Git & GitHub

(for the R user)

Francesca Vitalini

2021-09-22 14:00-17:00 CEST

Mirai Solutions

A Few Words About the Instructor

Francesca Vitalini

- Senior Solutions Consultant @ Mirai Solutions
- Physicist by training, PhD Computational Biophysics
- Experience in multiple programming languages
- Involved in open source, outreach and community activities
- more than 5 years experience working with R and version control

- Independent Zurich-based software development and consultancy firm
- Interdisciplinary team of experienced scientists, software engineers and IT architects with competencies from finance, risk management and actuarial knowledge to math, stats, modeling / simulation and machine learning
- More than 10 years experience in bringing smart and modern solutions to the industry, using cutting edge technology together with best practices and leveraging on open source technology
- Customer-focused, service-oriented and results-driven agile approach

Bringing ideas to life.

Smart. Agile. Personal.

Workshop Structure

- Version Control: why?
- Hands-on using command line, R, and RStudio: concepts are generic and programming language independent
- Example cases:
  - collaborative work: branches, forks and remotes
  - solving (merge) conflicts
  - can I time travel?
  - oh damn I have committed the wrong thing! Let's cherrypick.
- GitFlow: an Agile development approach
- Most common Git Commands / Git cheatsheet

Today's development:

- toy Rmarkdown example and integration with RStudio
- Git + GitHub
- **Ask questions!**

Workshop prerequisites

Pre-requisites for the hands-on part:

- A recent version of RStudio (>= 1.2 is recommended)
- R installation: R >= 3.6.x, ideally R 4.1.x
- GitHub account
- Successful installation of Git and smooth integration with RStudio and GitHub.

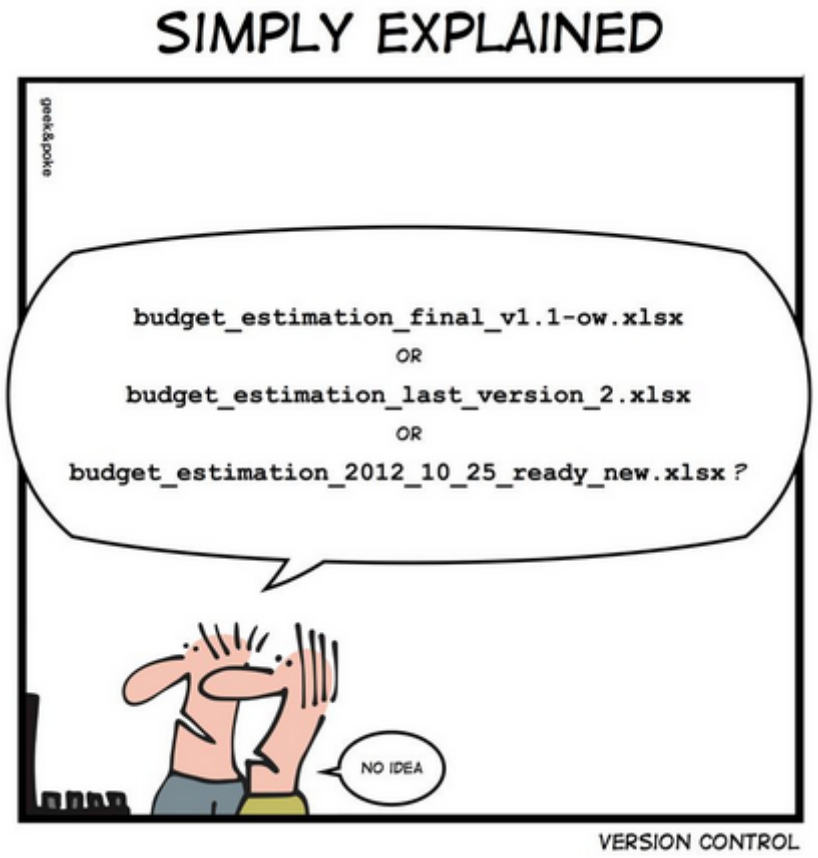
Optional Git-related tools:

- compareWith, an R package developed by Mirai Solutions, providing user-friendly RStudio addins based on Meld that make it easier to check differences with remote (and perform merge tasks) prior to pulling.

Why Version Control?

Version Control is a software development workflow tool to:

- keep a copy of your code in a safe and secure external shared location
- keep track of changes on the source code,
- easily revert errors and reduce project risk
- facilitate parallel development, both by:
  - easing collaboration and
  - preserving productive code whilst new development.



Geek&Poke

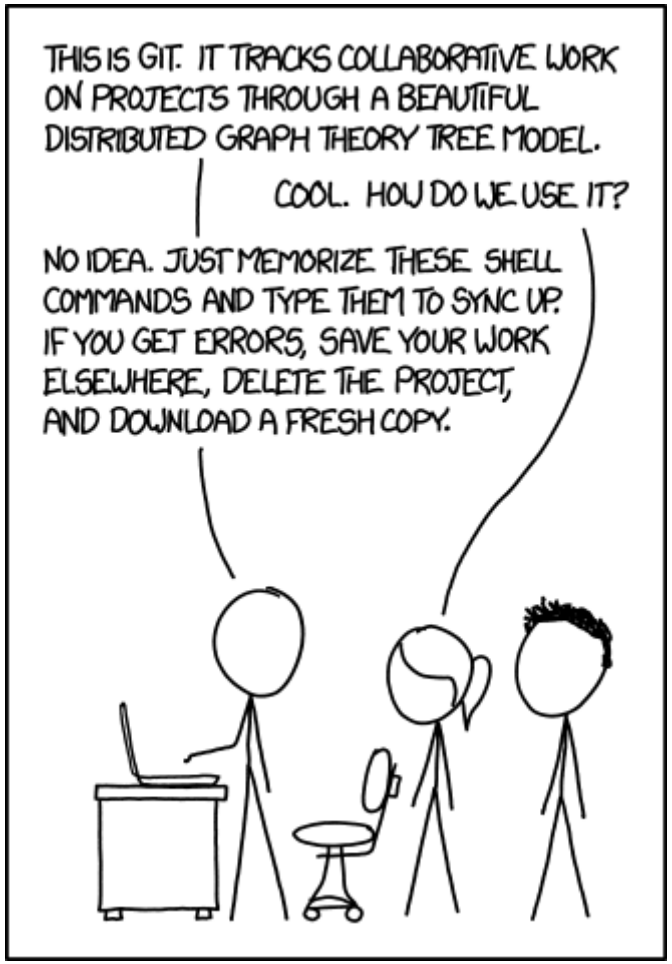
Benefits of Version Control

Why Version Control:

- provide backup of the source code;
- make room for safe testing without affecting the master code;
- keep history of how a piece of software came to be, and allow rollback to a previous version, including restoring of deleted files;
- enable collaborators to work on the same source files without affecting each other's work and simplify the integration of their development versions.

Without Version Control:

- "I don't know what I did but now nothing works anymore!"
- `my_code_v1`, `my_code_new`, `my_code_newer`, `my_code_latest`...
- "Ok Mario, I am done with my changes, now you can take the file!"
- "Tom, can you tell me what you changed so that I can integrate it in my file?"



xkcd

Why Git?

Git is a popular, free and open source version control system designed to be very efficient and to support distributed, non-linear workflows.

It helps collaborate, track changes and ease the coordination and communication with others.

Hosting services, like GitHub, Bitbucket, and GitLab provide a storing space for Git-based projects on the internet and an interface for visualizing the material, syncing it, communicate with others and manage the project.

It makes sense to have a remote host for your Git-managed project because it:

- keeps a copy safely away from your local machine
- maintains a copy of your work even if you mess up locally
- allows you to access your work from multiple machines
- allows others to access your work
- simplifies the integration of multiple people's work into a common source

Why GitHub

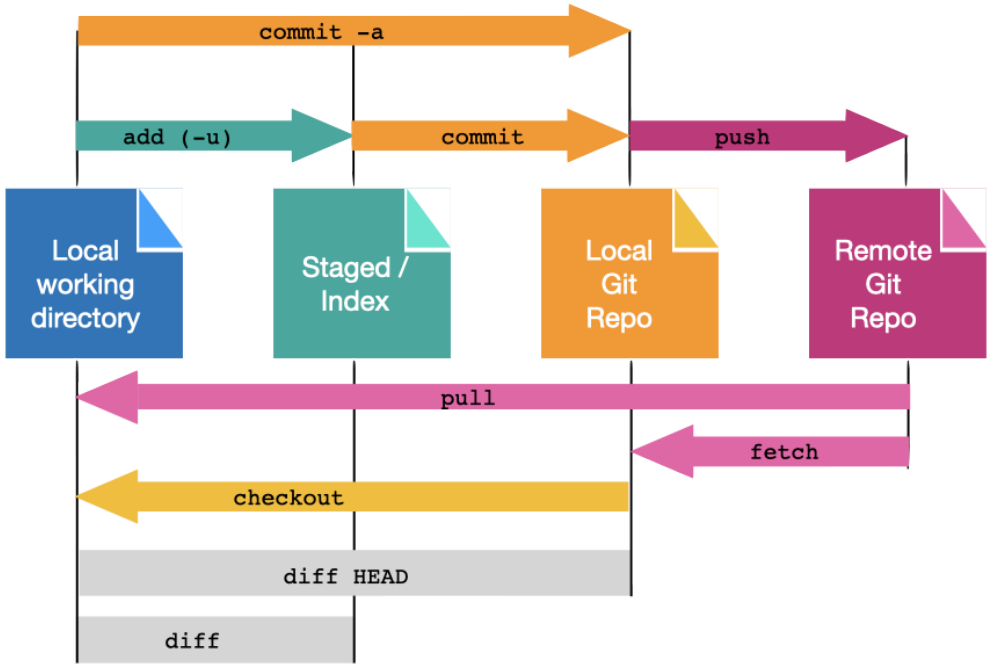
GitHub remote hosting, is the *de facto* standard for open source (R and many other languages) projects.

