4D on GitHub

By Ayoub Khali, Technical Services Engineer, 4D Inc.

Technical Note 19-11

Table of Contents

Introduction

Newbie or full-fledged senior developer, version control has been a staple in every work environment where multiple resources tend to collaborate in order to participate in what could be called a software's chain of production. No matter what their scenario might be, what language/frameworks they are working with, one piece of advice remains constant: The day a program or a webpage contains more than 10 lines of code, GIT becomes a necessity or as some may call it: a necessary evil. While getting to know GIT may seem a laborious and complicated task at first, once grasping the power and safety that versioning offers, coding and conflict management will not be a bother anymore.

This technical note will introduce GIT the revision control system with GITHUB as the hosting service and explain from scratch in details the installation and configuration steps, different ways of interacting with GIT repositories using GIT bash.

Overview

GIT, similar to its older counterparts (CVS, SVN...) is a revision control system primarily developed for Linux but that also runs on Mac and Windows systems. GIT took a radical approach that differs immensely from its legacy siblings, for it is distributed rather than centralized which means that every user has a complete copy and backup of the repository data stored locally, which in turn makes access to file history extremely efficient.

GIT simplifies teamwork and makes collaborations in projects much more manageable since every developer's working directory is itself a branch. All modifications on every branch can be merged within the master branch while also taking care of eventual conflicts that may appear. This allows multiple collaborators to work on the same files at the same time, knowing that merging the final files together won't be much of a hassle.

4D Project Mode

Since earlier versions 4D has been a supporter of team development and testing, using its binary system that allowed team members to work together via 4D server. Starting v17 R5, 4D introduced a new way that makes it possible for distributed teams to work on the same source which is project mode. This means that now it is possible to convert the binary source (.4DB file) into a project folder, with every element from the database (forms, methods…) as a text file, while still allowing the compilation of all of the work into a single file for deployment needs.

Converting into project mode will not affect the .4db file but instead will generate a new "Project" folder next to it. This means two things:

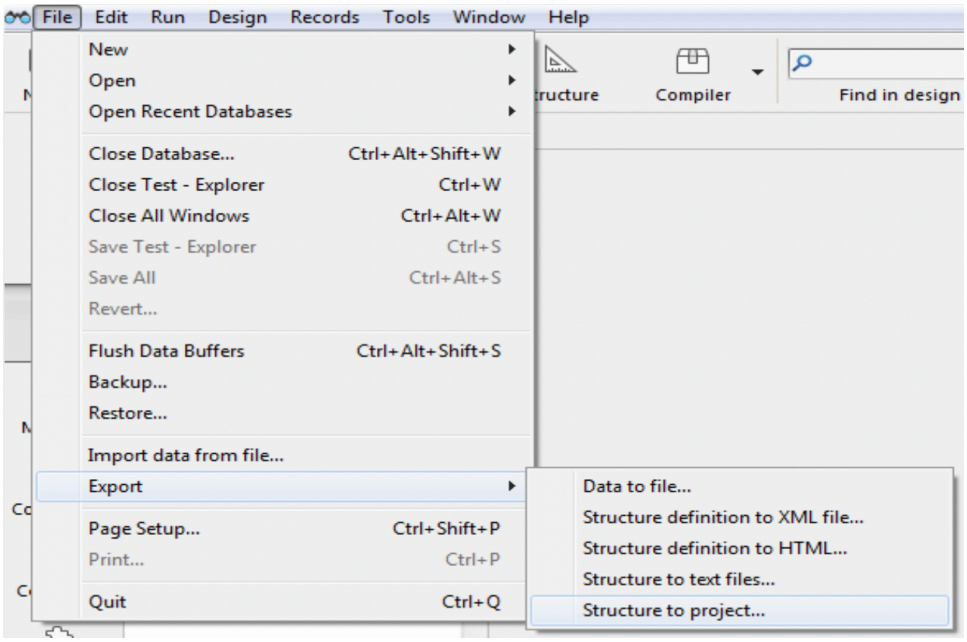
1. This will allow the conversion of the binary database multiple times which could prove beneficial for testing.

2. The conversion is one way only which means that once converted, reintegrating the current changes won't into the .4DB won’t be possible.

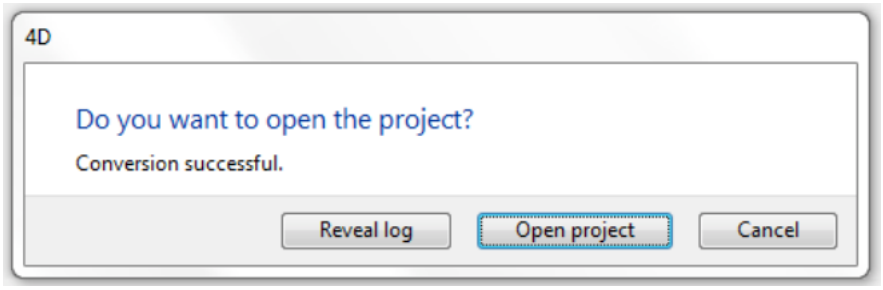
Moreover, converting an existing database is not the only feasible way to access project mode, creating a fresh database project from scratch remains an available option.

Converting a database into a project

The conversion process is simple. From the Design mode, select “File > Export > Structure to project…” menu item.



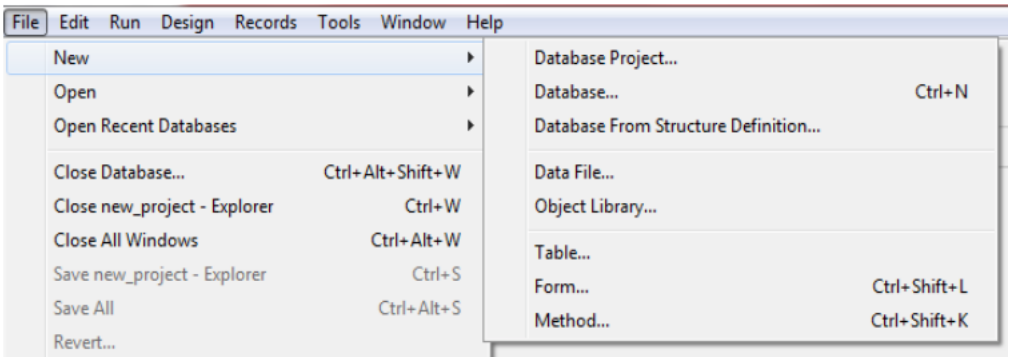
The following prompt message appears once a project has been created.



- Reveal Log: Display the folder containing the log files (JSON) generated during the conversion.
- Open project: Opens the database in project mode.
- Cancel: Remains in the .4DB mode. While in this mode, an attempt to export the structure for the second time will result in an alert informing the user that a project folder already exists. The user will be presented with an option to replace it.

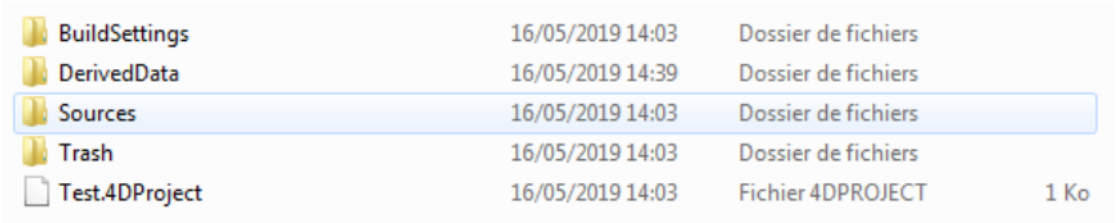
Create a new project

A new project can be created simply by selecting “File > New > Database Project…”.



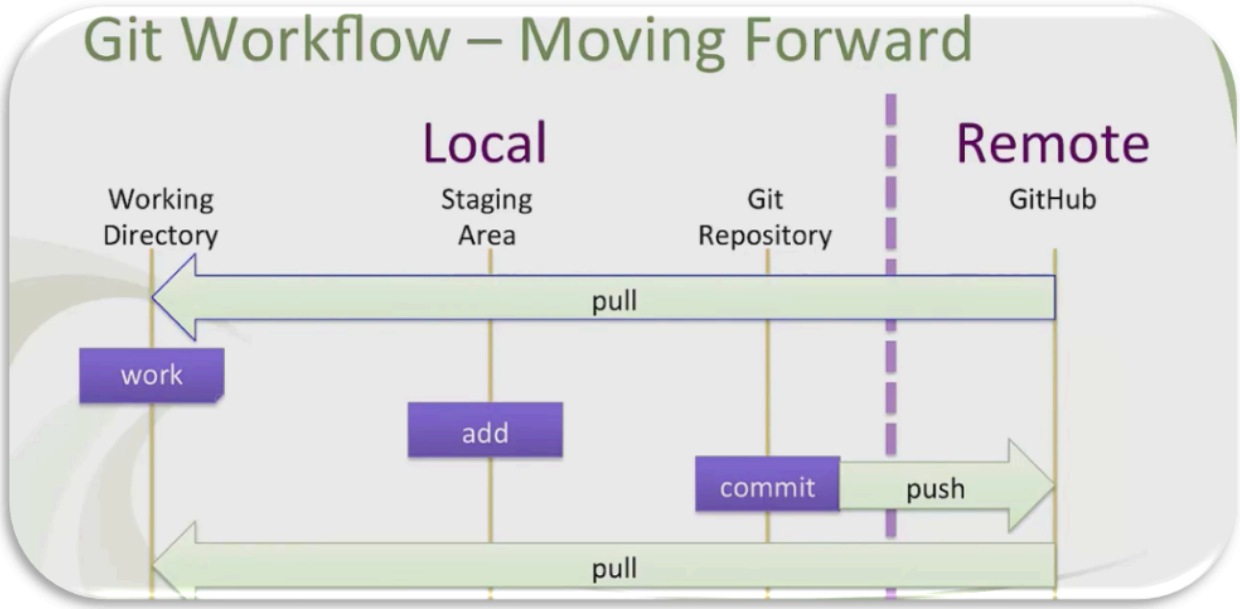
Contents inside a Project folder

Once a project has been generated, the project folder will contain every single element from the database (forms, methods, menus, structure…) as text files (.json, .4dm, .4DCatalog, and etc.) organized by folders.



- BuildSettings folder contains Buildapp.xml.
- DerivedData folder contains metadata about methods and forms (timestamp, destination, and etc.).
- Sources folder contains a file for every form, method, triggers, menus in addition to the database’s settings and structure.
- Trash folder contains deleted forms and methods from the Explorer Trash Bin.
- Test.4DProject

GIT Architecture



Let's first take a quick look at GIT's architecture and try to grasp the whole structure behind it.

The diagram could be divided into two major sections: **Local** and **Remote**.

Local

Working directory: This is where database files are present and where the .git folder will be generated. This area is also known as the "untracked" area of GIT. Any changes to these files will be marked in the working tree. Making changes in this area and not explicitly saving them with GIT can result in data loss. This loss of changes can occur since GIT was not aware of the changes made in the working directory. To sum it up GIT will notice any modifications made on the files present on the working directory but won't save them until the proper commands are entered.

Staging area: Or indexing area is where GIT tracks files and stages modifications that occur in every file in the working directory. The difference between untracked and unstaged is simple: an untracked file is basically every file that was not there in the previous snapshot of the repository (new files for example). In the other hand, unstaged state relates more to modifications, so every time a change is made in a file, this file becomes unstaged since the contents of this file differ from the working tree to the staging area.

GIT repository (or local repository): This is where all files will be saved every time a commit is executed. Committing takes all the changes in the staging area and sends them to the local repository available in the local machine, thus every commit is a request to GIT to track changes that occurred up to this point using the last commit as a comparison. Once a commit is over, the staging area will then remain empty. The one key feature of a distributed version control system is locally having access to the full repository history.

