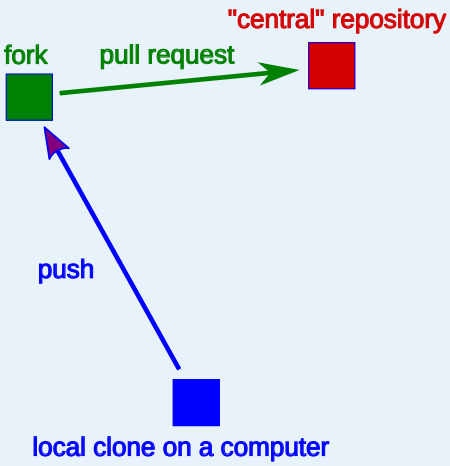


local

Step E: Open a pull request

Then file a pull request from the branch on your fork towards the main branch on the central repository.

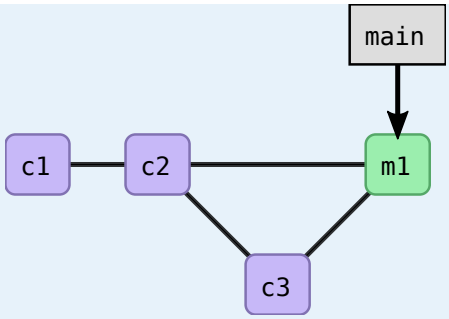
Pictorial representation for steps D and E



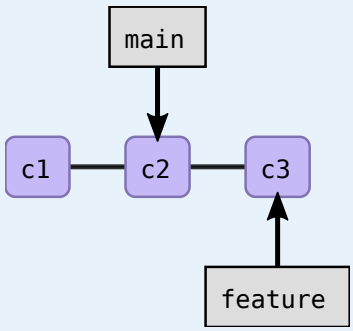
Push followed by a pull request.

A pull-request means: “please review my changes and if you agree, merge them with a mouse-click”.

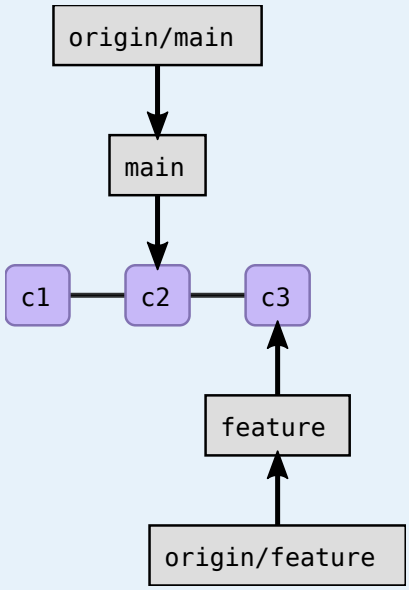
Once the pull-request is accepted, the change is merged:



central



fork



local

Step F: View your pull request

Go to the central repository, and list all pull requests and view your your pull request. Remember that the pull request is created on the original repostory.

Wait here until we integrate all pull requests into the central repo together.

Observe how the issues automatically close after the pull requests are merged (provided the commit messages contain the right keywords).

✍ (optional) Distributed-2: Send a conflicting pull request

If you complete parts A-E much earlier than others, try to open another pull request where you anticipate a conflict with your first pull request.

✍ (optional) Distributed-3: Making changes to your pull request after it has been opened.

You can do that by pushing **additional commits to the same branch** where you opened the pull request from. Observe how they end up added to your pull request.

Exercise: Part 2 - code review and merging changes

We do this step together. The instructor shows a submitted pull request, discusses what features to look at, and how to discuss and review.

At the same time, maintainers can review open pull requests from their exercises groups.

✍ (optional) Distributed-4: Squash merge a pull request

If you complete this exercise much earlier than others, create a new pull request with two or more commits.

Then, when reviewing the change as maintainer, accept these with “Squash and merge” and later compare the source and target repositories/branches how they differ after the small commits got squashed into one.