In addition to a well-designed user interface, GitHub offers two especially important features:

- Issues to track changes (bugs, new features, to-dos, impediments, etc.), which can be organized into Project Boards for easier management. By being referenced in a commit, in a branch or in a pull request, issues allow to keep a record of what was discussed directly in the history of the source code.
- Pull requests to visualize and preserve the discussion about the integration of independent development branches into the main branch.

Common Git commands

- Pull (`git pull`): Incorporate changes from a remote repository into the current local branch.
- Stage changes (`git add`): Prepare the content for the next commit.
- Commit (`git commit`): Record staged changes to the current branch.
- Push (`git push`): Update remote repository using the local branch.



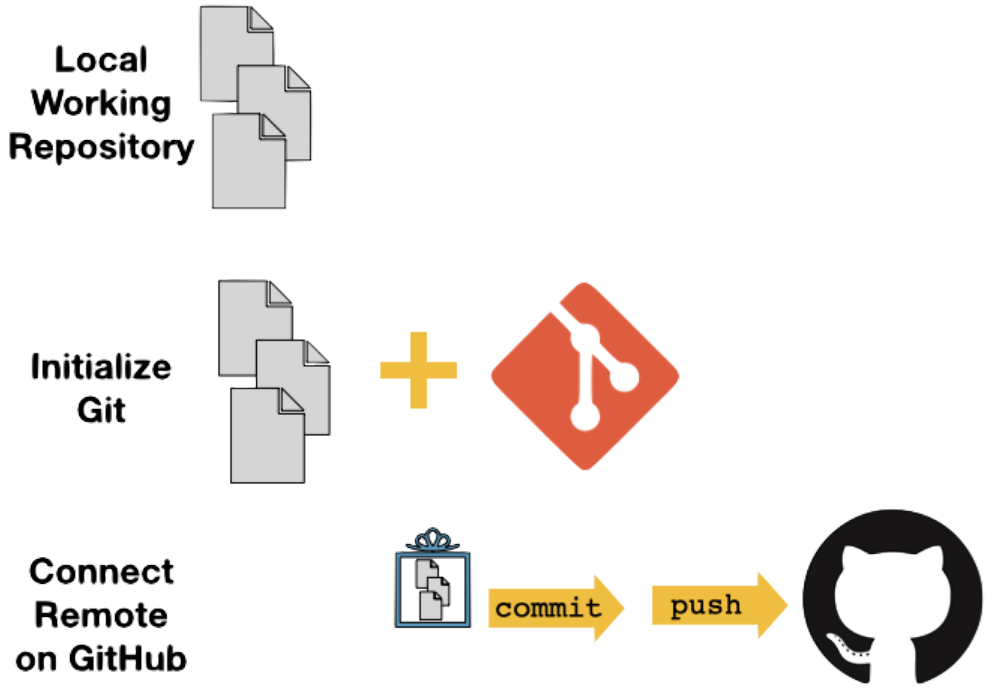
Add local repo to Git and GitHub

We will learn how to:

- initialize Git in an existing local working repo
- connect it with a remote on GitHub

Using:

- Command Line
- R Console
- RStudio



Add local repo to Git and GitHub - Command line

Create local repo

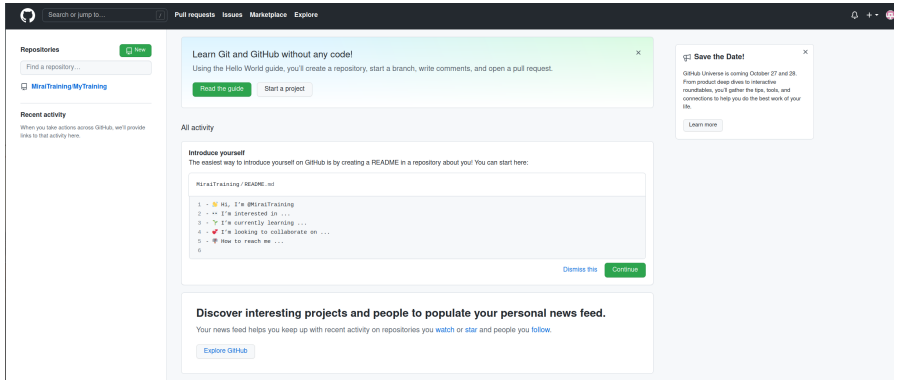
- create a GitHub repo (no initialization of files needed) and get its `<GITHUB_URL>`
- follow GitHub suggestions:
  - in the in directory, to initialize a Git repo, run `git init`

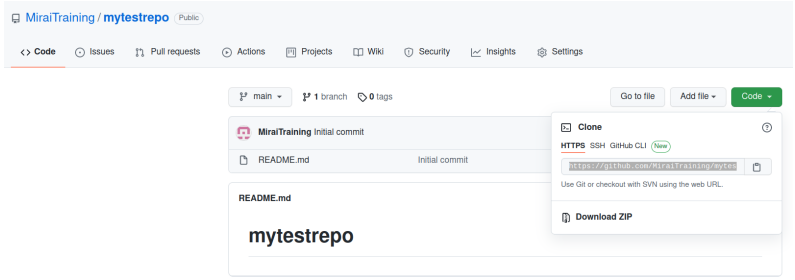
Setting Branches

- create a new main branch `git checkout -b main`
- run `git remote add origin <GITHUB_URL>` with the repo URL as `<GITUB_URL>`;

Connect GitHub

- stage new files `git add .` and commit them `git commit`
- `git push -u origin main`





Add local repo to Git and GitHub - RStudio

Initialization

As a new Project

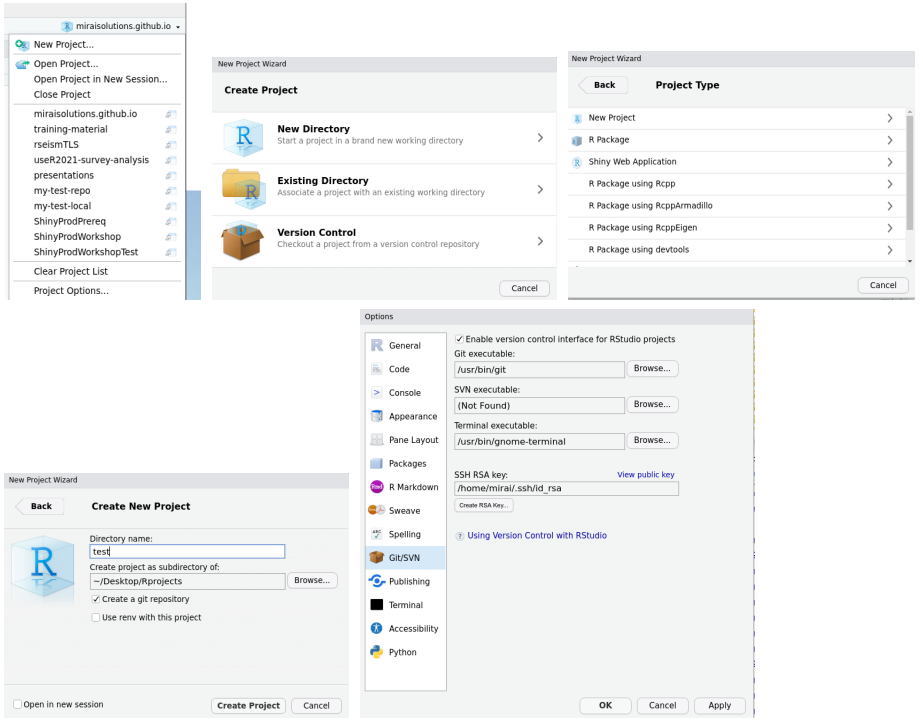
`File > New Project > New Directory > New Project`

select the *create a Git repository* option to activate Git tracking and Pane.

On an existing project:

`Tools > Project Options... > Git/SVN`

select **Git** under "Version control system". Then confirm the new Git repository in the dialog box.



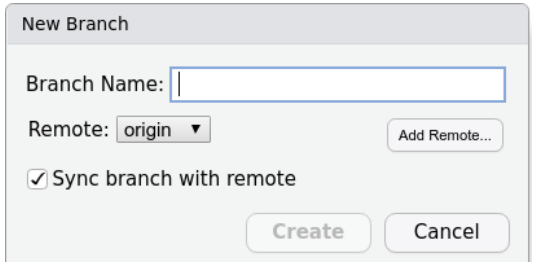
Add local repo to Git and GitHub - RStudio

Setting Branches Add a remote GitHub repo as `origin`:

- click on the icon "New branch"  on the Git Pane and name it "main".

Connect GitHub

- click "Add remote" and paste the GitHub URL; use "origin" as remote name; click "Add". Make sure the "sync branch with remote" option is checked.



- this creates a new "main" branch and switches to it.

Add local repo to Git and GitHub - with usethis

usethis is a workflow package by RStudio, to automate development tasks.

Initialization

- call `usethis::use_git()` to initialize the repository with a Git command - no connection to GitHub required. No branches are created and no remote is set. Restart RStudio to activate the Pane.