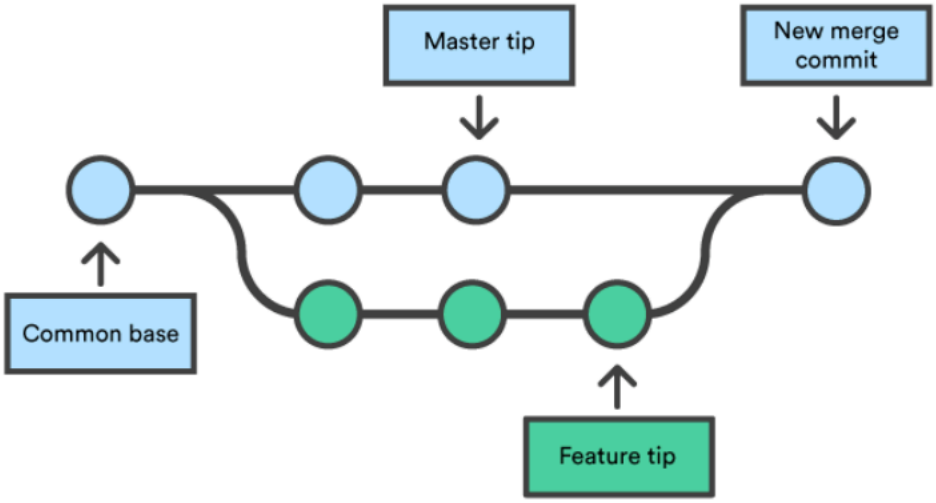
Remote

Remote GIT repository: A remote in GIT is basically a common repository where all team members can send and retrieve files. The remote repository is usually not in the local machine, it is stored in a code hosting service like GITHUB for example but could also be configured to work in an internal server. Team members can push commits to it when ready to share with the team. Note that using a remote repository remains optional in case the team consists of a single developer (unless it's used as backup).

GIT Branching

Branching is a key feature in many modern version control systems and so is the case in GIT. The main purpose of branching resides in its utility which means that instead of a master single branch where every commit will be pushed, different side branches can be involved in this process. Implementing workflows with multiple branches allows greater visibility and fewer conflicts in bigger projects. The master branch will store the official release history while the other branches serve as feature branches that will eventually be integrated/merged with the master branch once development is complete.

Take a look at the diagram below:



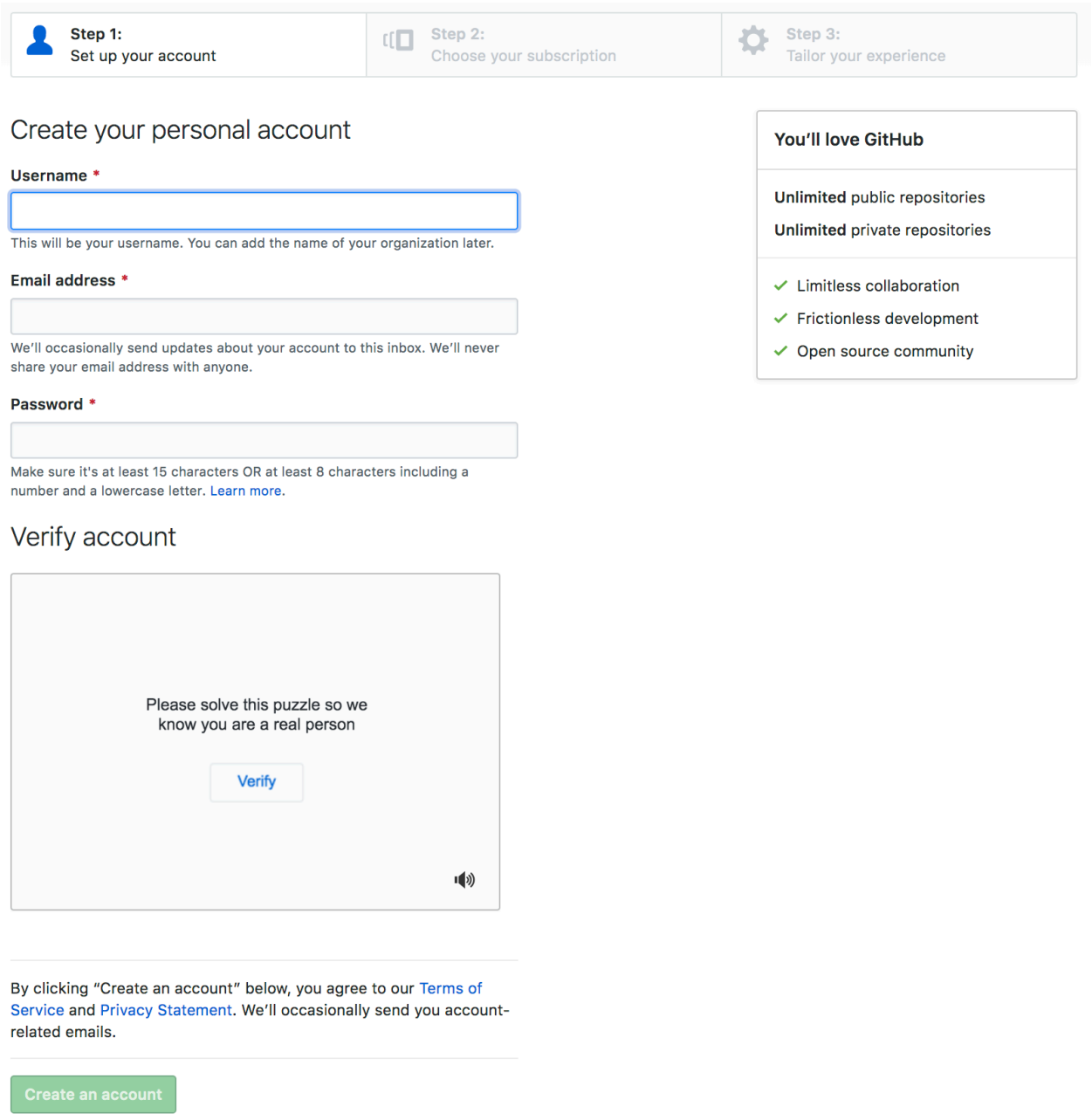
The blue branch represents the master branch while the green one is the feature branch. At project launch, both are in a common base (which means that no features are there yet), once the feature development starts the green branch diverts from the blue one then gets merged back to master when it is complete.

GIT Installation

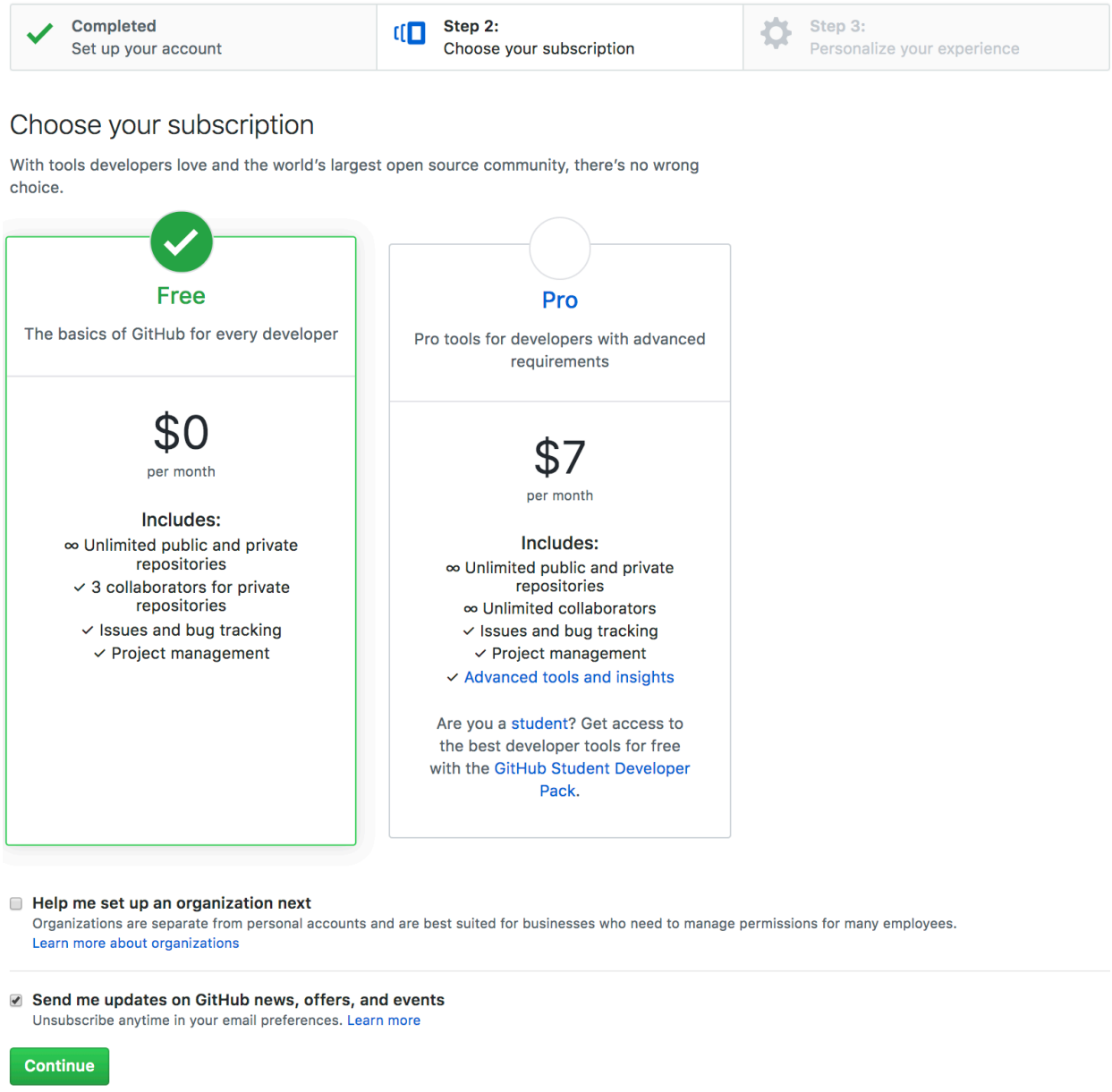
The first step would be creating an account on GitHub: https://github.com/join?source=header

There are 3 steps. Follow each step to complete an account creation.

