DATATHON – Git Introduction

November 2021

SCOR
The Art & Science of Risk

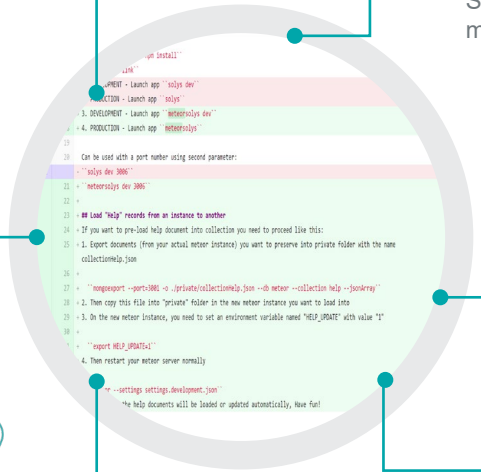
Why Git?

A useful tool to develop solutions

Maintain your code
Git can track modifications on any flat files containing text. It allows you to tag operational versions. You can then easily switch versions.

Share to a community
Git is made for collaborative works. People can access to specific part of your code and contribute to increase it. You always control the final version.

Trusted
Used by almost every developer



Organize your work
Split the job into independent branches you can merge once tests are passed

Back-up
It is also a way to safely back-up your works

Versioning
Track modification on a file

The git environment

What you need to know before starting

Your laptop

(also called “Git client”)

PUSH

PULL

- A Git environment is composed of **two devices : one remote platform and your laptop**
- The user **controls which files** need to be synchronized and **when** they should be
- The **remote platform** is a **central point** which **receives** modifications from users (“push”) working on the same git project.
- Users can **download** the last state of the project **whenever** they want (“pull”) on their laptop.

The platform

(also called “Git server”)

Starting with git (basics)

Below are basic steps to start and continue a project

Create a repository

The first step is to create a new project (or also called repository)

Clone the repository

clone is a git action to synchronize a remote repository with your laptop. It creates automatically a new folder with the name of the repo

git command

git clone <url>

Add a **remote** server into your config

You can also initialize a repository with existing working : you init and add information of the remote server you want to synchronize* with

git command

git init

git remote add origin <url>

Start your project, **add** new files

You can now add new files in the project folder. Since they are new, you have to tell to git that it has to start to track modifications on them.

git command

git add <newfilepath>

Validate the modification on those files.

Once you have finished an action on a specific file you can create a kind of restauration point: commit

git command

git commit -m <message to describe your work> <file>

Synchronize your work with the remote server

Once you have finished your work and tested it you can submit (push) your modification to the server

git command

git push

*If you have added remote rather than cloning, you need to set your working folder to get synchronized with the upstream folder. This step is performed once

git push –u origin -all

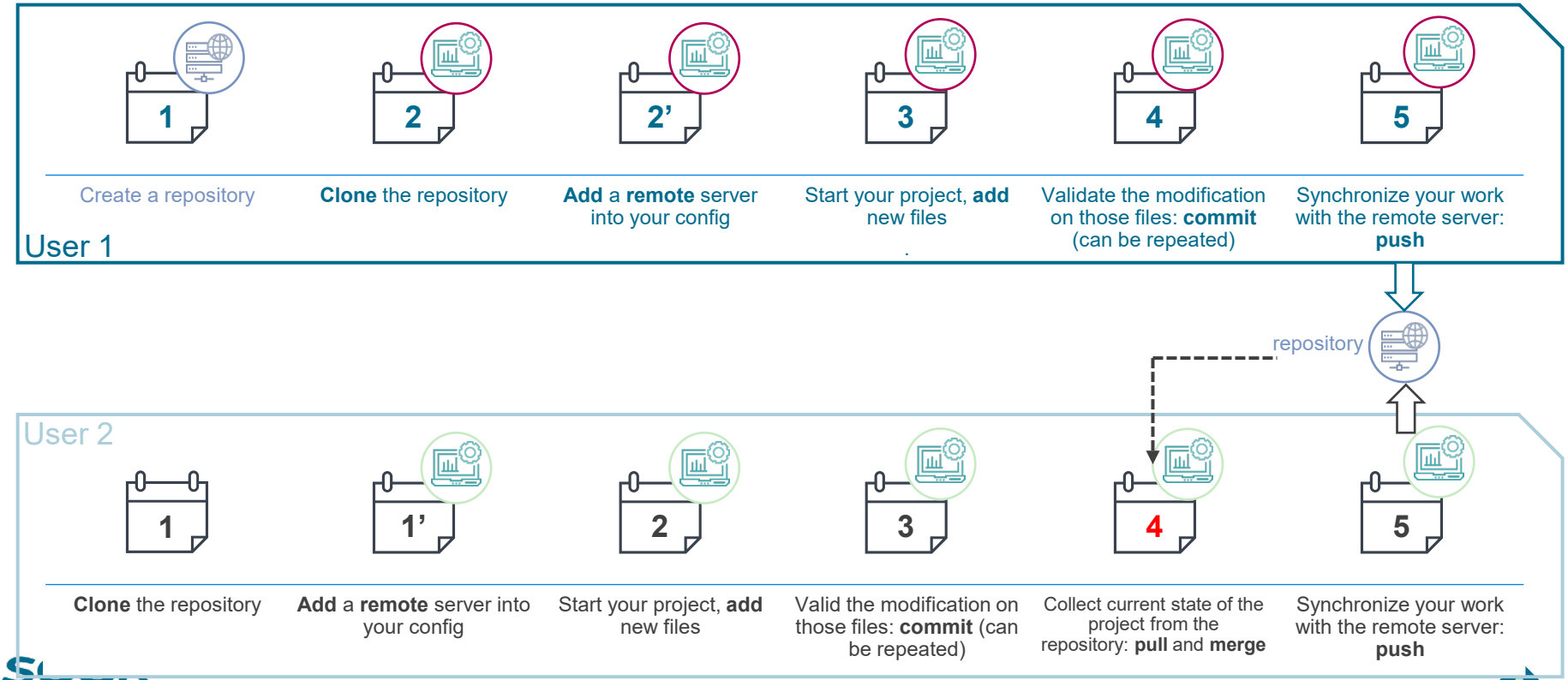
Action done on the git platform (already done in your case)