Connect GitHub

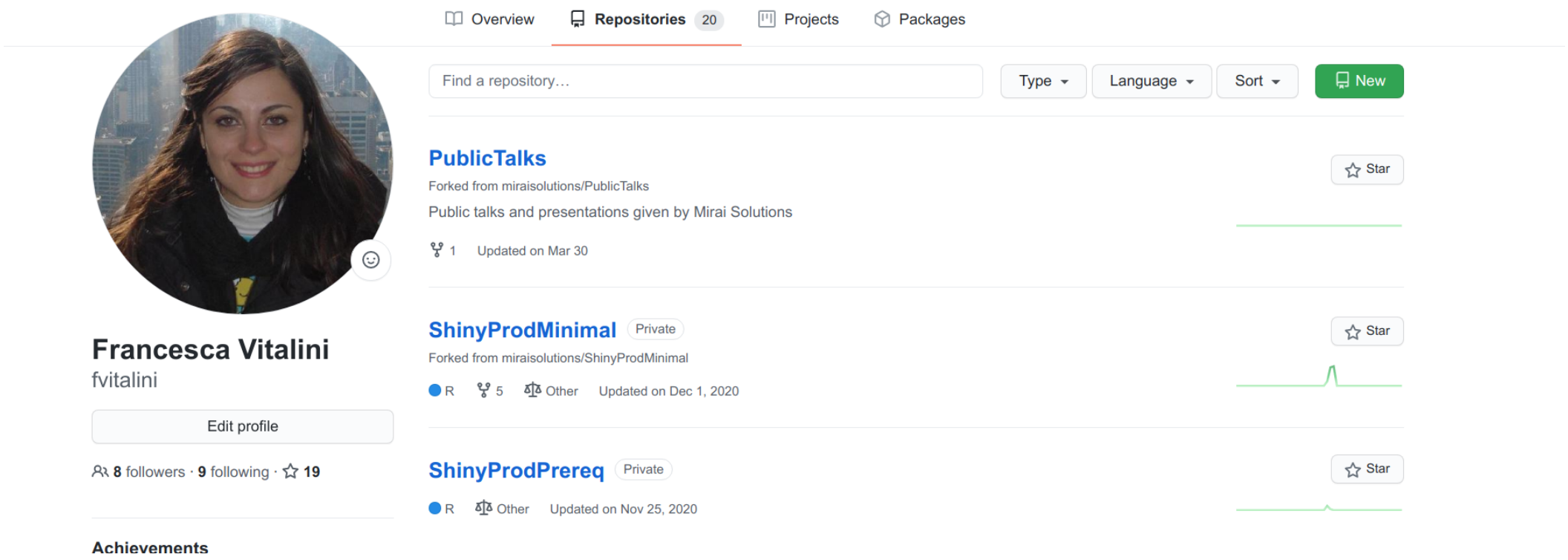
- run `usethis::use_github()` to create a repository on GitHub, set it as remote and perform the first commit.

Exercise: add Git & GitHub to your R Project

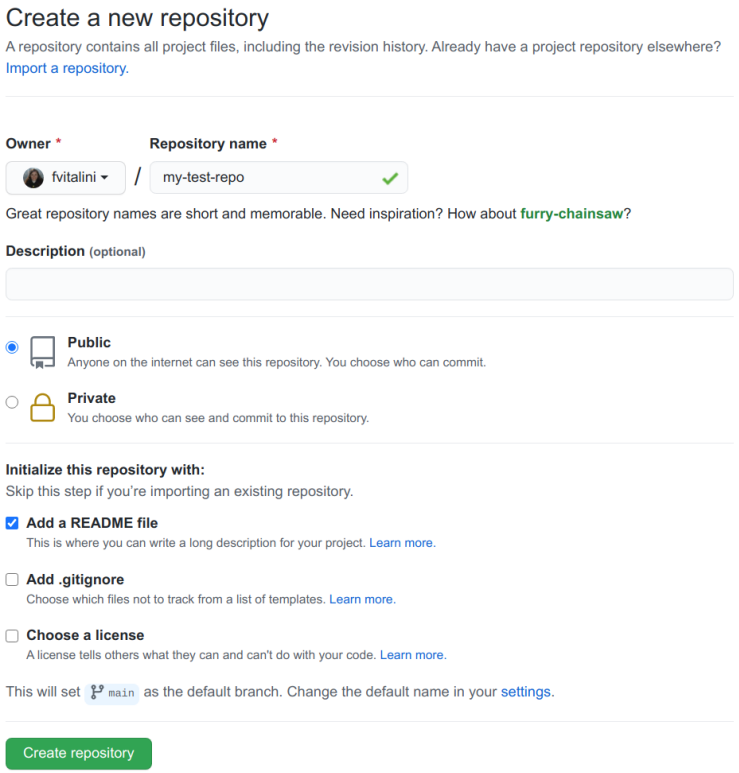
Following what we just did, add Git to your repository and connect it to GitHub.

Create a new (non-empty) repo on GitHub

On GitHub you can initialize a new repo by clicking on the "New" green button (you have to be logged in).

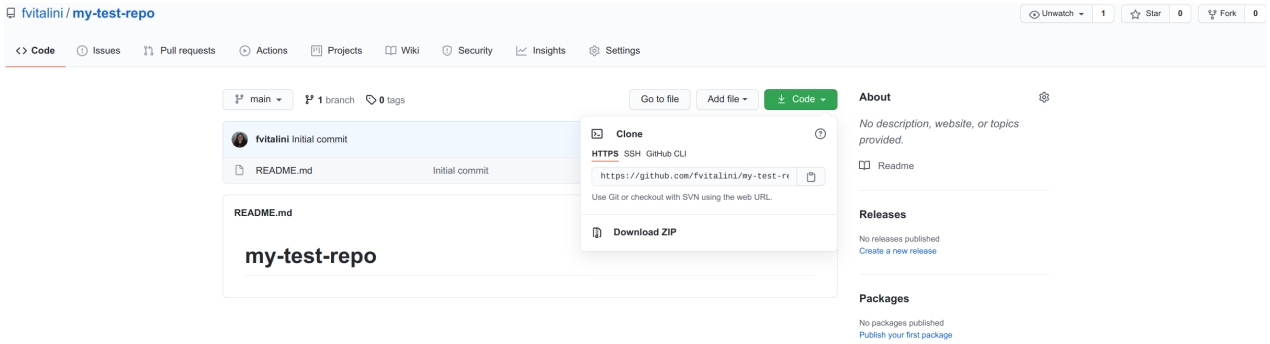


New GitHub repo settings



- give the new repo a name.
- you may describe what it does (e.g. "test how to use Git and GitHub")
- "Public" means anyone will be able to see the content. "Private" means only people with rights will be able to access it.
- initializing the repo with `README.md` will create a file in the repo.
- note that the repo will be initialized with a `main` branch (used to be called `master`)

Get remote repo locally



Command line:

- `git clone` + repo URL
- `git remote show origin` check the remote and branches
- `git remote -v` list the remote URL

In RStudio:

- `File > New Project... > Version Control > Git`
- Paste the GitHub repo URL in the the pop-up pane.

with `usethis`:

- `usethis::create_from_github("OWNER/REPO", fork = FALSE)` creates a local copy of an existing GitHub repo.

Exercise: check out your GitHub Project locally

- Create a Project on GitHub as shown in the lecture and check it out locally

Local Change

Open the `README.md` in RStudio or in a text editor and make some changes in the text.

(Command Line `echo "My local change" >> README.md`)

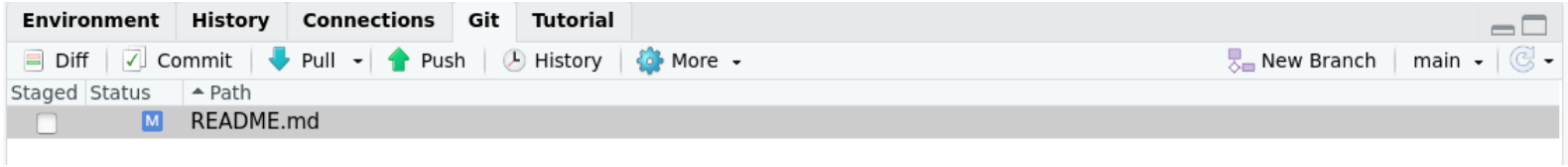
Check that the status of the local file is modified:

Command Line:

`git status` > which files have been modified `git diff` > shows what has changed in a file

RStudio:

- icons in the Git Pane



Commit & Push local changes

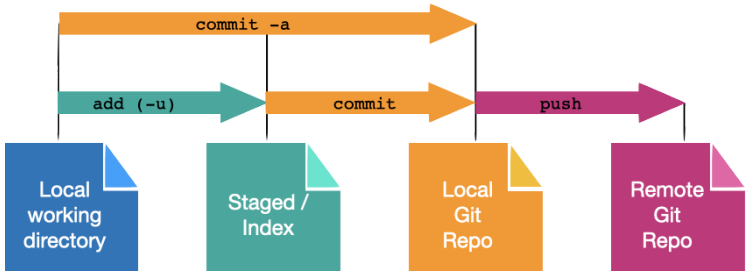
Stage the changed file to prepare it for the commit, commit to add the change to your local history, and push to make the change available on the remote. You'll need your GitHub Credentials.

Command line:

- `git add -A`
- `git commit -m "My first commit from my local computer"`
- `git push`

RStudio:

- click "Diff" in the Git Pane
- stage the file by ticking the box in the pop-up (after reviewing the diff)
- provide a message in the pop-up
- click "Commit" in the pop-up
- click "Push" in the pop-up to push changes to the remote



After pushing your changes, they should be available on the Git remote hosting service, e.g. GitHub.