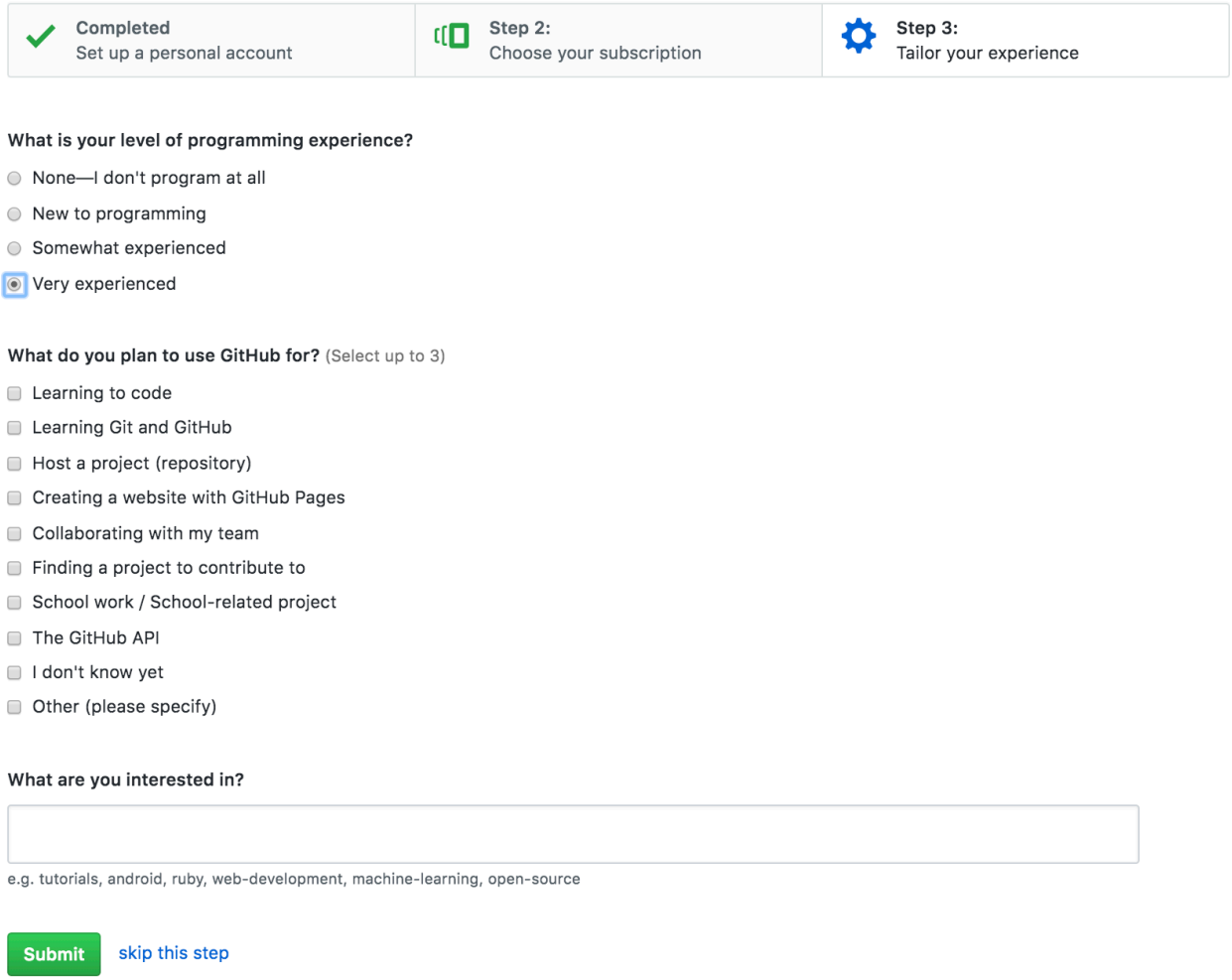
Step 1: Create a new account



Step 2: select “The basics of GitHub for every developer”



Step 3: specify the user experience and plan to use GitHub.



On Windows and macOS

1. Download the latest GIT installer:

- Windows Installer: https://gitforwindows.org/
- macOS Installer: https://sourceforge.net/projects/git-osx-installer/files/

2. Start the installer and follow up with the wizard (next) keeping the default options for now.
3. Open a terminal and verify that the installation was successful by typing the following command: GIT –version
4. Run the following commands in a terminal replacing XXXX with the actual name and email from the GitHub account:

```
git config --global user.name "xxxx xxxxx"
git config --global user.email "xxxxx@xxxx.com"
```

Note: Optional but very recommended: Install the git credential helper: Every interaction with the remote repository will require entering a username/password combination every time which can prove very annoying. Storing them using the GIT credential helper (git-credential-osxkeychain helper for mac) is simple, take a look at the following:

In Windows

To use the GCM, download the latest installer. To install, double-click GCMW-{version}.exe and follow the instructions presented. When prompted to select the terminal emulator for GIT Bash, choosing Windows' default console window would be the right choice.

Link to the installer: https://github.com/Microsoft/Git-Credential-Manager-for-Windows/releases/tag/1.18.5

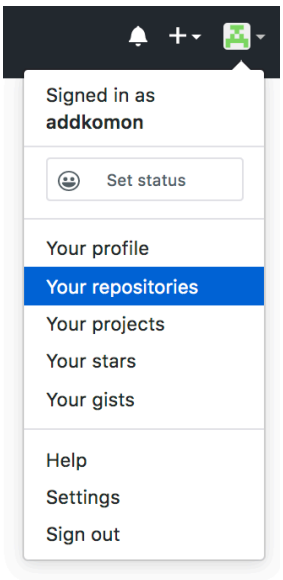
In macOS

Open a terminal window and type the following commands:

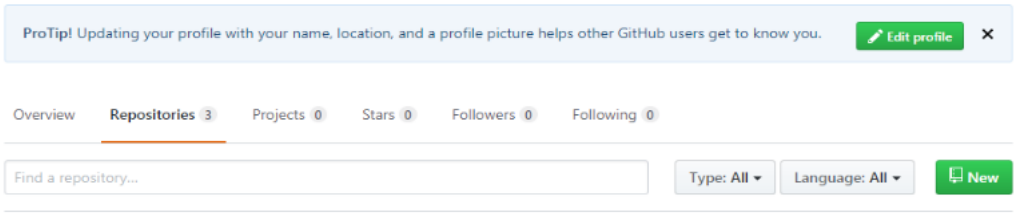
1. To download the tool
    - curl http://github-media-downloads.s3.amazonaws.com/osx/git-credential-osxkeychain -o git-credential-osxkeychain
2. Create folders
    - sudo mkdir /usr/local
    - sudo mkdir /usr/local/bin
3. Moving the files to the right directory
    - sudo mv git-credential-osxkeychain /usr/local/bin/
4. Making the file an executable
    - sudo chmod u+x /usr/local/bin/git-credential-osxkeychain
5. Configuring git to use osyxkeychain
    - git config --global credential.helper osxkeychain

Creating a First Remote Repository

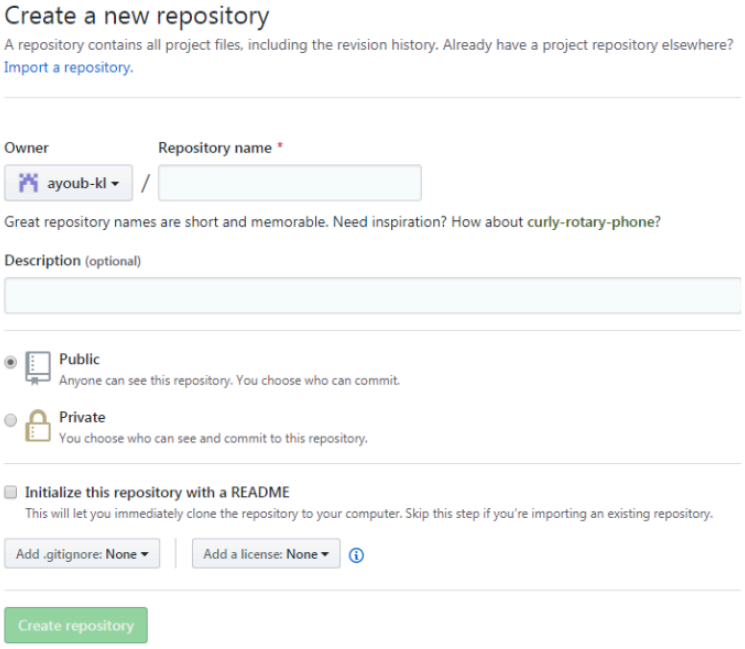
- Go to https://github.com and login
- Click on the upper right corner icon and select "Your repositories".



- The following displays a list of created remote repositories in the GitHub account.



Click “New”.



Repository name: Name of the remote repository that users will push into once created.

Description: An optional field used to describe the repository's content

Public / Private: This configures repositories visibility.

Initialize this repository with a README: Initialize the repository with a README file that could eventually describe the project in detail.

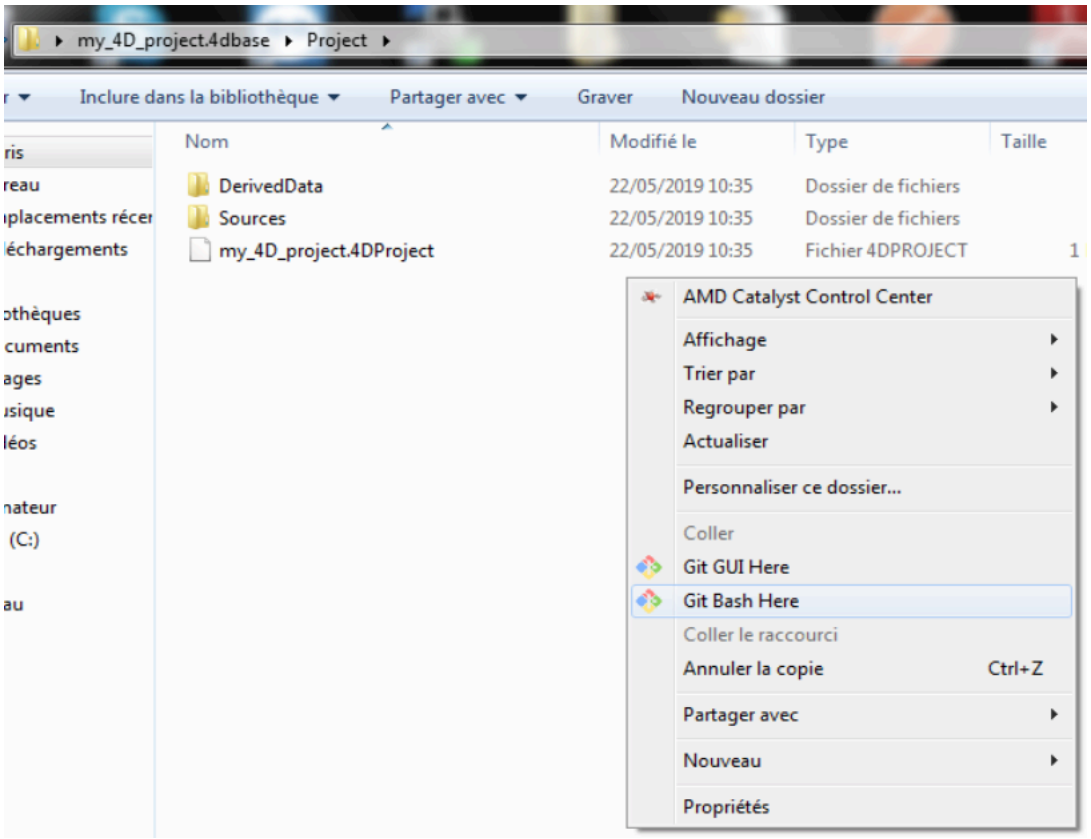
Add .gitignore: A file indicating all the files that are not meant to be tracked

Add a license: Adds a license to the project (GNU, OpenBSD….)

Implementing GIT with 4D Project

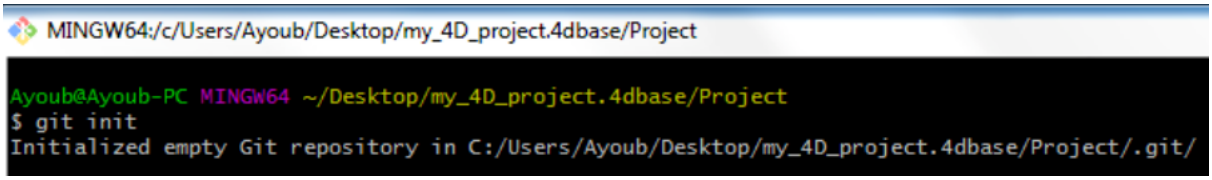
Now that GIT is installed and the remote repository created, proceeding to the creation of a local repository, implementing the 4D project that was either generated from a 4D database or newly created becomes the next step. In this case, the local repository will be initiated inside the 4D project folder.

Navigate to the newly generated project folder, right click and choose "Git bash here".



This will open a GIT bash terminal allowing us to enter commands to perform GIT operations.