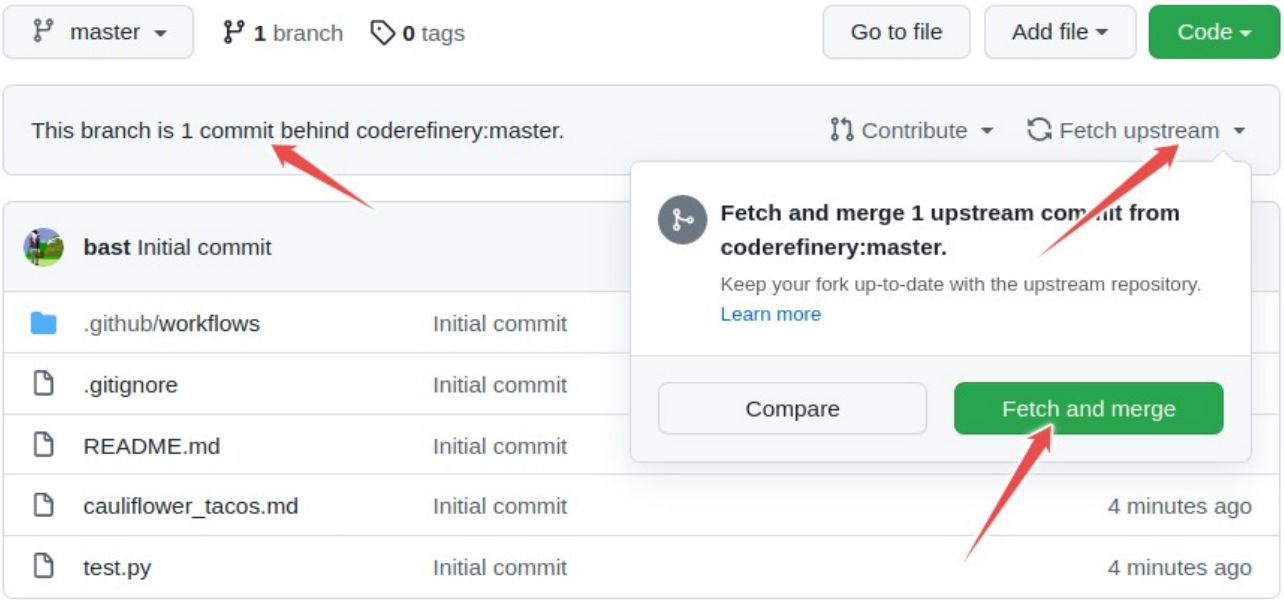
Exercise: Part 3 - Updating forks

We do this part **after the contributions from all participants have been integrated**.

Once this is done, practice to update your fork with the merged changes from others and verify that you got the files created by other participants.

Make sure that the contributions from other participants are not only on your local repository but really also end up in your fork.

On GitHub it is possible to update the fork by pressing a button (see screenshot below):



Updating the fork via GitHub web interface.

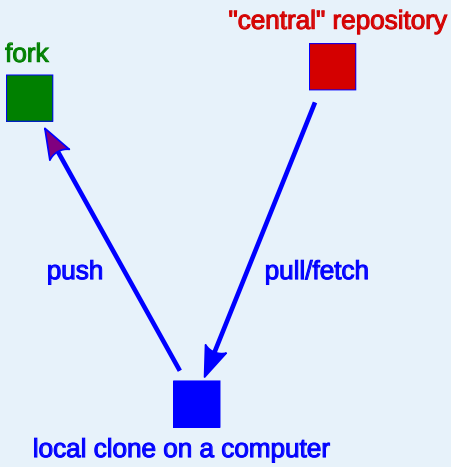
Updating the fork using the command line

Remotes are aliases. We can use the URLs a remote directly. instead of aliases like `origin` or `upstream`.

Here we pull from the central repo and push to our fork:

```
$ git switch main
$ git pull CENTRAL-REPOSITORY-URL main
$ git push FORK-URL main
```

Here is a pictorial representation of this part:

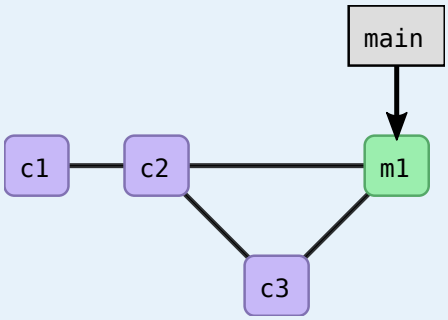


Pull followed by push to a different remote.