Hints: Notes on Commit Messages

The commit message should be **informative** and provide a **meaningful history** of the changes (also when blaming a file) and the motivation behind them.

Commit Messages Guidelines

A good practice is to use a compact first line summarizing what the commit is about, and providing more details as new lines / bullets after leaving an empty line. This structure is interpreted by Git tools and hosting services like GitHub.

It is also good practice to use imperative for the summary.

How to write meaningful commits is something you learn by experience. For some bad commit messages examples have a look here.

Commit messages are mandatory. In the command line, if you forget the `-m`, Git will prompt you to enter one.

Hints: Notes on Commit & Push Strategies

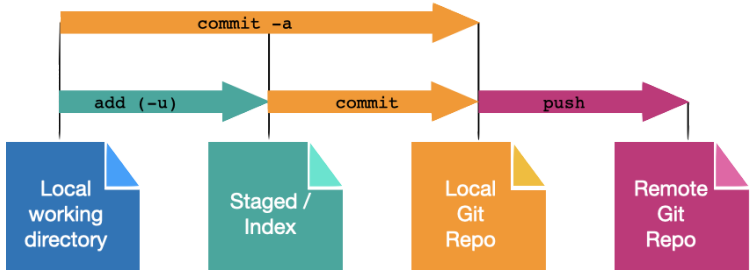
A commit should include all the modified files relevant for one change. Better to be granular to isolate changes, but remember that too many commits may also be confusing.

Moreover, committing and pushing often makes each integration smaller, easier to control, and safer (i.e. less error prone).

If you have created a new project in RStudio, a `.gitignore` and an `*.RProj` file will be created for you. Those files should be committed to ensure the same behavior of the project in all local copies.

Exercise: Commit changes to a file

- Modify the README file locally and push your changes to version control



05:00

Add a New File

Create a new markdown file in RStudio:

```
File > New File > Rmarkdown File
```

Save the file with a name in your current working directory.

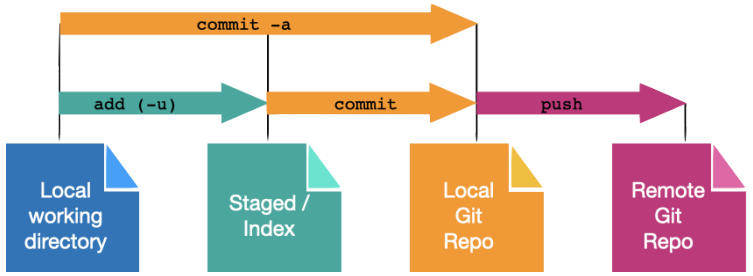
Add the new file to version control

In RStudio

- by clicking on the box next to the file name in the Git Pane
- by clicking Commit in the Git dropdown menu to open the pop-up

Via command line

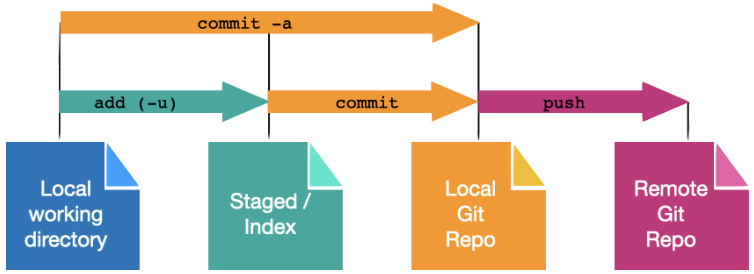
```
git add <filename>
```



Exercise: Add a Few File

- Create a new markdown file locally and add it to version control

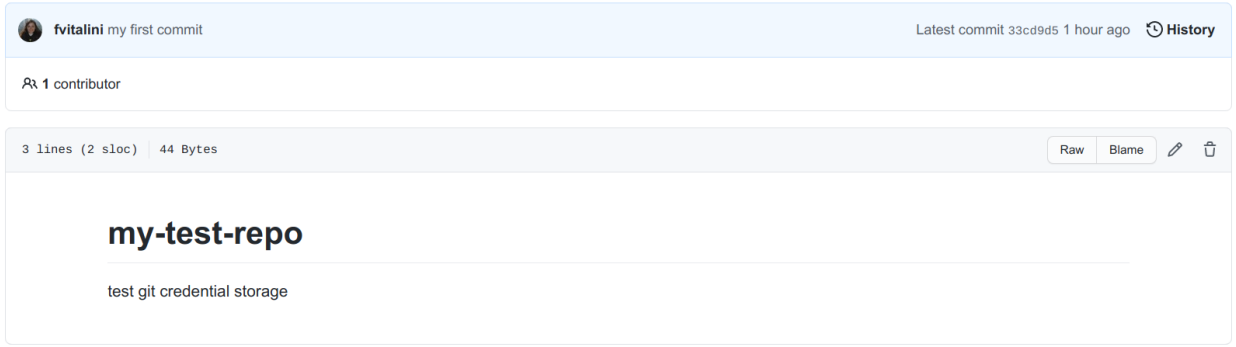
Reason for markdown format is that they are automatically rendered in GitHub



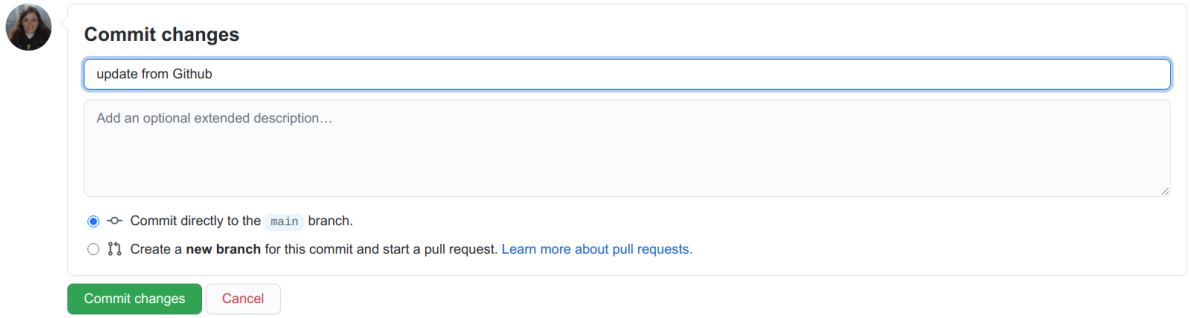
05:00

Make a change on GitHub

Open the `README.md`. Click on the pencil icon on the top right corner of the file to edit.



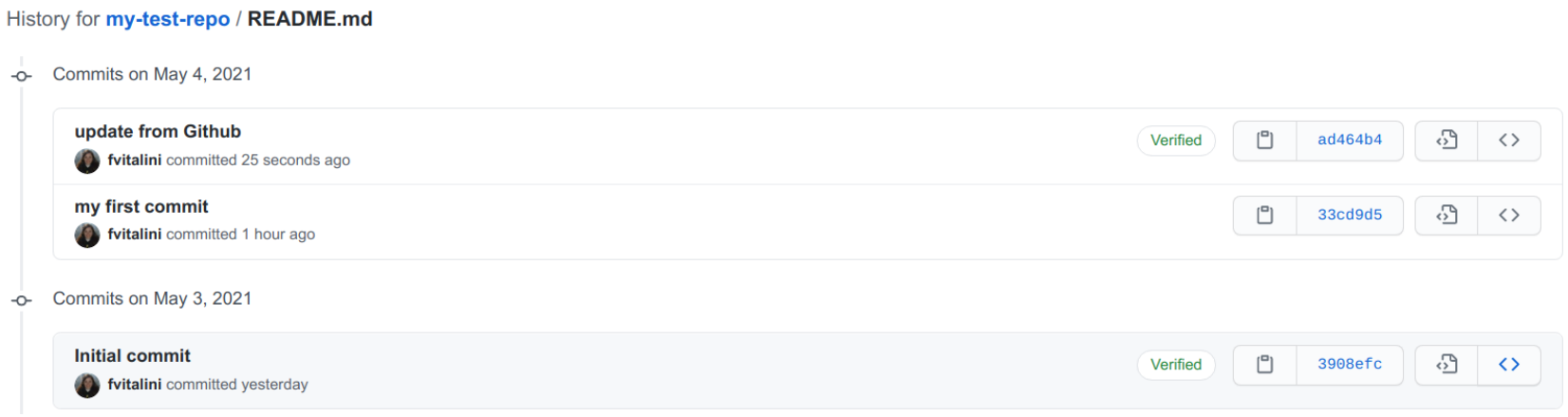
Add a text line, then commit your changes



View Commit History on GitHub

The browser will show you the updated file.

Clicking on "History" on the top right corner of the file you can visualize the past commits affecting the file.

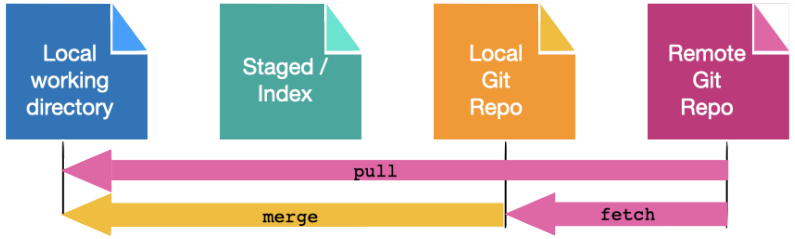


Get Remote Changes Locally

To get the changes that have been pushed to your remote (either pushed from a difference machine or by doing a commit on the remote host directly), `pull` them into your local repository.