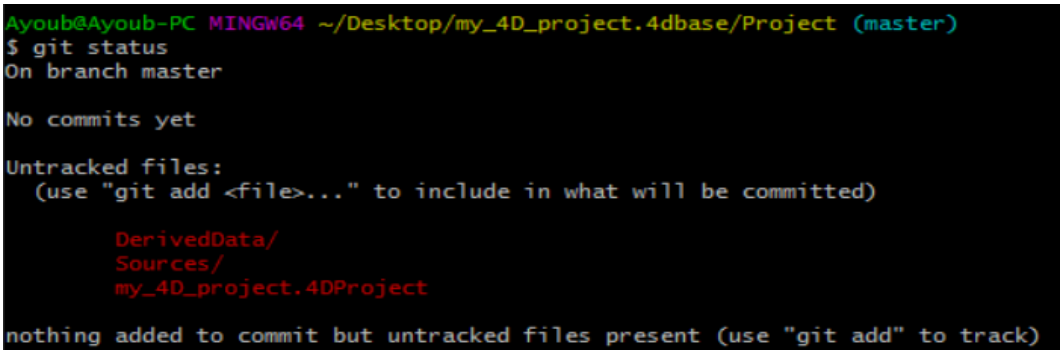
The next step would be initializing a local repository:



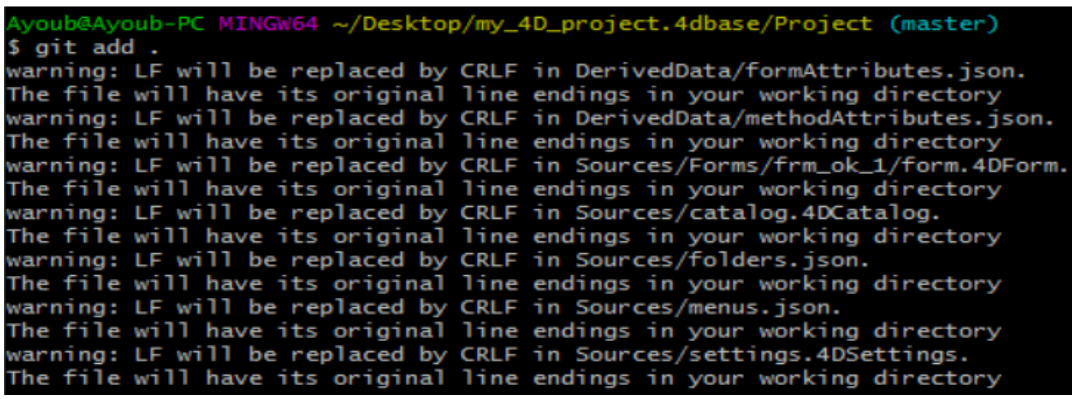
Git init will create an empty local repository, it simply does so by creating a .git directory containing template files and subdirectories.

Since the **git init** was executed in a 4D project folder, GIT already has a list of untracked files available.

The **git status** command will list all modified files (or new ones) which can be added to the local repository. The result shows that 3 elements are not included in the index yet, which means that committing changes with these elements cannot be done for now. In order to do so, staging (adding to index) these folders and files will be the first step.



The **git add** command will prepare the files available in the working tree for the next commit by introducing them into the staging zone.



Git add parameters:

Git add <directory>: to add all the files in a specific directory to the index.

Git add <file>: to add a specific file to the index.

Git add . : Inspects the whole working directory looking for any new, deleted or modified files then adds them to the index.

```
Ayoub@Ayoub-PC MINGW64 ~/Desktop/my_4D_project.4dbase/Project (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   DerivedData/formAttributes.json
        new file:   DerivedData/methodAttributes.json
        new file:   Sources/Forms/frm_ok_1/form.4DForm
        new file:   Sources/Methods/meth_alert_1.4dm
        new file:   Sources/Methods/meth_alert_2.4dm
        new file:   Sources/catalog.4DCatalog
        new file:   Sources/folders.json
        new file:   Sources/menus.json
        new file:   Sources/settings.4DSettings
        new file:   my_4D_project.4DProject
```

Running **git status** once more shows that files are ready to be committed.

```
Ayoub@Ayoub-PC MINGW64 ~/Desktop/my_4D_project.4dbase/Project (master)
$ git commit -m "Committing the 4D database my_4D_project"
[master (root-commit) 575a411] Committing the 4D database my_4D_project
 10 files changed, 199 insertions(+)
 create mode 100644 DerivedData/formAttributes.json
 create mode 100644 DerivedData/methodAttributes.json
 create mode 100644 Sources/Forms/frm_ok_1/form.4DForm
 create mode 100644 Sources/Methods/meth_alert_1.4dm
 create mode 100644 Sources/Methods/meth_alert_2.4dm
 create mode 100644 Sources/catalog.4DCatalog
 create mode 100644 Sources/folders.json
 create mode 100644 Sources/menus.json
 create mode 100644 Sources/settings.4DSettings
 create mode 100644 my_4D_project.4DProject
```

Git commit sends whatever files were available in the staging area to the local repository, thus freeing the index. A commit is a snapshot of the elements in the working directory. However, GIT does not copy all files every time a commit is made but instead includes them as a set of changes from one repository version to another.

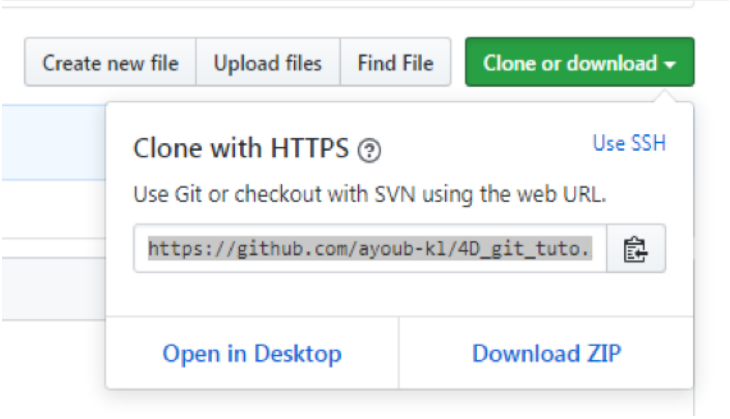
Git commit –m "message": The committing messages have to clear and explicit for backup purposes.

Git commit –a –m "message": The –a parameter is useful when trying to commit changes from files that were already added once in their lifetime to the repository.

Now that the files in the working directory have been successfully committed to the local repository, the next step would be pushing these changes to the remote repository to be shared with the rest of the team. One important note is that before pushing any changes to the remote repo., it is considered essential to check beforehand if any recent changes have been pushed by a fellow team member. But before doing any of this, linking the remote repository to the local one is a priority.

```
Ayoub@Ayoub-PC MINGW64 ~/Desktop/my_4D_project.4dbase/Project (master)
$ git remote add origin "https://github.com/ayoub-kl/4D_git_tuto.git"
```

Git remote add origin "link_to_repo": Adds a remote repository named "origin" (By default, origin is used to define a remote repo, however naming it otherwise won't matter). Copying the repository's link can be done as follows:



The link is available from GitHub account / upper right icon / repositories / created_repo.

Git status compares the local branch with its remote counterpart (the remote tracking branch) but this does not always mean that the remote tracking branch is up to date. To ensure that, running a **git fetch** will actually update the remote tracking branch thus allowing the branch to stay up to date with the remote repository (remote branch).

```
Ayoub@Ayoub-PC MINGW64 ~/Desktop/my_4D_project.4dbase/Project (master)
$ Git status
On branch master
nothing to commit, working tree clean
```

Running it will display that the working tree is clean with no commits or changes available, however, in this case, the newly created repository was initialized with a readme file which is not present in our local repository. Running a **git push** would generate a pretty self-explanatory error message:

```
Ayoub@Ayoub-PC MINGW64 ~/Desktop/my_4D_project.4dbase/Project (master)
$ git push origin master
git: 'credential-osxkeychain' is not a git command. See 'git --help'.
To https://github.com/ayoub-kl/4D_git_tuto.git
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to 'https://github.com/ayoub-kl/4D_git_tuto.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

This remains a special case since an auto-generated README is present on the remote repository but not locally. Which leads to the next step which is updating the local repository with changes from the remote, it is basically done using the following commands:

Git pull origin remote_branchname: Downloads and merges changes from the remote repository with the local repository.

Or for a more sophisticated way:

Git fetch origin remote_branchname: Downloads content from the remote repository but does not integrate them in the local repository.

+

Git merge origin/remote_branchname: merges changes fetched by git fetch into the local repository.

In case a newly created repository was initialized with a README file, a simple **git pull or git fetch + git merge** won't be able to update the local repository with the same files at the remote, for they have unrelated histories.

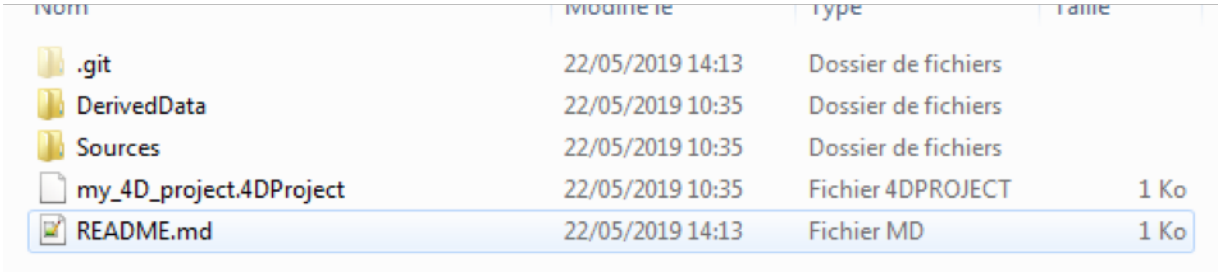
```
Ayoub@Ayoub-PC MINGW64 ~/Desktop/my_4D_project.4dbase/Project (master)
$ git pull origin master
From https://github.com/ayoub-kl/4D_git_tuto
 * branch            master     -> FETCH_HEAD
fatal: refusing to merge unrelated histories
```

In order to fix this, a parameter (flag) is needed with the **git pull** command.

```
Ayoub@Ayoub-PC MINGW64 ~/Desktop/my_4D_project.4dbase/Project (master)
$ git pull origin master --allow-unrelated-histories
From https://github.com/ayoub-kl/4D_git_tuto
 * branch            master     -> FETCH_HEAD
hint: Waiting for your editor to close the file...
Merge made by the 'recursive' strategy.
 README.md | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 README.md
```

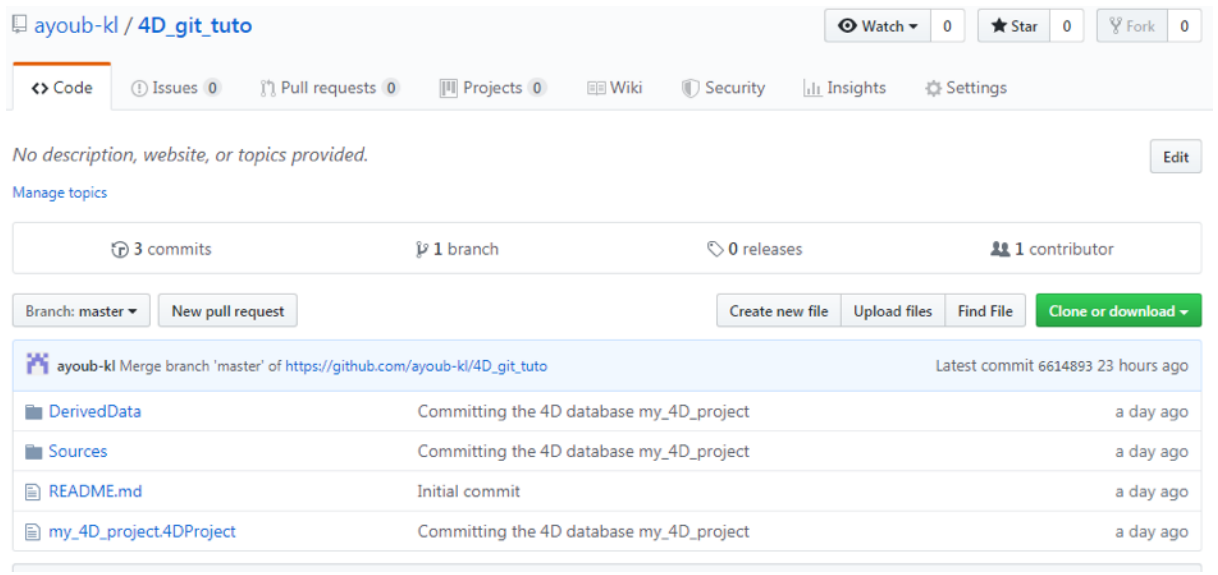
Git pull origin remote_branchname --allow-unrelated-histories: will pull from the remote repository while ignoring the unrelated histories between the repositories. This only should be used as a workaround.

Now that the initial pull is complete, notice that the README.md file has been added to the project folder. Pushing the project to the remote repository now becomes available.