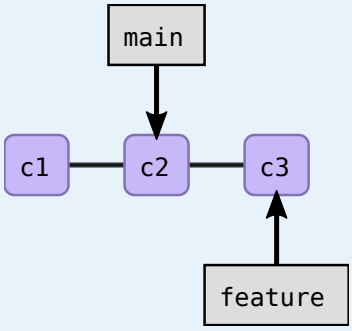
Updating the fork using the command line (longer version)

Below is a step by step recipe with pictorial representations which hopefully makes clear what happens in each step.

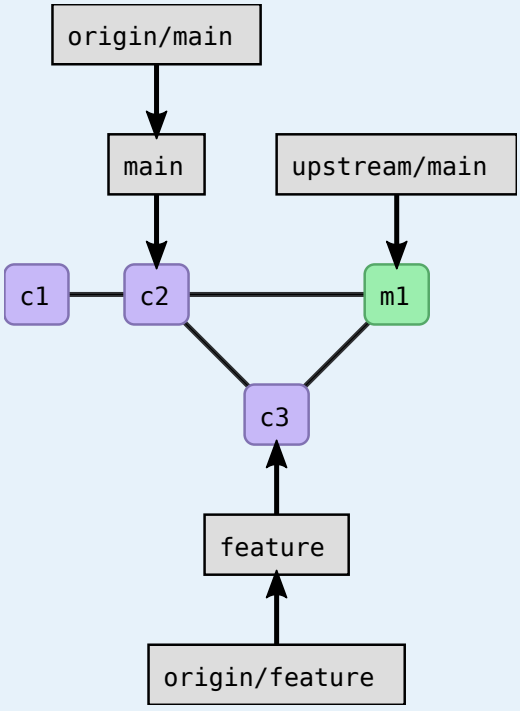
```
$ git remote add central CENTRAL-REPOSITORY-URL
$ git fetch central
```



central

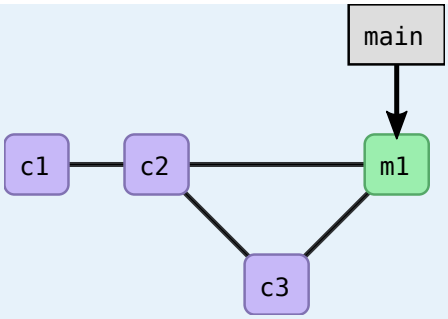


fork

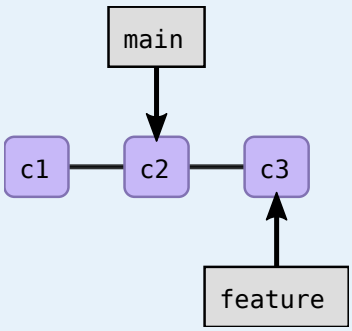


local

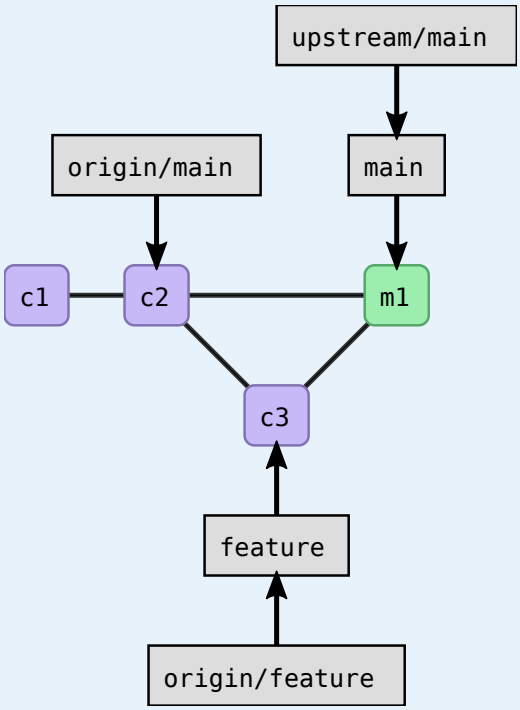
```
$ git switch main
$ git merge central/main
```