Command Line

```
git pull
```



RStudio

via the pull button in the Git pane.

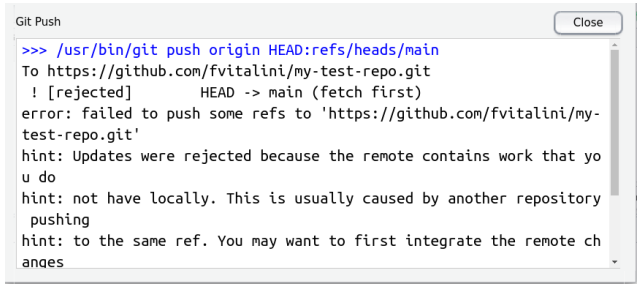
After pulling your local repo will be synchronized with the remote repo.

Hint: use compareWith to check merge differences and solve conflicts.

Exercise: Integrate changes

- Modify your Markdown file on GitHub (**add a line of text**). Commit + Push.
- Modify the same Markdown file locally (**add a different line**). Commit.

When trying to push Git will notice that there are some changes that you do not have and will report an exception.



To fix this, pull (`git pull` or pull button in the RStudio Git pane) the changes from the remote host. As this is a simple case, Git will be able to integrate the changes without any conflict, i.e. a *fast-forward merge* has occurred.

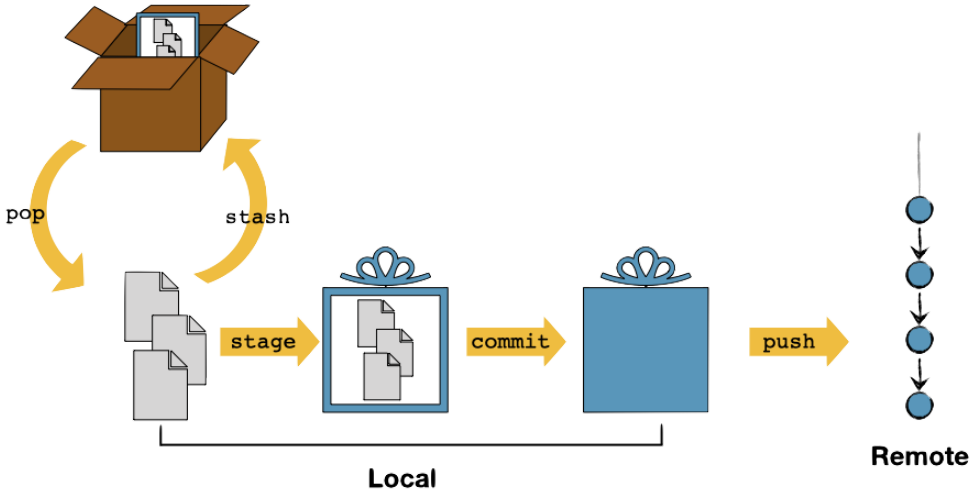
What would have happened if you had pulled before committing?

Integrating local changes without committing

If the local working directory is not in a clean state, i.e. there is uncommitted work, Git will prevent you from pulling from the remote, as the action would overwrite untracked work.

You can:

- if you are ready to commit:
  - `commit` your local changes prior to pulling to the current branch
  - pull your changes (potentially using `pull --rebase` to avoid merge commits in the history, check the appendix for more info)
- if you are not ready to commit:
  - park your changes with `stash` prior to pulling
  - reapply your changes with `stash pop`

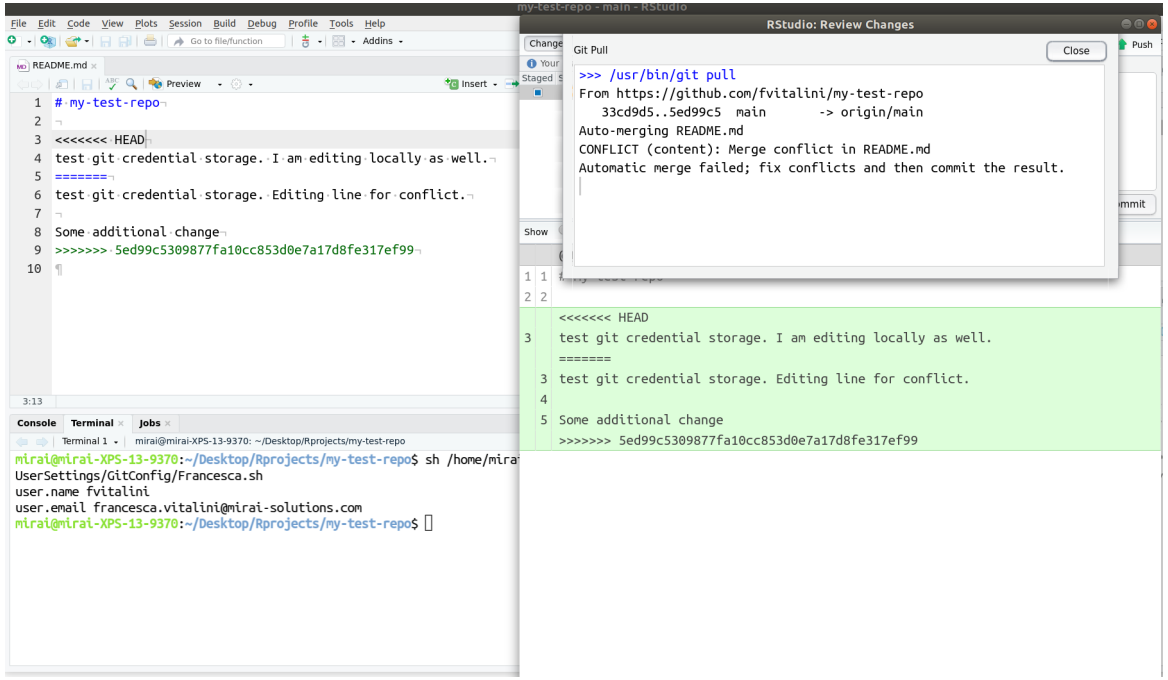


Integrate changes with conflict

Go to GitHub and modify one line in the README.

Go to your local repo and modify the same line **without** pulling changes from remote first. Commit and push.

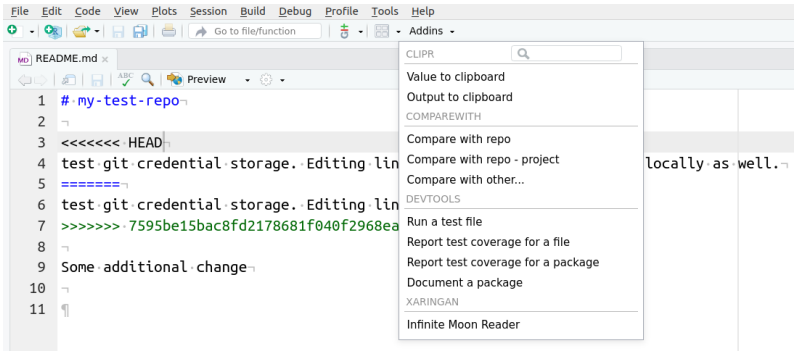
Pushing will detect a conflict.

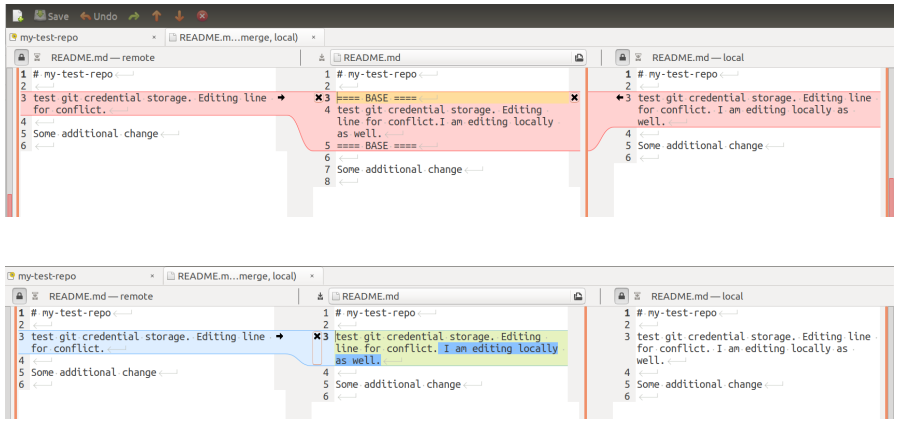


Integrate changes with conflict

Check files in conflict via `git status` or as identified in the Git pane in RStudio.

Conflicts can be resolved manually, e.g. by editing the identified conflict section, or through some merge tool, like the RStudio addin compareWith.





You can edit the file and resolve conflicts directly in Meld.

Once the conflict has been resolved, stage, commit and push the changes to solve the merge conflict.

Hint: pull often and use branches to avoid merging conflicts!

Do it yourself!

Go to GitHub and modify one line in the README.

Go to your local repo and modify the same line **without** pulling changes from remote first. Commit and push.

Solve the conflict.

Investigate the past: Commit history

You can access the history of commits

- **on GitHub** (under "commits" at repo level or under "history" at file level):

Click on the hyperlink associated with the commit SHA to view the diff.

`git blame`, or the "Blame" tab in GitHub, shows by line: who last touched it, when, and the associated commit message.

- **Locally, command line**:

Browse the `git log`.

- **Locally, RStudio**:

History button on the Git Pane or by file using the Git drop-down menu.

I need a Time Machine!