```
Ayoub@Ayoub-PC MINGW64 ~/Desktop/my_4D_project.4dbase/Project (master)
$ git push origin master
git: 'credential-osxkeychain' is not a git command. See 'git --help'.
Enumerating objects: 20, done.
Counting objects: 100% (20/20), done.
Delta compression using up to 8 threads
Compressing objects: 100% (15/15), done.
Writing objects: 100% (19/19), 2.70 KiB | 1.35 MiB/s, done.
Total 19 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), done.
To https://github.com/ayoub-kl/4D_git_tuto.git
   a1f863e..6614893  master -> master
```

Take a look at the remote repository to confirm that all the files have been sent successfully.



Important note: The following GIT command becomes a necessity in order to link the local branches with remote branches (upstream). In this case, linking the local master branch with the remote master branch is done like this:

```
Ayoub@Ayoub-PC MINGW64 ~/Desktop/my_4D_project.4dbase/Project (master)
$ git branch --set-upstream-to=origin/master master
Branch 'master' set up to track remote branch 'master' from 'origin'.
```

Once executed, pushing and pulling from master (remote main branch) will be much easier, simply typing **git push** or **git pull** without any additional parameters will do the trick.

In GIT branches can simply be defined as pointers to specific commits. Whenever a feature is still in a development phase, pushing the changes to a different branch than master could prove very wise since master should always only harbor the tested production ready version. There are two types of branches:

- Local branches: branches available locally in the working tree
- Remote tracking branches: Branches linking local work to its remote counterpart on the central repository.

TO create a new branch and switch to it, use the commands below:

- **Git branch branch_name**
- **Git checkout branch_name**

Git checkout –b branch_name has also the same effect of the two commands.

Once the feature is ready to be deployed, merging the feature branch with the master branch can be done this way:

Git merge source_branch

```
Ayoub@Ayoub-PC MINGW64 ~/Desktop/my_4D_project.4dbase/Project (master)
$ git merge my_feature_branch
Already up to date.
```

Make sure to check out the master branch (to be in it) before attempting to merge it with the feature branch. In this case, no changes were made in a feature branch (thus “Already up to date”)

Sample database

A sample database has been provided with this technical note, which will be used to explain in details different situations and complications that may appear in a work environment.

Note 1: The demo was designed in v17R5 since it is the release in which the project mode is available. Using an older version won’t be possible.

Note 2: It is recommended to follow up with the following demos in order to fully grasp GIT’s workflow with a 4D database.

Demo 1: Interaction with a remote repository

1. Go to the project folder and launch a git bash (right click)
2. Initiate a git repository using **git init**
3. Add a remote repository using **Git remote add origin "link_to_repo"**
4. Add an upstream link to the remote branch master using **Git branch –set-upstream-to=origin/master master**
5. Add, commit and push the files from the project file to the local and remote repository using the following:

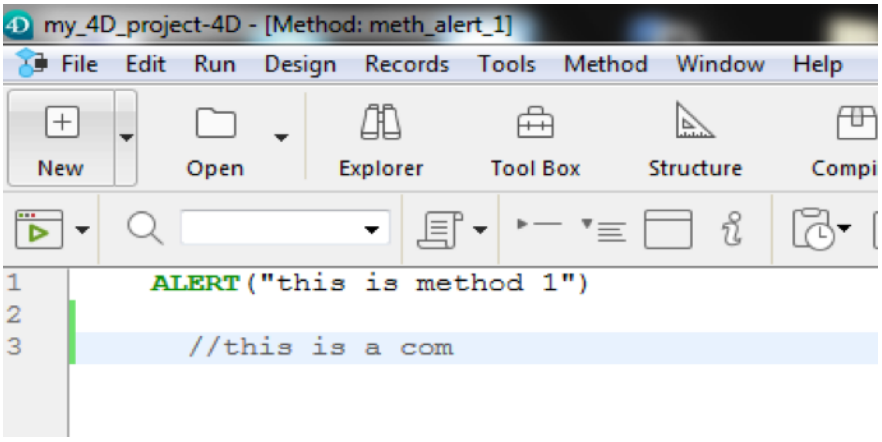
    - **Git add**
    - **Git commit –m "committing our project folder"**
    - **Git push**

    Run a **git status** command to verify the state of the branch (that everything has been committed successfully).

```
Ayoub@Ayoub-PC MINGW64 ~/Desktop/my_4D_project.4dbase/Project (master)
$ Git status
On branch master
nothing to commit, working tree clean
```

    This should be the result if all the operations have been executed with success.

6. Open the sample database my_4D_project (my_4D_project.4DProject) using v17R5
7. Let's modify the method "meth_alert_1" by adding a commentary



    Running a **git status** this time will detect that modifications have been made to the working tree.

```
Ayoub@Ayoub-PC MINGW64 ~/Desktop/my_4D_project.4dbase/Project (master)
$ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   DerivedData/methodAttributes.json
        modified:   Sources/Methods/meth_alert_1.4dm

no changes added to commit (use "git add" and/or "git commit -a")
```

8. Now let's save our modifications by executing the same commands as in step 5. (try to change the commit message). Once done git status should return no more untracked or unstaged files.
9. Let's say the last commit was faulty, in order to revert it running the following commands are necessary:

   Git reset –hard parameter, the parameter can be as follows:

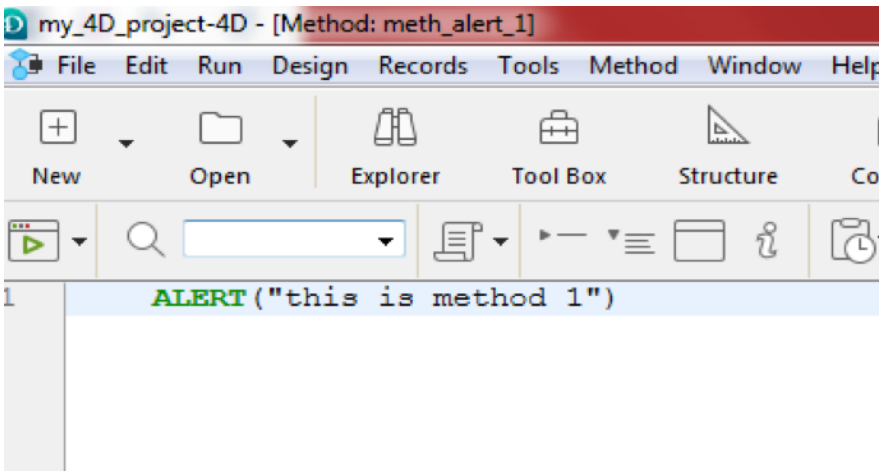
   Git reset –hard HEAD~1: This resets the last commit by HEAD referring to the current branch

   Git reset –hard commit_number: This will reset all the commits up to the commit which number is passed as a parameter. The first thing to do would be getting the list of all commits using the following command: **git log**

```
Ayoub@Ayoub-PC MINGW64 ~/Desktop/my_4D_project.4dbase/Project (master)
$ git log
commit 4ef81eaa0b7465353bb173c10dee489d71c123d0 (HEAD -> master, origin/master)
Author: ayoub <ayoub-kl@outlook.com>
Date:   Fri May 24 12:06:49 2019 +0000

    commiting new changes to method meth_alert_1

commit 66148936b9a91903ca2ce2d188a6dddcd3980f5d (new_feature_branch, my_feature_branch)
Merge: 575a411 a1f863e
Author: ayoub <ayoub-kl@outlook.com>
Date:   Wed May 22 14:13:13 2019 +0000

    Merge branch 'master' of https://github.com/ayoub-kl/4D_git_tuto

commit a1f863eda1a5fde6224526ed0bab6161f0520f94
Author: ayoub-kl <46133199+ayoub-kl@users.noreply.github.com>
Date:   Wed May 22 13:32:28 2019 +0000

    Initial commit

commit 575a411fe2c157573b23e4025b5b088b9e6e642c
Author: ayoub <ayoub-kl@outlook.com>
Date:   Wed May 22 11:27:29 2019 +0000

    Committing the 4D database my_4D_project
```

The result shows all the commits done to the local repository with all important details (author, date, commit number...). Running the following will successfully reset the working directory to its previous state:

Git reset –hard 66148936b9a91903ca2ce2d188a6dddcd3980f5d



Notice that the commentary has been deleted after the reset.

A final push is necessary in order to update the remote repository with the method in its current state. Running a simple **git push** won't work this time since the local branch will be behind its remote counterpart (since the commit was reset).

```
Ayoub@Ayoub-PC MINGW64 ~/Desktop/my_4D_project.4dbase/Project (master)
$ git push
git: 'credential-osxkeychain' is not a git command. See 'git --help'.
To https://github.com/ayoub-kl/4D_git_tuto.git
 ! [rejected]        master -> master (non-fast-forward)
error: failed to push some refs to 'https://github.com/ayoub-kl/4D_git_tuto.git'
hint: Updates were rejected because the tip of your current branch is behind
hint: its remote counterpart. Integrate the remote changes (e.g.
hint: 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

The message suggests running a **git pull,** however, running it would nullify the effect of the reset that was run locally. Solving this is done simply as follows:

```
Ayoub@Ayoub-PC MINGW64 ~/Desktop/my_4D_project.4dbase/Project (master)
$ git push --force
git: 'credential-osxkeychain' is not a git command. See 'git --help'.
Enumerating objects: 7, done.
Counting objects: 100% (7/7), done.
Delta compression using up to 8 threads
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 459 bytes | 229.00 KiB/s, done.
Total 4 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To https://github.com/ayoub-kl/4D_git_tuto.git
 + 4ef81ea...6eb4ec4 master -> master (forced update)
```

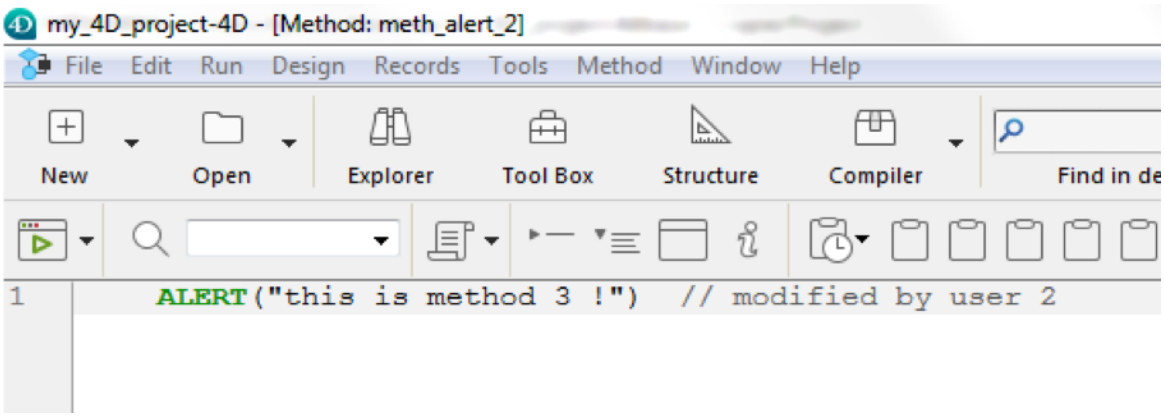
Git push –force: Will push changes from the local repository to the remote one while ignoring the tip's placement.

Demo 2: Conflict management

When working with version control systems in an environment where multiple developers tend to collaborate back and forth, managing eventual conflicts becomes an everyday dilemma. Let's take a look at an example of a conflict using 4D methods/forms and GIT. In order to achieve this, it is necessary to simulate a secondary user/Local git repository by duplicating the database folder (my_4D_project.4dbase).

Once duplicated, open the copied database in a second instance of v17R5 (no need to duplicate the v17R5 folder, launching the exe a second time will open a new instance).