central

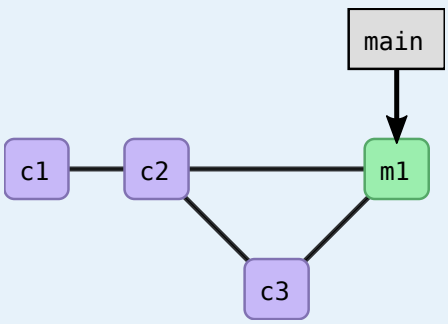


fork

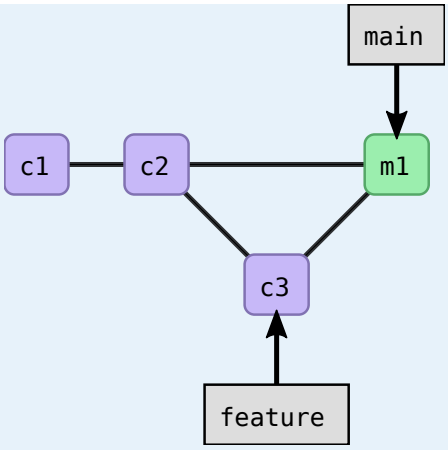


local

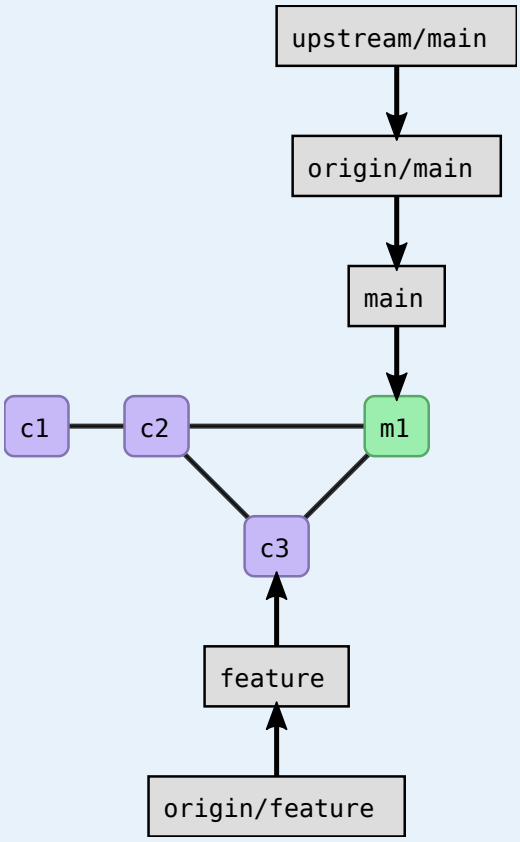
```
$ git push origin main
```



central


main
c1
c2
m1
c3
feature

fork


upstream/main
origin/main
main
c1
c2
m1
c3
feature
origin/feature

local

- Luke Skywalker: *You know, I did feel something. I could almost see the remote.*
- Ben Kenobi: *That's good. You've taken your first step into a larger world.*

[from Star Wars Episode IV - A New Hope]

💬 Discussion: Always create a feature branch

For each pull request create a new branch. Motivation:

- Limits the risk that commits get accidentally appended to an open pull request (remember: pull requests are from branch to branch, not from commit to branch).
- History-rewrite (rebased and/or squashed commits) on the central repository does not lead to a diverging branch on the fork.

How to contribute changes to somebody else's project

❗ Objectives

- Avoid frustration and surprises by first discussing and then coding.

Instructor note

- 15 min teaching/discussion

Contributing very minor changes

- Clone or fork+clone repository
- Create a branch
- Commit and push change
- Open a pull request or merge request

If you observe an issue and have an idea how to fix it

- Open an issue in the repository you wish to contribute to
- Describe the problem
- If you have a suggestion on how to fix it, describe your suggestion
- Possibly **discuss and get feedback**
- If you are working on the fix, indicate it in the issue so that others know that somebody is working on it and who is working on it
- Submit your fix as pull request or merge request which references/closes the issue

❗ Motivation

- **Inform others about an observed problem**

- Make it clear whether this issue is up for grabs or already being worked on

If you have an idea for a new feature

- Open an issue in the repository you wish to contribute to
- In the issue, write a short proposal for your suggested change or new feature
- Motivate why and how you wish to do this
- Also indicate where you are unsure and where you would like feedback
- **Discuss and get feedback before you code**
- Once you start coding, indicate that you are working on it
- Once you are done, submit your new feature as pull request or merge request which references/closes the issue/proposal

Motivation

- **Get agreement and feedback before writing 5000 lines of code** which might be rejected
- If we later wonder why something was done, we have the issue/proposal as reference and can read up on the reasoning behind a code change