What if I committed something and then discovered this messed it all up? Can I go back in time? **Yes!**

`-- A -- B -- C` --> `-- A -- B -- C -- B'`

- You can **revert** a past (pushed) commit with `git revert <SHA>`. It creates a new commit undoing whatever change had been committed. If you want to undo what you did, let's say, the last commit, use `git revert HEAD`. To undo what you did four commits ago, you can use `git revert HEAD~3`. `git revert HEAD~3..HEAD` undoes all four commits.

`-- A -- B -- C` --> `-- A -- B`

- You can **reset** to a past commit to return to a previous status of your code. E.g. if you want to go back to the status before the, let's say, last three commits, you can use `git reset HEAD~3`.
- To change a bad commit message or the content of the last commit, you can use `git commit --amend`.

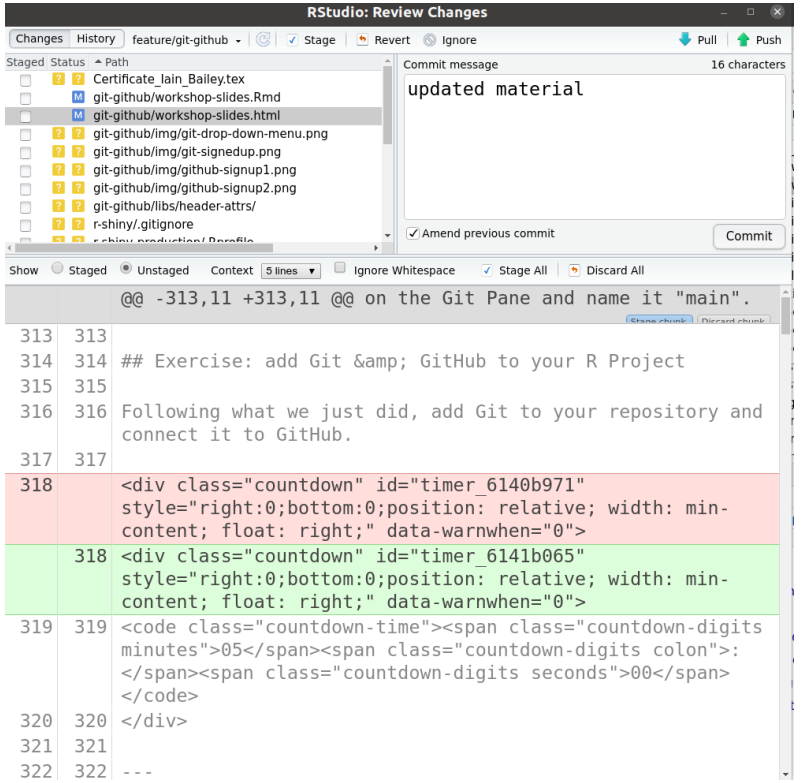
Notice: if the commit(s) being amended or reset were pushed, the action will disrupt the history for your collaborators (it will require force push). Use with care.

Some more about undoing things in Git can be found here.

Amend a commit in RStudio

You can amend the last commit directly in RStudio.

Click on Commit in the Git Pane and pick the "Amend previous commit" checkbox.



Exercise: be a Timelord

- Do a commit with a small change and a dummy message (do not push).
- Amend commit message and content
- Push changes
- Go to GitHub and check the <HSA>
- Revert the last commit

Exercise: What happened to my Git?

Given the current commit history, what happened & what should I do?

On the remote:

```
... -- A -- B -- C -- D -- E -- F
```

But I want:

```
... -- A -- B -- C -- D -- E -- F
```

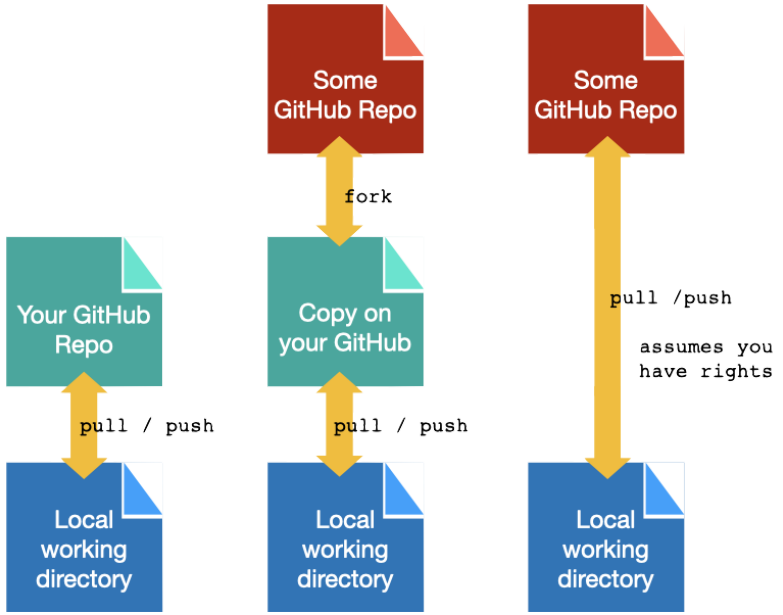
In my local repo:

```
... -- A -- B -- C -- X -- Y -- Z
```

Remotes

Your `origin` can be:

- a repo on GitHub under your account
- a repo on someone else's Github where you have appropriate rights or have been added as a collaborator (otherwise you'll have read-only permissions)
- a **fork** of someone else's GitHub repo, i.e. a copy under your GitHub account that can be managed independently of the parent repository



Fork

Take over on your repository someone else's work by creating a fork...

...in GitHub

click on the Fork button to create a copy under your account.

Then follow the same steps as to check out a remote repo locally using the Fork URL (`git remote clone <FORK_URL>`).

Additionally `git remote add upstream <SOURCE_URL>` is needed, to track changes in the source repo.

...in R

use

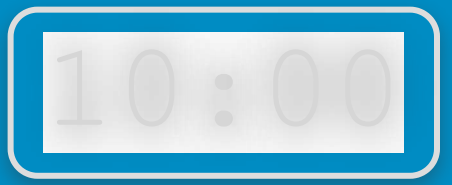
```
usethis::create_from_github("OWNER/REPO", fork = TRUE)
```

To integrate the changes of a Fork back into the main Source, a Pull Request is needed.

Exercise: Fork my repo

- go to https://github.com/fvitalini/my-test-repo
- fork the Repo
- make a local copy of your fork
- update the README with "I <YOUR NAME> have attended the workshop 'Git & GitHub' on <CURRENT DATE>"
- commit & push to your fork

Break



GitFlow

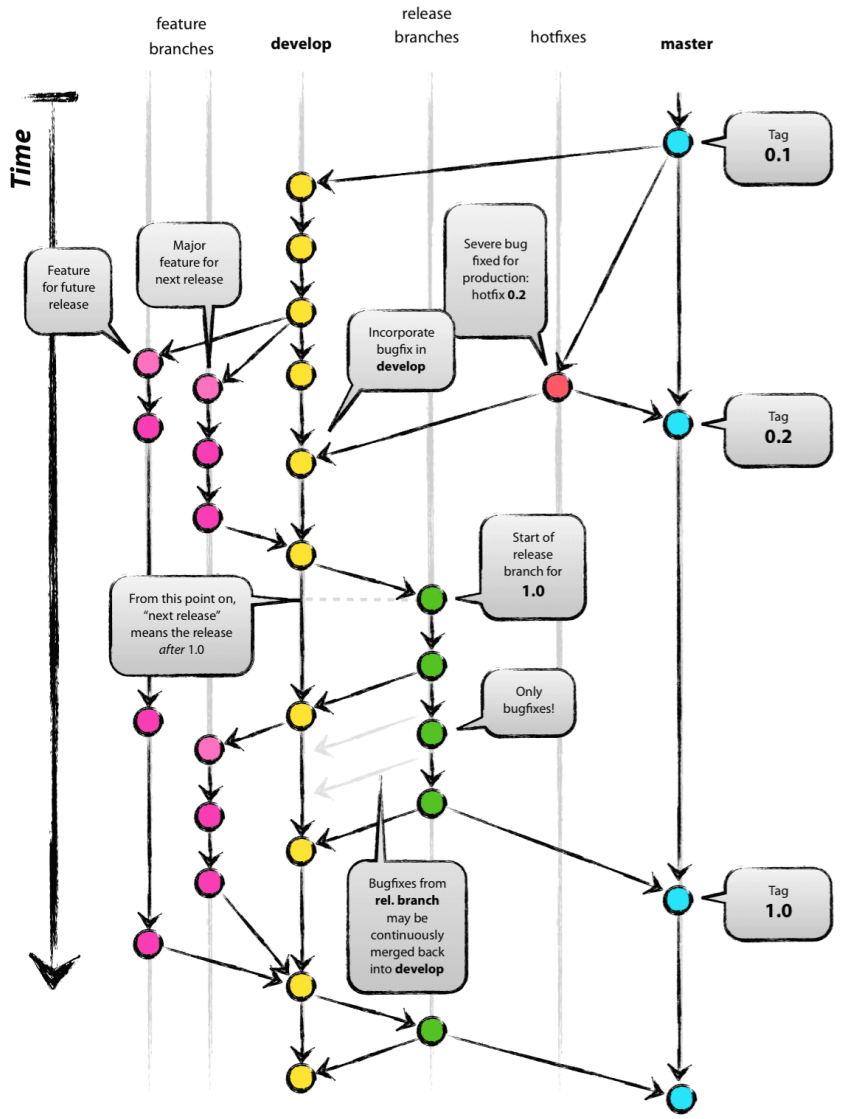
GitFlow is a popular and successful branching model that enables collaborative, controlled development and releases, assuring the safety and quality of the released product.