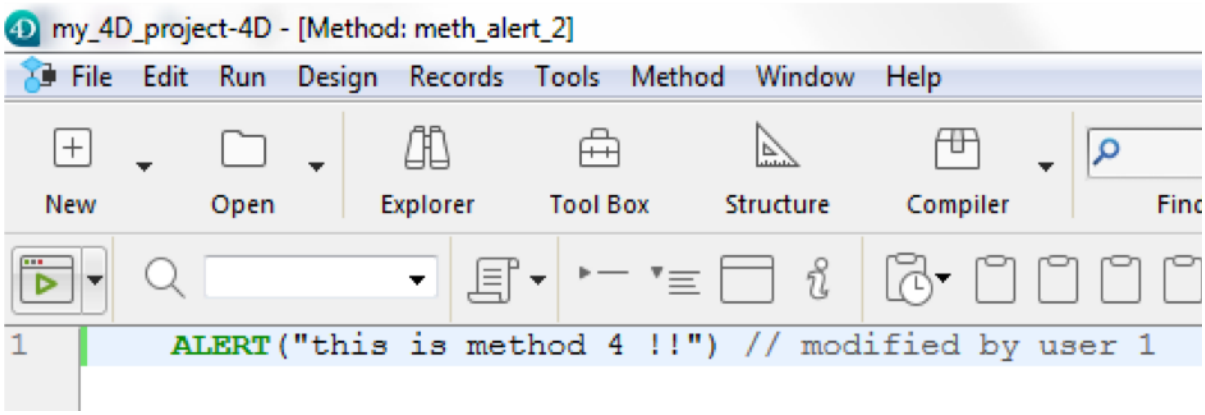
1. Go to the new project folder and launch a git bash (right click).
2. In order to simulate a conflict, open meth_alert_2 in the new database and modify it as follows:



Running a **git_status** will detect that a method has been modified. Let's commit and push these new changes.

```
Ayoub@Ayoub-PC MINGW64 ~/Desktop/my_4D_project.4dbase - Copie/Project (master)
$ git add .
warning: LF will be replaced by CRLF in DerivedData/methodAttributes.json.
The file will have its original line endings in your working directory

Ayoub@Ayoub-PC MINGW64 ~/Desktop/my_4D_project.4dbase - Copie/Project (master)
$ git commit -m "commiting changes to meth_alert_2"
g[master c9d9fe4] commiting changes to meth_alert_2
 2 files changed, 3 insertions(+), 3 deletions(-)
it
Ayoub@Ayoub-PC MINGW64 ~/Desktop/my_4D_project.4dbase - Copie/Project (master)
$ git push
git: 'credential-osxkeychain' is not a git command. See 'git --help'.
Enumerating objects: 13, done.
Counting objects: 100% (13/13), done.
Delta compression using up to 8 threads
Compressing objects: 100% (7/7), done.
Writing objects: 100% (7/7), 708 bytes | 708.00 KiB/s, done.
Total 7 (delta 3), reused 0 (delta 0)
remote: Resolving deltas: 100% (3/3), completed with 3 local objects.
To https://github.com/ayoub-kl/4D_git_tuto.git
   6eb4ec4..c9d9fe4  master -> master
```

3. At the same moment, user 1 decided to modify the same method, so modifying the same method in the first database will be as follows:



4. Commit and push the new changes.

```
Ayoub@Ayoub-PC MINGW64 ~/Desktop/my_4D_project.4dbase/Project (master)
$ git add .
warning: LF will be replaced by CRLF in DerivedData/methodAttributes.json.
The file will have its original line endings in your working directory
gi
Ayoub@Ayoub-PC MINGW64 ~/Desktop/my_4D_project.4dbase/Project (master)
$ git commit -m "commiting new changes to meth_alert_2"
[master d8cdb79] commiting new changes to meth_alert_2
 2 files changed, 2 insertions(+), 2 deletions(-)

Ayoub@Ayoub-PC MINGW64 ~/Desktop/my_4D_project.4dbase/Project (master)
$ git push
git: 'credential-osxkeychain' is not a git command. See 'git --help'.
To https://github.com/ayoub-kl/4D_git_tuto.git
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to 'https://github.com/ayoub-kl/4D_git_tuto.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

Forgetting to run a **git fetch** to check if whether the remote repository has any new commits is essential, in this case not doing so and trying to push caused the error (updates were rejected…).

5. Running git pull as suggested will cause a conflict to arise, take a look at the following:

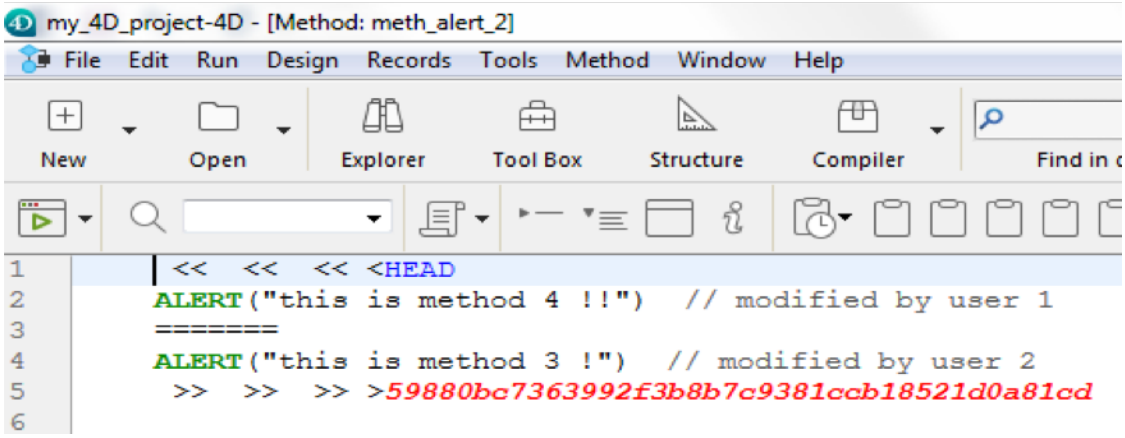
```
Ayoub@Ayoub-PC MINGW64 ~/Desktop/my_4D_project.4dbase/Project (master)
$ git pull
remote: Enumerating objects: 13, done.
remote: Counting objects: 100% (13/13), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 7 (delta 3), reused 7 (delta 3), pack-reused 0
Unpacking objects: 100% (7/7), done.
From https://github.com/ayoub-kl/4D_git_tuto
   0699720..59880bc  master     -> origin/master
Auto-merging Sources/Methods/meth_alert_2.4dm
CONFLICT (content): Merge conflict in Sources/Methods/meth_alert_2.4dm
Auto-merging DerivedData/methodAttributes.json
CONFLICT (content): Merge conflict in DerivedData/methodAttributes.json
Automatic merge failed; fix conflicts and then commit the result.

Ayoub@Ayoub-PC MINGW64 ~/Desktop/my_4D_project.4dbase/Project (master|MERGING)
```

Merging the changes made by user 2 into the local changes could not be solved automatically, also notice that the word master has now changed to master | Merging. There are three ways of conflict resolution methods available:

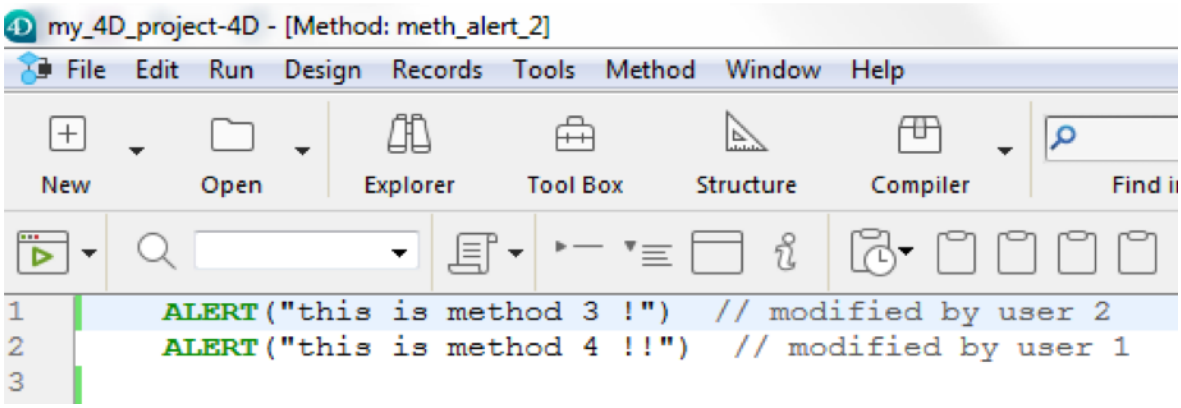
- Git mergetool: A tool that comes along with GIT installation, many other mergetools are available and could be download and integrated to GIT.
- Manual merging: Since all the elements from the 4D database are now generated as text files, merging conflicts can be done via simple text editors (Notepad…)

- 4D editor: This is only true for methods since their json file can be properly displayed in the editor. Forms for example won't be displayed since their json files must be interpreted and rendered by 4D.

6. Take a look at the third method (4D editor), in this case dealing with a method conflict:



Notice that both versions are available in the method editor, with HEAD referring to the local changes and **59880bc7363992f3b8b7c9381ccb18521d0a81cd** representing user 2 commit number. In order to solve this conflict, simply keep the correct part of the code.



7. Concluding the conflict is simple, pick the correct parts of the code, then run the following commands to save the changes and merge the conflict:

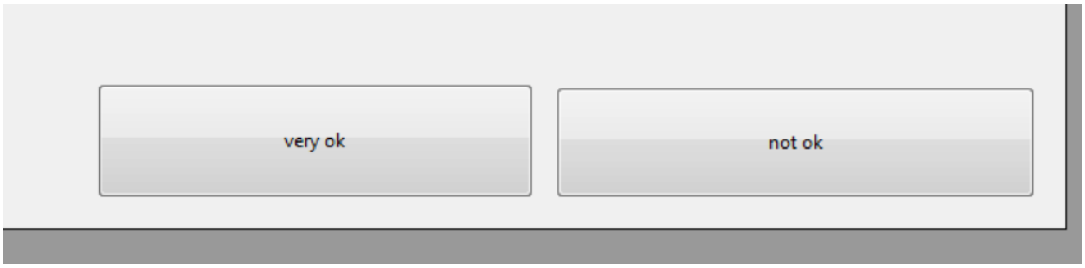
```
Ayoub@Ayoub-PC MINGW64 ~/Desktop/my_4D_project.4dbase/Project (master|MERGING)
$ git add .
warning: LF will be replaced by CRLF in DerivedData/methodAttributes.json.
The file will have its original line endings in your working directory

Ayoub@Ayoub-PC MINGW64 ~/Desktop/my_4D_project.4dbase/Project (master|MERGING)
$ git commit -m "conflict solved for meth_alert_2 keeping both codes"
[master e9446dc] conflict solved for meth_alert_2 keeping both codes

Ayoub@Ayoub-PC MINGW64 ~/Desktop/my_4D_project.4dbase/Project (master)
$ git status
On branch master
Your branch is ahead of 'origin/master' by 2 commits.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean

Ayoub@Ayoub-PC MINGW64 ~/Desktop/my_4D_project.4dbase/Project (master)
$ git push
git: 'credential-osxkeychain' is not a git command. See 'git --help'.
Enumerating objects: 26, done.
Counting objects: 100% (26/26), done.
Delta compression using up to 8 threads
Compressing objects: 100% (14/14), done.
Writing objects: 100% (14/14), 1.35 KiB | 1.35 MiB/s, done.
Total 14 (delta 7), reused 0 (delta 0)
remote: Resolving deltas: 100% (7/7), completed with 4 local objects.
To https://github.com/ayoub-kl/4D_git_tuto.git
   59880bc..e9446dc  master -> master
```

Once this is done, user 2 can simply run **git pull** to get the new changes made by user 1, so that both users can now have the same code in meth_alert_2.