WIP (work in progress) merge requests and draft pull requests

- Convention: Pull requests or merge requests starting with “WIP” or “Draft” are not to be merged yet
- They are there to **collect feedback on unfinished work**
- On GitHub you can create draft pull requests which cannot be merged until marked ready for review.
- Also GitLab offers same mechanism (merge request starting with “WIP” or “Draft”)

Motivation

- Collect feedback before it is finished and before it becomes more difficult to change
- Communicate to others what is partially done if it affects their work

Licenses matter

- If you submit code that is derivative work or code somebody else wrote, clarify license
- If you receive pull requests with a lot of code, **clarify its license and copyright** with the submitter, before merging

How to make sure that you don’t merge malicious code

(this is typically not a problem for most of us but can be a problem for some)

- Since commit hashes depend on all their parents you cannot modify the past without all future hashes changing
- Projects like https://github.com/torvalds/linux or https://github.com/Homebrew/brew have to be extremely careful what they accept
- Browse the code changes before merging them
- If you get an extremely large changeset, ask for more information
- Possibly verify whether the submitter is not trying to impersonate somebody you know
- Git commits can be PGP signed to verify authenticity

Keypoints

- Communicate and discuss before coding massive changes.
- **Cross-reference discussions, proposals, and code changes**.

Hooks

Objectives

- Learn how to couple scripts to Git repository events.

Instructor note

- 10 min teaching/demonstration

Sometimes you would like Git events (commits, pushes, etc.) to **trigger scripts** which take care of some tasks. **Hooks are scripts that are executed before/after certain Git events**. They can be used to enforce nearly any kind of policy for your project. There are client-side and server-side hooks.

Client-side hooks

You can find and edit them here:

```
$ ls -l .git/hooks/
```

- `pre-commit`: before commit message editor (example: make sure tests pass)
- `prepare-commit-msg`: before commit message editor (example: modify default messages)
- `commit-msg`: after commit message editor (example: validate commit message pattern)
- `post-commit`: after commit process (example: notification)
- `pre-rebase`: before rebase anything (example: disallow rebasing published commits)
- `post-rewrite`: run by commands that rewrite commits
- `post-checkout`: after successful `git checkout` (example: generating documentation)

- `post-merge`: after successful merge
- `pre-push`: runs during `git push` before any objects have been transferred
- `pre-auto-gc`: invoked just before the garbage collection takes place

See also https://pre-commit.com, a framework for managing and maintaining multi-language pre-commit hooks.

Example for a `pre-commit` hook which checks whether a Python code is PEP 8-compliant using pycodestyle:

```
#!/usr/bin/env bash

# ignore errors due to too long lines
pycodestyle --ignore=E501 myproject/
```

Server-side hooks

You can typically edit them through a web interface on GitHub/GitLab.

- `pre-receive`: before accepting any references
- `update`: like `pre-receive` but runs once per pushed branch
- `post-receive`: after entire process is completed
- Typical use:
  - Maintenance work
  - Automated tests
  - Refreshing of documentation/website
  - Sanity checks
  - Code style checks
  - Email notification
  - Rebuilding software packages

Actions, workflows, and continuous integration services

GitHub and GitLab let you define workflows/actions/recipes which are triggered by e.g. `git push` or by a release (tag creation). They can be customized and almost any automation you can think of becomes possible.

These services use hooks under the hood. These days, project are more likely to use these higher-level services rather than Git hooks directly.

You can read more about these services here:

- GitHub Actions

- GitLab CI

In our projects we use these services to:

- Build websites
- Build documentation
- Run tests
- Create containers
- Package and upload packages
- Spellchecking

Non-bare and bare repositories

❗ Objectives

- Understanding the difference between non-bare and bare repositories.
- Being able to create a common repository for a group on our local computer or server.

Instructor note

- 10 min teaching/demonstration

Non-bare repository

- A **non-bare repository** contains `.git/` as well as a snapshot of your tracked files that you can directly edit called **the working tree** (the actual files you can edit).
- **This is where we edit and commit changes**.
- When we create a repository with `git init`, it is a non-bare, "normal", repository.

Bare repository

- A **bare repository** contains only the `.git/` part, no files you can directly edit.
- By convention the names of bare repositories end with `.git` to emphasize this.
- We never do actual editing work inside a bare repository.
- GitHub, GitLab, etc. store a bare repository.
- You can also create a bare repository on your computer/server to store your private repository.