According to the GitFlow approach, the central repo holds two main branches with an infinite lifetime:

- `main` (or `master`): reflects a production-ready state
- `develop`: reflects a state with the latest delivered developments for the next release

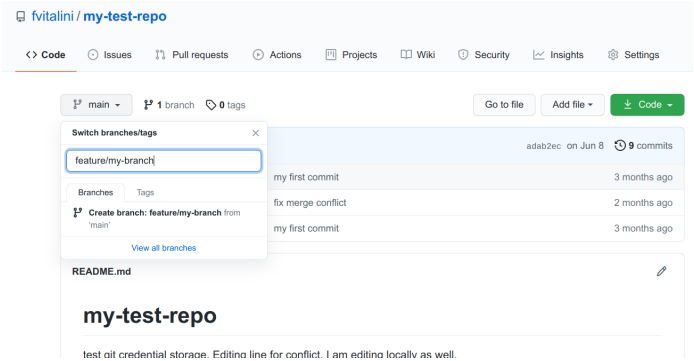
In addition

- `feature/`, `release/`, `hotfix/` branches

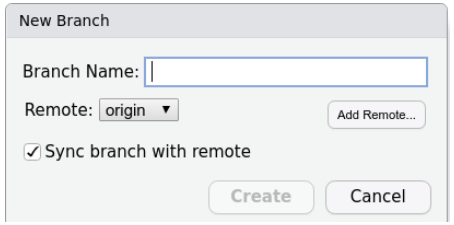


Create Branches

- On **GitHub** create a new branch by writing a branch name (e.g. `feature/my-branch`) that does not correspond to an existing branch in the branch dropdown. Note that the current branch will serve as base for the newly created branch. The new branch will be available in your local repository upon the next `pull` (or `fetch`) operation.



- In **RStudio**: click on the branch button in the Git pane and provide the branch name in the pop-up window.



- From **command line**:

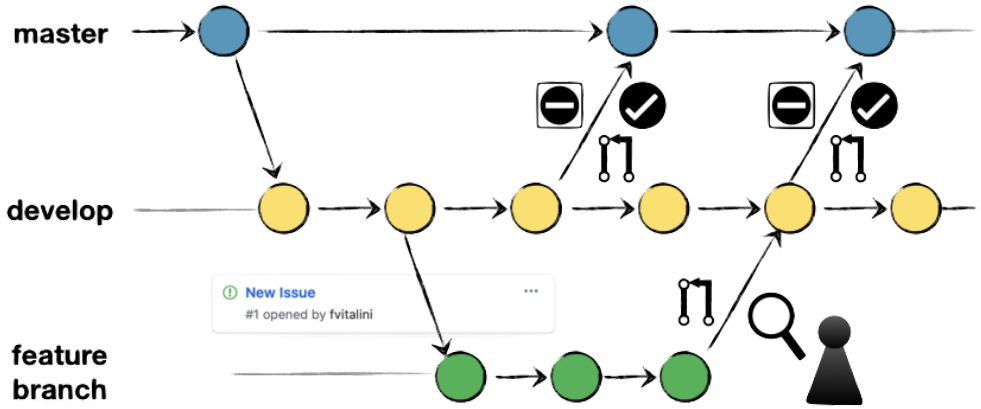
```
git checkout -b feature/my-branch
git push -u
```

creates and switches to the new branch and sets the upstream remote tracking.

Pull requests and branch control

It is important that the code is checked before it is merged back into the main development / release line, to ensure robustness and stability of the code. This can be ensured by setting restrictions on branches:

- Settings > Branches
- `develop` becomes the default branch
- Branch protection for `main` and `develop`
  - Commits to these branches are pushed only via pull requests (PR)
  - Require status checks (e.g. Travis CI, GitHub Actions) to pass before merging a pull request
  - Require pull request reviews from collaborators before merging



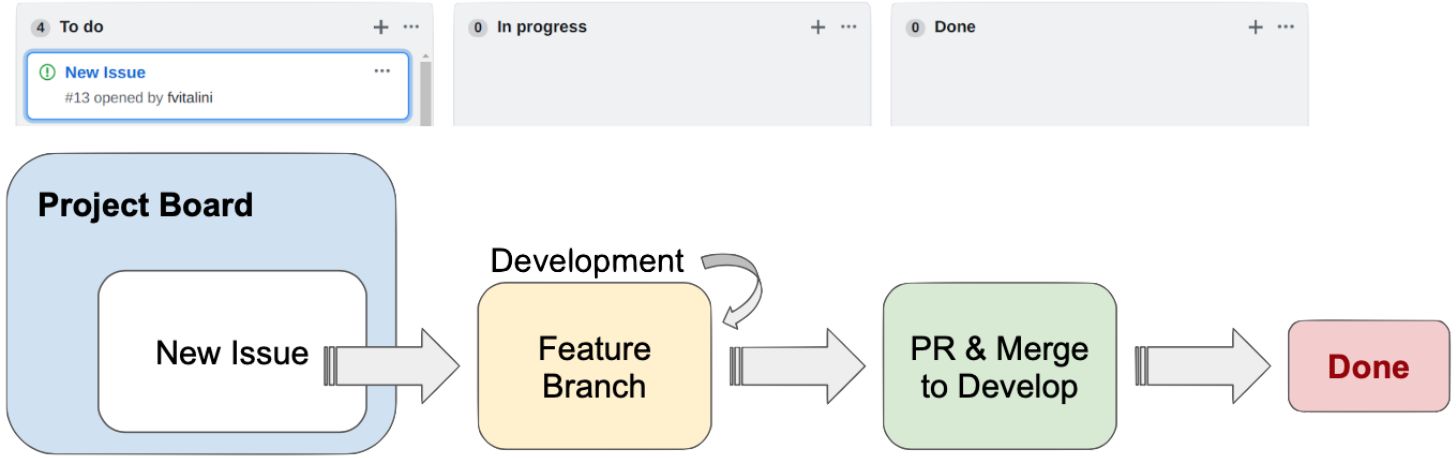
Issues, projects, pull requests

Track development items as GitHub Issues and Projects (visual boards)

- **NOTE:** for forked repos, Issues / Projects are typically disabled by default => Repository **Settings**

GitFlow:

- `feature/<ISSUE_NR>_<branchname>` branches for developing individual issues in isolation
- Development reviewed and safely integrated into `develop` via pull request
- Mention the `<ISSUE_NR>` in your commit messages and pull requests for documentation and automate tracking of open / close issues

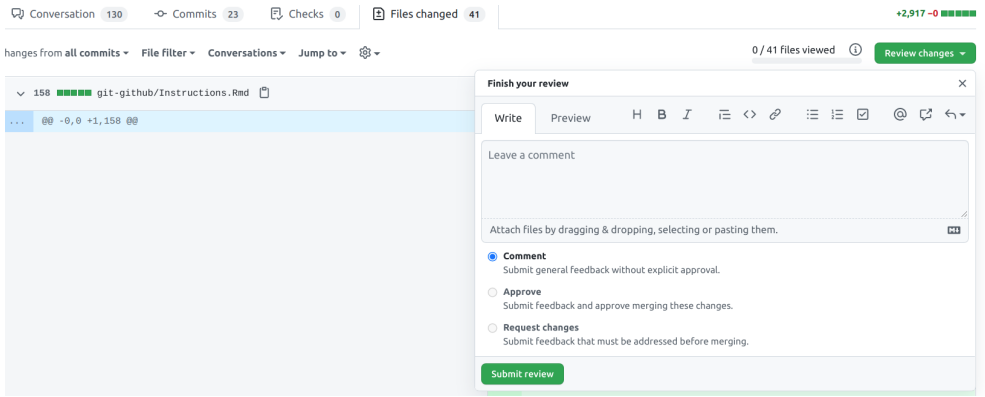


Reviews on GitHub

GitHub offers great aid for Pull Requests Reviews.

The reviewer can:

- make comments
- propose suggestions that can be seemingly integrated (standalone or in a batch) directly on GitHub
- request changes
- approve the Pull Request



Exercise: Work with Branches

- From `main`, create a `develop` branch
- Change the protection rules of the branches on GitHub
- Create a `feature` branch and check it out locally
- Modify the README with some text. Commit + push
- Make a pull request to integrate your changes into the `develop` branch

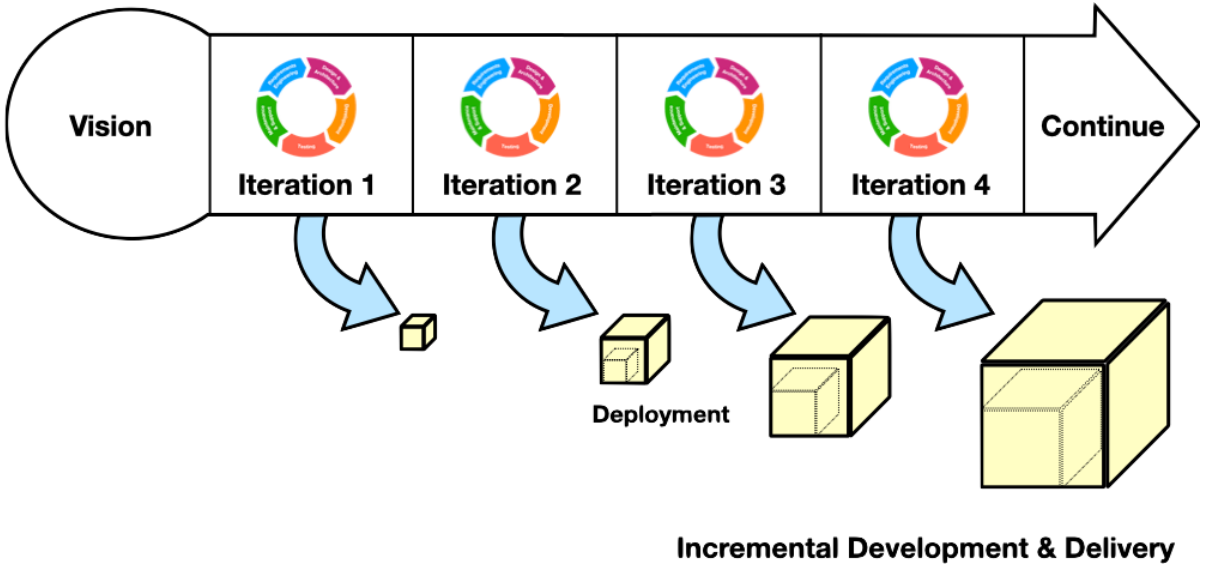
Exercise: Pull Request

Using your Fork of my repo https://github.com/fvitalini/my-test-repo, create a pull request to merge your changes with the main repo.