```
Ayoub@Ayoub-PC MINGW64 ~/Desktop/my_4D_project.4dbase - Copie/Project (master)
$ git pull
remote: Enumerating objects: 26, done.
remote: Counting objects: 100% (26/26), done.
remote: Compressing objects: 100% (7/7), done.
remote: Total 14 (delta 7), reused 14 (delta 7), pack-reused 0
Unpacking objects: 100% (14/14), done.
From https://github.com/ayoub-kl/4D_git_tuto
   59880bc..e9446dc  master     -> origin/master
Updating 59880bc..e9446dc
Fast-forward
 DerivedData/methodAttributes.json | 4 ++--
 Sources/Methods/meth_alert_2.4dm  | 6 +++++-
 2 files changed, 7 insertions(+), 3 deletions(-)
```

Next let's take a quick look at the first conflict merging method via git mergetool. Supposing that a form conflict was created and needs solving (the same way the method conflict was created, take a look at steps 2, 3, 4, 5). Both user 1 and user 2 modified the 4D form "frm_ok_1" containing 1 ok button, take a look at the following:

User 1:

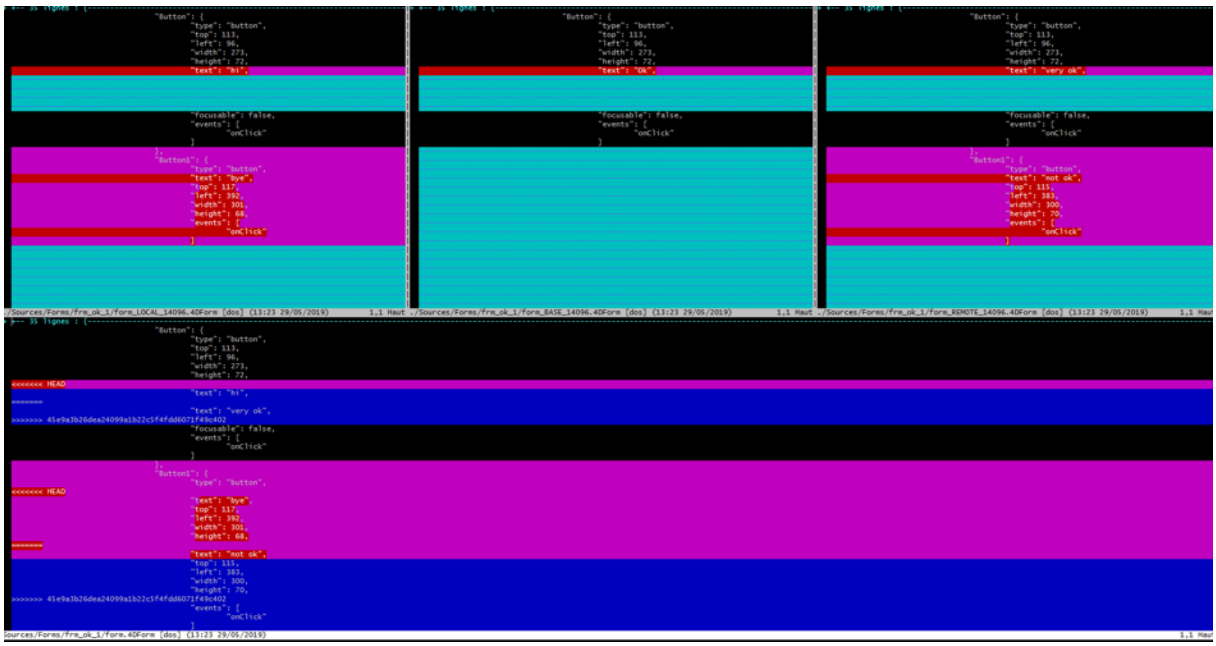


User 2:

Following the steps from 2 to 5 the following step would be using git mergetool to solve this conflict.

```
Ayoub@Ayoub-PC MINGW64 ~/Desktop/my_4D_project.4dbase - Copie/Project (master|MERGING)
$ git mergetool

This message is displayed because 'merge.tool' is not configured.
See 'git mergetool --tool-help' or 'git help config' for more details.
'git mergetool' will now attempt to use one of the following tools:
opendiff kdiff3 tkdiff xxdiff meld tortoisemerge gvimdiff diffuse diffmerge ecmerge p4merge araxis bc codecompare emerge vimdiff
warning: LF will be replaced by CRLF in DerivedData/formAttributes.json.
The file will have its original line endings in your working directory
warning: LF will be replaced by CRLF in DerivedData/methodAttributes.json.
The file will have its original line endings in your working directory
Merging:
Sources/Forms/frm_ok_1/form.4DForm

Normal merge conflict for 'Sources/Forms/frm_ok_1/form.4DForm':
  {local}: modified file
  {remote}: modified file
Hit return to start merge resolution tool (vimdiff): |
```

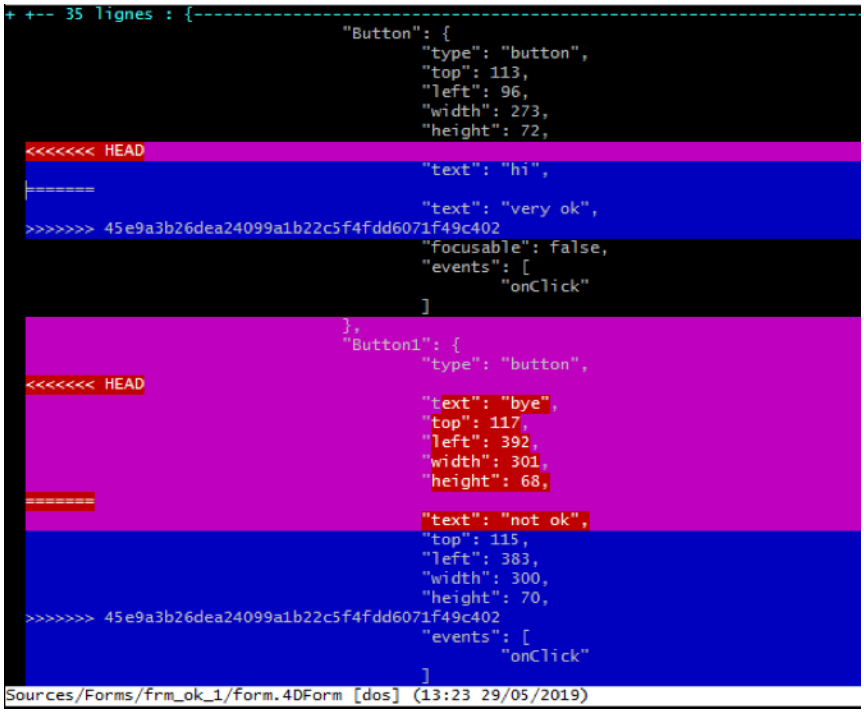
Then press enter:



The screen is divided into 4 zones:

- Upper left: Json file representing the local form changes.
- Upper right: Json file representing the remote form changes.
- Upper center: Json file representing the original form.
- Lower zone: Final json file that is going to be kept locally after the conflict is solved.

In order to solve the conflict, take a look at the lower zone (modification can be made there):



Same as the earlier example (methods), notice the json is delimited by tags (<<<<<HEAD and >>>>>>>> commit_number). From the head tag to “======” representing the local changes, with the remote changes starting from “======” to the tag with the commit number.

Simply keep the part of the wanted part of text while removing the tags from the file. This is how the final result will look like:

```
],
"pageFormat": {
    "paperName": "A4",
    "paperWidth": "595pt",
    "paperHeight": "841pt"
},
"pages": [
    null,
    {
        "objects": {
            "Button": {
                "type": "button",
                "top": 113,
                "left": 96,
                "width": 273,
                "height": 72,

                "text": "hi",

                "focusable": false,
                "events": [
                    "onClick"
                ]
            },
            "Button1": {
                "type": "button",

                "text": "bye",
                "top": 117,
                "left": 392,
                "width": 301,
                "height": 68,

                "events": [
                    "onClick"
                ]
            }
        }
```

Modifications made:

- Tags removed
- On version kept

Demo 3: Making use of branches & stashing

Branches are no doubt one of the key features of GIT, this demo will make explain basic branch features and operations.

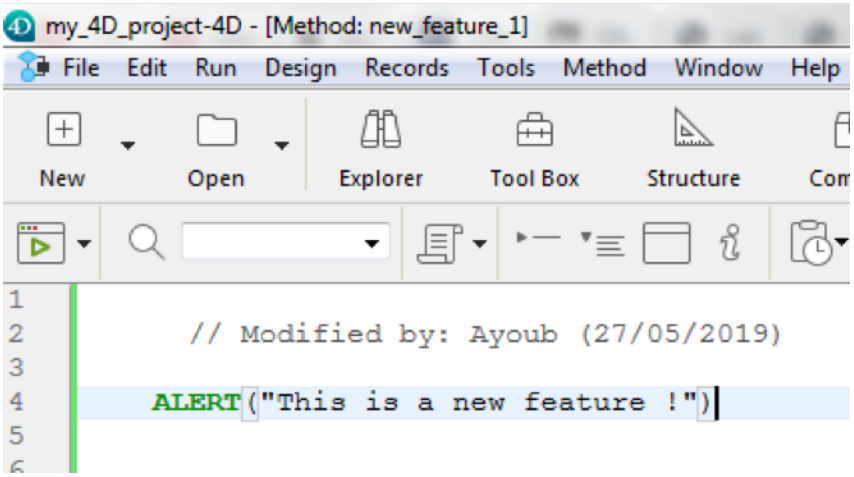
1. Go to the project folder and launch a git bash (right click)
2. Create a new branch named feature1

```
Ayoub@Ayoub-PC MINGW64 ~/Desktop/my_4D_project.4dbase/Project (master)
$ git branch feature1

Ayoub@Ayoub-PC MINGW64 ~/Desktop/my_4D_project.4dbase/Project (master)
$ git checkout feature1
Switched to branch 'feature1'

Ayoub@Ayoub-PC MINGW64 ~/Desktop/my_4D_project.4dbase/Project (feature1)
$
```

Note that the name of the current branch changes when checking out.

- **Git branch branch_name**: Creates a new branch named feature1
- **Git checkout branch_name**: a git command designed to allow navigation between git branches. Checking out a specific branch updates the working directory to match the version that is stored in that specific branch.
- **Git checkout –b branch_name**: Will create a new branch and navigate to it (checkout)

3. Create a new method named “new_feature_1”, then type in some code



4. To save our changes both locally and remotely, take a look at the following:

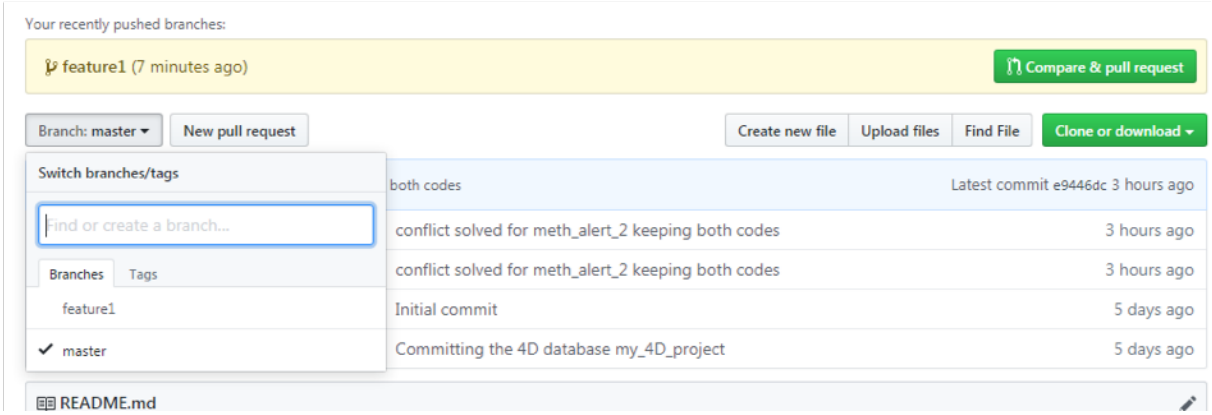
```
Ayoub@Ayoub-PC MINGW64 ~/Desktop/my_4D_project.4dbase/Project (feature1)
$ git add .
warning: LF will be replaced by CRLF in DerivedData/methodAttributes.json.
The file will have its original line endings in your working directory
warning: LF will be replaced by CRLF in Sources/catalog.4DCatalog.
The file will have its original line endings in your working directory
warning: LF will be replaced by CRLF in DerivedData/1/formAttributes.json.
The file will have its original line endings in your working directory