Action done from your laptop

SCOR
The Art & Science of Risk

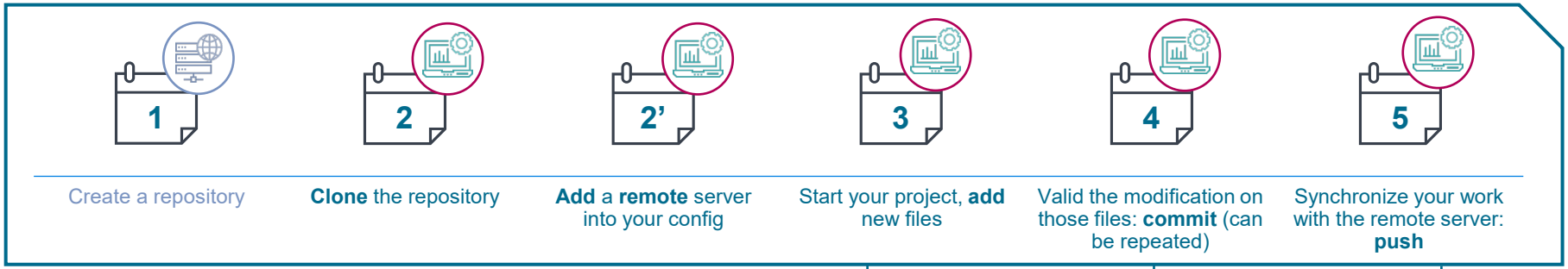
Managing git with several users (basics)

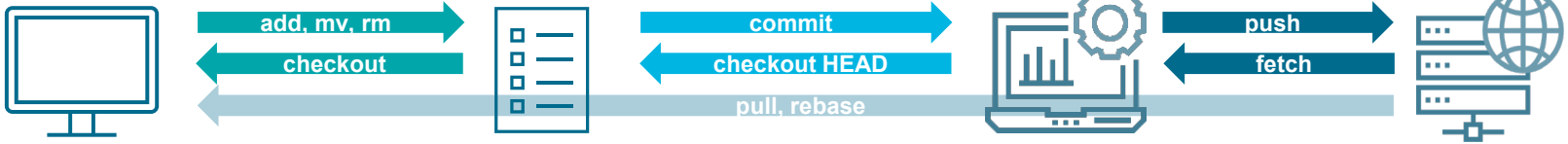
What happens when several users are on the project?



What happens behind the shelves

Understanding Git mechanisms and terminology





Workspace

After initializing it, you start working on your **workspace** by **adding, editing or removing** files.

Index

Git uses an **index** to **track changes** in files. By default, files are ***untracked*** and need to be ***added*** to the index to be tracked. Index is also a **staging area** in which you add **files modifications**.

Local Repository

When changes submitted to the index are ***committed*** to the **local repository**, a **checkpoint** will be created. The local repository is a local copy of the remote repository and can be synchronized at will.

Remote Repository

Remote repository acts as a **pivot** for all users. **Synchronization** with local repository is triggered through ***pushes*** and ***pulls***.