Agile development

GitFlow fits well with Agile frameworks:

- During a development cycle a set of Issues is worked on according to prioritization
- At the end of each development cycle it is possible to perform a release and have a development increment onto production
- Each feature can be inspected and feedback can be given before its release, keeping the final product always up to date with the end user needs



Incremental Development & Delivery

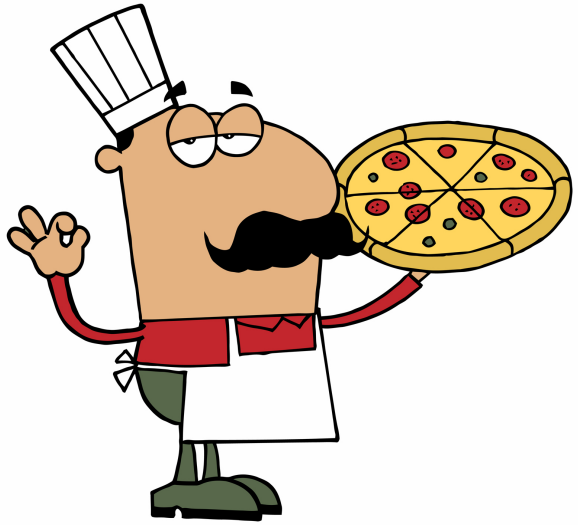
Release

A Release is the deployment to production of a consolidated set of development. Each new release determines a new version (minor or major) of the software.

According to the GitFlow approach, no work should be pushed directly to `master`/`main`, which is updated as part of a release with a pull request from a stable `develop` branch or a `release` branch.

You can have **Release tags** marking the status of the code at a certain point in time.

According to another Branching Model you can open release branches from `develop` (no `master` / `main` needed), marking the different releases of your product.



Ref:clipart-library

Resources

- Git documentation
- Happy Git and GitHub for the useR, Jenny Bryan, Jim Hester
- Git and GitHub, R packages, Hadley Wickham
- Visual Git Guide
- Gitflow
- Oh Shit Git!

Get in touch

Other Services by Mirai Solutions:

- Training customized to your business case - technical and agile topics
- Data Analytics & Software development: from your needs to a functional implementation
- Infrastructure design, management and maintenance

Get in touch:

- LinkedIn: https://www.linkedin.com/company/mirai-solutions-gmbh/
- email: info@mirai-solutions.com
- website: https://mirai-solutions.ch

Thank you!

Extra: Other common Git commands

Remotes

- `git remote -v` lists all remotes.
- `git remote add origin <GITHUB_URL>` adds the remote `<GITHUB_URL>` with nickname `origin`. The remote is typically named `origin`, or `upstream` for forks.
- `git remote set-url origin <GITHUB_URL>` changes the remote url of `origin` to `<GITHUB_URL>`. This way you can fix typos in the remote url.
- `git push --set-upstream origin main` push local changes to remote branch main while setting it as the default remote.
- `git clone <URL GitHub repo>` new local Git repo from a repo on GitHub.

Extra: Other common Git commands

Status

- `git status` indicates the status of the current branch, i.e. lists changes or untracked files, and the sync state with remotes.
- `git log` to show history of commits.
- `git diff` compare two Git versions.

Origin

- `git push origin main` push from a local branch to remote main branch (`git push [remote-name] [branch-name]`).
- `git pull origin <branch-name>` pull from a different branch on remote.

Branches

- `git branch [branch-name]` create branch.
- `git checkout [branch-name]` switch branches.There should not be uncommitted changes.
- `git checkout -b [branch-name]` create branch and switch to it.

Extra: Other common Git commands

Fix History

- `git reset HEAD^` undo last commit, i.e. reset local working directory to parent of the current commit and leave modified files as uncommitted.
- `git reset --hard HEAD^` to undo last commit and remove any changes that were not reflected in the commit-before-last.
- `git reset --soft HEAD^` to undo the last commit, but keep the files as they are; changes will be shown as staged.
- `git reset --mixed HEAD^` (default) to undo the last commit, but keep the files as they are; differences will be shown as modifications.
- `git commit --amend -m "New commit message"` change last commit message.
- `git push --force` overwrite all pushed commits not in sync with local commit history and push your latest commit. Necessary for amending or resetting pushed commits.

Extra: Fetch vs Pull

`git fetch origin`

Fetch only downloads new data from a remote repository without integrating it with the local working directory. It is a great way to view what has changed in a harmless way.

To integrate the changes it can be followed by a `git merge`.

`git pull origin main`

Pull downloads and integrates new data into the current local working directory. Since it tries to merge remote changes with your local ones, merge conflicts may occur.

Merge conflicts can be handled manually in RStudio, or with the aid of a tool, e.g. Mirai's RStudio addin compareWith.

If you have uncommitted local changes, Git may prevent you from performing a `pull`.

Use `git stash` to store local changes temporarily without committing them.

Extra: Pull vs Pull rebase

`git pull --rebase`

can be used to create a nicer commit history (less cluttered and more linear) during integration, as it avoids a merge commit.

It does not prevent conflicts, which will be more if they occur, but potentially less complex to resolve as they would each include smaller bits of development.

Use `git rebase --abort` to interrupt the rebasing.

Notice that now you can do pull with rebase directly from RStudio.

More about rebase and merge: https://git-scm.com/book/en/v2/Git-Branching-Rebasing#_rebase_vs_merge

Extra: Cherrypicking

`git cherry-pick <commit-hash>` is used to apply the changes introduced by some existing commit. It is common to use cherry-picking to choose a commit from one branch and apply it onto another.

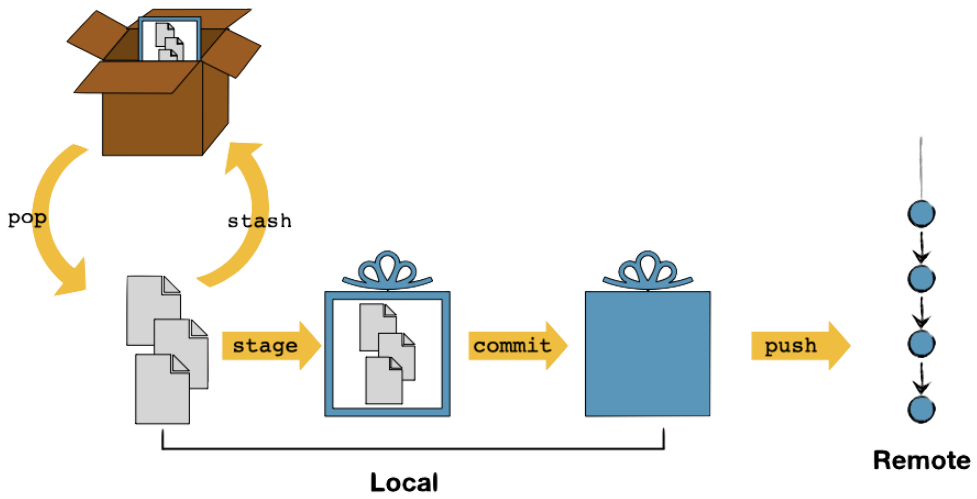
In contrast to `merge` and `rebase` which apply many commits onto the current branch, cherry-picking only applies the increments introduced by the picked commit.

Extra: Git Stash

`git stash (push)`

stores the current state of the local working directory and reverts the working directory to match the HEAD commit.

A stash is by default listed as `"WIP on <branch> …"`, but a more descriptive message can be provided via the `-m` argument.



`git stash list`

lists the stashed modifications, which can be inspected via `git stash show`, and restored (potentially on top of a different commit) with `git stash apply` (`git stash pop` is the same but also removes the stash entry from the listing).

Extra: Credentials

Password-based authentication for Git is deprecated, one should use a Personal Access Token instead (PAT), a must for Two-factors Authentication. This works for HTTPS remotes

Creating a PAT on GitHub

Creating a PAT in RStudio

```
usethis::create_github_token()
```

Extra: Store Credentials

Store Credentials in Git is different by operating system.

Store the Credentials by command line (LINUX):

```
git config --global credential.helper cache
```

In RStudio you can use either of these two packages:

- gitcreds

```
library(gitcreds)
gitcreds_set()
gitcreds_get()
```

- credentials

```
library(credentials)
set_github_pat()
```