Ayoub@Ayoub-PC MINGW64 ~/Desktop/my_4D_project.4dbase/Project (feature1)
$ git commit -m "committing method new_feature_1 in feature1 branch"
[feature1 7bd8e22] committing method new_feature_1 in feature1 branch
 4 files changed, 13 insertions(+), 2 deletions(-)
 create mode 100644 DerivedData/1/formAttributes.json
 create mode 100644 Sources/Methods/new_feature_1.4dm
```

```
Ayoub@Ayoub-PC MINGW64 ~/Desktop/my_4D_project.4dbase/Project (feature1)
$ git push --set-upstream origin feature1
git: 'credential-osxkeychain' is not a git command. See 'git --help'.
Enumerating objects: 15, done.
Counting objects: 100% (15/15), done.
Delta compression using up to 8 threads
Compressing objects: 100% (8/8), done.
Writing objects: 100% (10/10), 1.04 KiB | 213.00 KiB/s, done.
Total 10 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
remote:
remote: Create a pull request for 'feature1' on GitHub by visiting:
remote:      https://github.com/ayoub-kl/4D_git_tuto/pull/new/feature1
remote:
To https://github.com/ayoub-kl/4D_git_tuto.git
 * [new branch]      feature1 -> feature1
Branch 'feature1' set up to track remote branch 'feature1' from 'origin'.
```

Next thing would be pushing the changes to the remote repository, but since there is no remote branch named feature1 yet, it can be done as follows:

Git push –set-upstream origin feature1: Will push the local changes to our remote repository, creating a new branch named feature1 and linking it to its local counterpart.

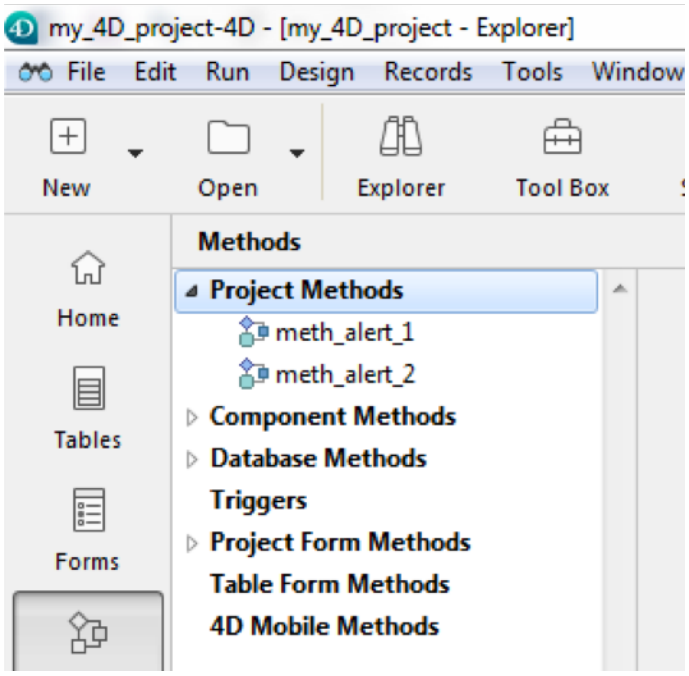
Notice that a new branch has been created in the remote repository. Feature1 contains the same files as the master branch + the newly created method (new_feature_1).



Also, notice that switching to the master branch, the newly created method will not exist.

```
Ayoub@Ayoub-PC MINGW64 ~/Desktop/my_4D_project.4dbase/Project (feature1)
$ git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.
```

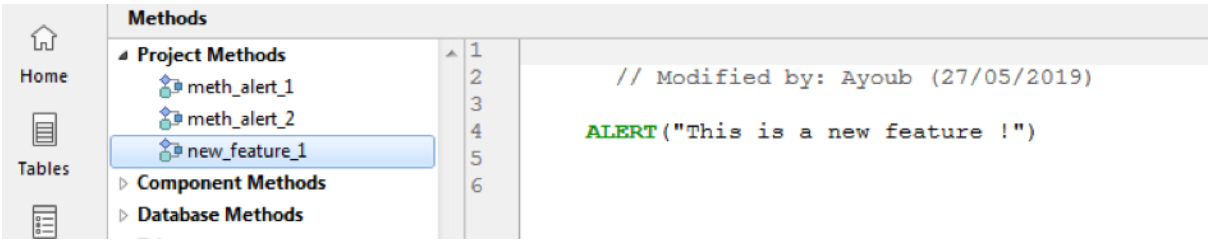
The new_feature_1 method doesn't exist yet in the master branch. Making changes and saving them on master won't affect the feature1 branch. The only way to get these changes from one to another would be merging. This way, each branch only contains the content that it is supposed to have and nothing else.



5. Let's merge the changes in feature1 with the master branch:

```
Ayoub@Ayoub-PC MINGW64 ~/Desktop/my_4D_project.4dbase/Project (master)
$ git merge feature1
Updating e9446dc..7bd8e22
Fast-forward
 DerivedData/1/formAttributes.json | 3 +++
 DerivedData/methodAttributes.json | 3 +++
 Sources/Methods/new_feature_1.4dm | 6 ++++++
 Sources/catalog.4DCatalog         | 3 +--
 4 files changed, 13 insertions(+), 2 deletions(-)
 create mode 100644 DerivedData/1/formAttributes.json
 create mode 100644 Sources/Methods/new_feature_1.4dm
```

Git merge feature1: Will merge changes from feature1 into master. Notice that the method new_feature_1 has been added to 4D methods.

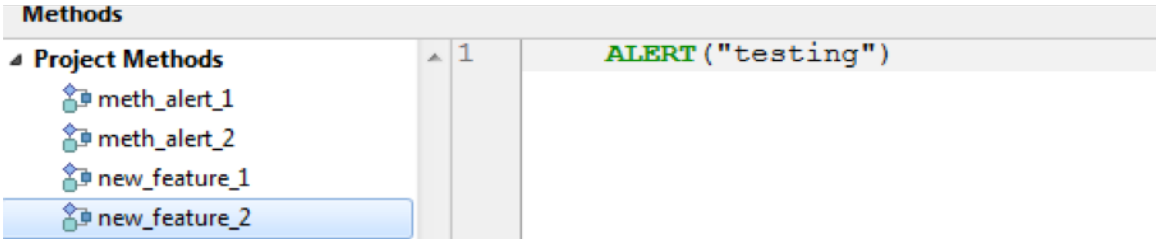


Important note: Imagine that a developer had to skip the last two steps (4 and 5), which could happen for many reasons (not ready to commit, incomplete feature code ….) and then switched to master branch in order to fix some kind of issue. This looks okay at first, but once realizing that all the methods and files from the feature branch

are now accessible from the master branch, which could eventually end up committed into the wrong branch causing production failures and conflicts.

6. Checkout feature1 branch then creates a new method in 4D named new_feature_2. Switch back to the master branch and notice that the method still exists which contradicts the whole concept of branching

```
Ayoub@Ayoub-PC MINGW64 ~/Desktop/my_4D_project.4dbase/Project (master)
$ git checkout feature1
Switched to branch 'feature1'
M       DerivedData/methodAttributes.json

Ayoub@Ayoub-PC MINGW64 ~/Desktop/my_4D_project.4dbase/Project (feature1)
$ git checkout master
Switched to branch 'master'
M       DerivedData/methodAttributes.json
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)
```



This can be easily prevented by using git stash whenever switching between branches becomes necessary while pending changes still are not saved. git stash records the current state of the working directory + the index, saving the local modifications.

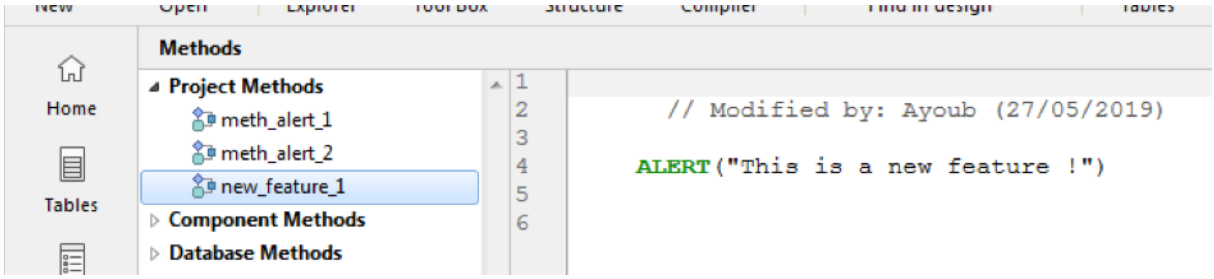
7. Note that **git stash** will only save the files that are being tracked by GIT, running a **git add** will be necessary beforehand.

```
Ayoub@Ayoub-PC MINGW64 ~/Desktop/my_4D_project.4dbase/Project (feature1)
$ git status
On branch feature1
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        Sources/Methods/new_feature_2.4dm

nothing added to commit but untracked files present (use "git add" to track)
```

```
Ayoub@Ayoub-PC MINGW64 ~/Desktop/my_4D_project.4dbase/Project (feature1)
$ git add Sources/Methods/new_feature_2.4dm

Ayoub@Ayoub-PC MINGW64 ~/Desktop/my_4D_project.4dbase/Project (feature1)
$ git stash
Saved working directory and index state WIP on feature1: 7bd8e22 committing method new_feature_1 in feature1 branch
```

8. Notice that upon switching to master branch once again, the method doesn't exist anymore since it was stashed.

```
Ayoub@Ayoub-PC MINGW64 ~/Desktop/my_4D_project.4dbase/Project (feature1)
$ git checkout master
Switched to branch 'master'
Your branch is ahead of 'origin/master' by 1 commit.
```



9. Once ready to get back to working on the pending feature, switch to feature branch then execute **git stash apply to** get the 4D method new_feature_2 back.

```
Ayoub@Ayoub-PC MINGW64 ~/Desktop/my_4D_project.4dbase/Project (master)
$ git checkout feature1
Switched to branch 'feature1'
M       DerivedData/methodAttributes.json

Ayoub@Ayoub-PC MINGW64 ~/Desktop/my_4D_project.4dbase/Project (feature1)
$ git stash apply
On branch feature1
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   Sources/Methods/new_feature_2.4dm

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   DerivedData/methodAttributes.json
        modified:   Sources/catalog.4DCatalog
```

10. Deleting the feature1 branch can be done easily: **Git branch –d feature1**. Note that it is necessary to be on a different branch before deleting.

```
Ayoub@Ayoub-PC MINGW64 ~/Desktop/my_4D_project.4dbase/Project (master)
$ git branch -d feature1
Deleted branch feature1 (was 7bd8e22).
```

Conclusion

This technical note provided general information about GIT architecture and its features, in addition to detailed step by step lifelike situations and conflicts that a developer from any background may encounter, adapted to a specific 4D context and way of use. By introducing project mode, 4D revealed new horizons for its developers that are looking to incorporate useful and powerful management tools like GIT, allowing it to quickly become an indispensable tool.

Most of the basic GIT commands were put to use in a multitude of scenarios portraying what a 4D developer will live to confront in a work environment, yet more advanced commands and technics are available to fulfill every little detail and quench the tech-savvy thirst of every senior developer out there.

Implementing version control with the new 4D file system will exploit to a maximum the freshly offered flexibility and will embody a foundation stone to a new era of collaboration.

Resources

Convert an existing database into a Project. (2019, April 24). Retrieved May 22, 2019, from https://blog.4d.com/convert-an-existing-database-into-a-project/

4D PROJECT: EMBRACE A NEW ERA OF COLLABORATION. (2019, April 24). Retrieved May 23, 2019, from https://blog.4d.com/4d-projects-embrace-the-new-era-of-collaboration/

Git commands. (n.d.). Retrieved May 24, 2019, from https://git-scm.com/doc

Git workflow. (n.d.). Retrieved May 24, 2019, from https://stackoverflow.com

Git Credential Caching on Mac OS X. (2013, August 5). Retrieved May 22, 2019, from http://tech.lds.org/wiki/Git_Credential_Caching_on_Mac_OS_X