If we have enough time, the instructor demonstrates how to create a bare repository on the local computer:

✍ Bare-1: Create and use a bare repository

- Create a new local repository with `git init`.

```
$ cd /path/to/example
$ git init
```

- Populate it with a file and a commit or two.
- Create one or two branches.
- Clone this repository on the same computer with either `--bare` or `--mirror`:

```
$ git clone --bare /path/to/example /path/to/example-bare
```

- Inspect the bare repository.
- Clone the bare repository:

```
$ git clone /path/to/example-bare /path/to/example-clone
$ cd /path/to/example-clone
```

- Inside the clone inspect `git remote -v`.
- Inside the clone create a commit and push the commit to `origin`.
- The bare repository can be cloned several times and one can exercise pushing and pulling changes.

Keypoints

- We do programming work inside non-bare repositories.
- We can create a local common bare repository where we can push to and pull from.

Quick reference

Other cheatsheets

See the git-intro cheatsheet for the basics.

- Interactive git cheatsheet
- Very detailed 2-page git cheatsheet

Glossary

- **remote**: Roughly, another git repository on another computer. A repository can be linked to several other remotes.
- **push**: Send a branch from your current repository to another repository
- **fetch**: Update your view of another repository
- **pull**: Fetch (above) and then merge

- **origin**: Default name for a remote repository.
- **origin/NAME**: A branch name which represents a remote branch.
- **main**: Default name for main branch.
- **merge**: Combine the changes on two branches.
- **conflict**: When a merge has changes that affect the same lines, git can not automatically figure out what to do. It presents the conflict to the user to resolve.
- **issue**: Feature of web repositories that allows discussion related to a repository.
- **pull request**: A GitHub/Gitlab feature that allows you to send a code suggestion using a branch, which allows one-button merging. In Gitlab, called "**merge request**".
- **git hook**: Code that can run before or after certain actions, for example to do tests before allowing you to commit.
- **bare repository**: A copy of a repository that only is only the `.git` directory: there are no files actually checked out. Directory names usually like `something.git`

Commands we use

This excludes most introduced in the git-intro cheatsheet.

Setup:

- `git clone URL [TARGET-DIRECTORY]`: Make a copy of existing repository at <url>, containing all history.

Status:

- `git status`: Same as in basic git, list status
- `git remote [-v]`: List all remotes
- `git graph`: see a detailed graph of commits. Create this command with `git config --global alias.graph "log --all --graph --decorate --oneline"`

General work:

- `git switch BRANCH-NAME`: Make a branch active.
- `git push [REMOTE-NAME] [BRANCH:BRANCH]`: Send commits and update the branch on the remote.
- `git pull [REMOTE-NAME] [BRANCH-NAME]`: Fetch and then merge automatically. Can be convenient, but to be careful you can fetch and merge separately.
- `git fetch [REMOTE-NAME]`: Get commits from the remote. Doesn't update local branches, but updates the remote tracking branches (like origin/NAME).
- `git merge [BRANCH-NAME]`: Updates your current branch with changes from another branch. By default, merges to the branch is is tracking by default.
- `git remote add REMOTE-NAME URL`: Adds a new remote with a certain name.

Instructor guide

Schedule Day 3

- 08:50 - 09:00: Soft start and icebreaker question
- 09:00 - 09:15: Recap Git, any HedgeDoc questions to highlight
- 09:15 - 09:30: Concepts around collaboration
  - Explain terms: Pull, push, clone, fork. Focus on pull and not fetch.
  - Focus more on clone and less on generating from templates and importing.
- 09:30 - 11:00: Centralized workflow
  - 9:30 - 9:45: Explain concepts
  - 9:45 - 9:55: Break
  - 9:55 - 10:00: Inform clearly what is expected outcome
  - 10:00 - 10:30: Exercise
  - 10:30 - 11:00: Instructors go through the exercise. Discussion and answering questions
- 11:00 - 12:00: Lunch Break
- 12:00 - 13:10: Distributed version control and forking workflow
  - 12:00 - 12:15: Concepts and what are exercise outcomes
  - 12:15 - 12:45: Exercise
  - 12:45 - 12:55 Break
  - 12:55 - 13:10: Instructors go through excercises. Discussion and answering questions
- 13:10 - 13:30: How to contribute changes to somebody else's project and Q&A

Why we teach this lesson

In order to collaborate efficiently using Git, it's essential to have a solid understanding of how remotes work, and how to contribute changes through pull requests or merge requests. The git-intro lesson teaches participants how to work efficiently with Git when there is only one developer (more precisely: how to work when there are no remote Git repositories yet in the picture). This lesson dives into the collaborative aspects of Git and focuses on the possible collaborative workflows enabled by web-based repository hosting platforms like GitHub.