SCOR
The Art & Science of Risk

Command summary

Main basic commands to start a project

- git clone <url> : starts a project by downloading a starting folder from the platform. **This has to be used only once**.
- git remote add origin <url>: adds the url of the remote platform in the git configuration. ‘origin’ is a default name to design the main remote.
- git add <newfilepath>: this command has to be called each time a file is created or modified into the project folder. It tells git to start tracking modifications for new files. If the command is not called, the new file will never be synchronized with the remote.
- git mv <oldfilepath> <newfilepath> moves or renames a file
- git rm <filepath>: removes a file locally and tells git to carry this change in the repository
- git checkout <filepath>: restores a file to its previous commit state as long as changes are not added to the index. If file has been added, you may undo the add through a git checkout HEAD <filepath>
- git commit –m <message> <file>: can be explained as a validation task. It basically creates a checkpoint of the file. Message has to be explicit so it is easier to track modifications in case you need to restore this state of your code.
- git status: lists untracked and modified files
- git diff: shows modifications on files that have not been added to index
- git push <remotename> <branch>: uploads your work (commits) on the remote server. The remote name is one of those you defined with the git remote add command (« origin » by default). The branch name is « master » by default.
- git pull <remotename> <branch>: allows a user to download the current state of the remote repository and synchronize with the local files. It merges by default files modifications or ask for merging manually if it cannot solve conflicts.
- git fetch <remotename> <branch>: is similar to pull but only grabs history of commits locally without synchronizing files

Working in teams (basics)

Sharing only selected files

Ignoring files

Purpose : Avoid tracking files that should remain local or which pollute the repository (e.g. data, code caching, personal settings)

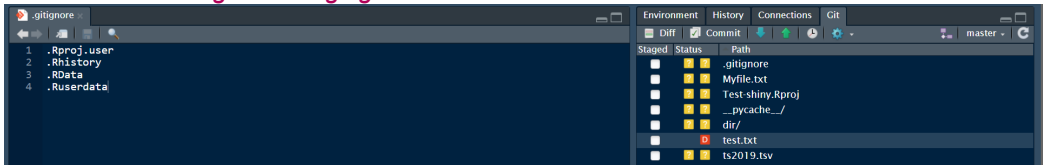
Solution : Create a .gitignore file at the root of your project, and specify which files should be ignored

Going further : The .gitignore file is a shared file across the project. If you want to specify ignore files only for you, you can use the .git/info/exclude

Example :

- `Myfile.txt` ignores file Myfile.txt
- `dir/*` ignores everything in folder dir
- `*cache*` ignores anything that has cache in its name
- `*tsv` ignores everything ending with tsv

> Prior to adding files in .gitignore



> After adding files in .gitignore

Files in the .gitignore cannot be added anymore (unless using the –f force flag).

Working in teams (basics)

Solving conflicts

Conflicts

Purpose : Pulls, pushes and merges* can sometimes lead to conflicts in files. Git prompts an alert, and you must solve conflict to pursue your coding.

Solution : Files entering in conflict will be altered so that you can compare both version of the file with well defined separators :

```
<<<<<<
>>>>>>
======
```

To solve the conflict, you need to remove those separators and keep the desired portion of the code (any, none or both) before committing again.

Example :

Let's admit that both you and your colleague have a version A of a file. You did some modifications resulting in a version B1 while you colleague did other modifications resulting in a version B2 and both of you want to push changes.

A

```
Hello World
My name is
1+1=2
```

B1

```
It's 3:00 PM
Hello World
My name is Mario
1+1=2
```

B2

```
Hello World
My name is Luigi
1+1=2
2+2=4
```

B1 pushes his changes. When B2 wants to push too, Git will ask him to pull changes of B1 before. Some lines (e.g. top and bottom) can be merged without conflict, but an issue will appear on the same modified line.

No conflict

Conflict

```
It's 3:00 PM
Hello World
<<<<< HEAD
My name is Mario
=====
My name is Luigi
>>>>> Some commitid
1+1=2
2+2=4
```

- To solve the issue, B2 needs to remove the 3 lines containing the separators and choose which version of the code he wants to keep (none, B1 version, B2 version, or both) before pushing chances again.

** See next slide for the notion of merge*

Working in teams (basics)

Working with branches

Branches

Purpose : When working in teams, one of the best practices is to work with branches. Branches enable creating copies of existing work so that modifications can be made in parallel prior to bringing them back to the "reference" branch (master or develop).

Solution : Branches can be created from any commit and relies on the `checkout` command. Like for files, checkout can be used to navigate between commits or branches. Branches can also be merged through the `merge` or `rebase` commands.

Going further : Git flows list best practices for branches. These will be detailed in advanced training.

Example :

- 1. Switch to the branch and/or commit id from where to start your branch. In most cases, the starting point will be the "master" or the "develop" branch.
  `git checkout <branchname>` or `git checkout <commitid>`
- 2. Create your new branch by adding the flag –b
  `git checkout -b <newbranchname>`
- 3. When you create a branch for the first time, you need to push it upstream and have the remote track it. This action need to be done only once.
  `git push –u origin <newbranchname>`
- 4. Proceed to all your modifications, adds, commits, pushes, etc.
- 5. When you want to bring your changes to another branch, use checkout to go the target branch, and then merge changes originating from the new branch. Resolve conflicts if necessary, just like for pushes / pulls.
  `git checkout <branch_to_receive_merge>`
  `git merge <branch_sending_modifications>`
- 5'. Rebase can also be used as an alternative for merge. However, since rebase squashes history of commits, it is generally recommended to use merge when working in teams.
  `git rebase <branch_sending_modifications>`

Thank You

SCOR IT Innovation

SCOR
The Art & Science of Risk