This lesson is meant to directly benefit workshop participants who have prior experience with Git, enabling them to put collaborative workflows involving code review directly into practice when they return to their normal work. For novice Git users (who may have learned a lot in the git-intro lesson) this lesson is somewhat challenging, but the lesson aims to introduce them to the concepts and give them confidence to start using these workflows later when they have gained some further experience in working with Git.

Intended learning outcomes

By the end of this lesson, learners should:

- Understand the concept of remotes
- Be able to describe the difference between local and remote branches
- Be able to describe the difference between centralized and forking workflows
- Know how to use pull requests or merge requests to submit changes to another projects
- Know how to reference issues in commits or pull/merge requests and how to auto-close issues
- Know how to update a fork
- **Be able to contribute in code review as submitter or reviewer**

Interesting questions you might get

- If participants run `git graph` they might notice `origin/HEAD`. This has been omitted from the figures to not overload the presentation. This pointer represents the default branch of the remote repository.

Timing

- The centralized collaboration episode is densest and introduces many new concepts, so at least an hour is required for it.
- The forking-workflow exercise repeats familiar concepts (only introduces forking and distributed workflows), and it takes maybe half the time of the first episode.
- The "How to contribute changes to somebody else's project" episode can be covered relatively quickly and offers room for discussion if you have time left. However, this should not be skipped as this is perhaps the key learning outcome.

Preparing exercises

Exercise leads typically prepare exercise repositories for the exercise group (although the material speaks about "maintainer" who can also be one of the learners). Preparing the first exercise (centralized workflow) will take more time than preparing the second (forking workflow). Most preparation time is not the generating part but will go into communicating the URL to the exercise group, communicating their usernames, adding them as collaborators, and waiting until everybody accepts the GitHub invitation to join the newly created exercise repository.

Live stream:

- Create the centralized exercises **in an organization** (not under your username) so that you can give others admin access to add collaborators. Also this way you can then fork yourself if needed.
- For CR workshops, the exercises were placed under https://github.com/cr-workshop-exercises.
- We have created two versions of each **a day in advance** to signal which one might end up being discussed on recording/stream:

    - `centralized-workflow-exercise-recorded`
    - `centralized-workflow-exercise`
    - `forking-workflow-exercise-recorded`
    - `forking-workflow-exercise`

- Protect the default branch of the two `centralized-*` repositories.
- We create a organization team, `stream-exercise-participants`. The *centralized* workflow exercise repos have this team added as a collaborator (*not* forking - they fork so they don't need write access there).
- We have collected usernames of people who want to contribute via issues on GitHub. Make a fifth repository, `access-requests`, create a sample access request issue there, and have learners make a new issue in that repository. The day before

    - Why a fifth repository? So that learners don't get emails from all other access requests once they get added to the team
    - Sample text:

```
On Thursday we will all practice how to collaborate using Git/GitHub
and one ambitious thing we will try is to collaborate with
participants following via stream. This does not apply to teams and
exercise groups who will create their own exercise repositories and
these groups can ignore the rest of this section.

For this to work we will need to give you access to a practice
repository.  This is option, you could just watch instead. We delete
these repositories after the workshop and will not use your username
for anything other than this exercise.

If you would like to participate in this, could you please open an
issue here.  Give any title like "please add me" and then click
submit:
- https://github.com/cr-workshop-exercises/access-requests/issues/new

Example for how I requested access:
- https://github.com/cr-workshop-exercises/access-requests/issues/1
```

Typical pitfalls

Difference between pull and pull requests

The difference between pull and pull requests can be confusing, explain clearly that pull requests or merge requests are a different mechanism specific to GitHub, GitLab, etc.

Pull requests are from branch to branch, not from commit to branch

The behavior that additional commits to a branch from which a pull request has been created get appended to the pull request needs to be explained.

Other practical aspects

- In in-person workshops participants really have to sit next to someone, so that they can see the screens. From the beginning.
- Emphasize use of `git graph` a lot, just like in the git-solo lesson.