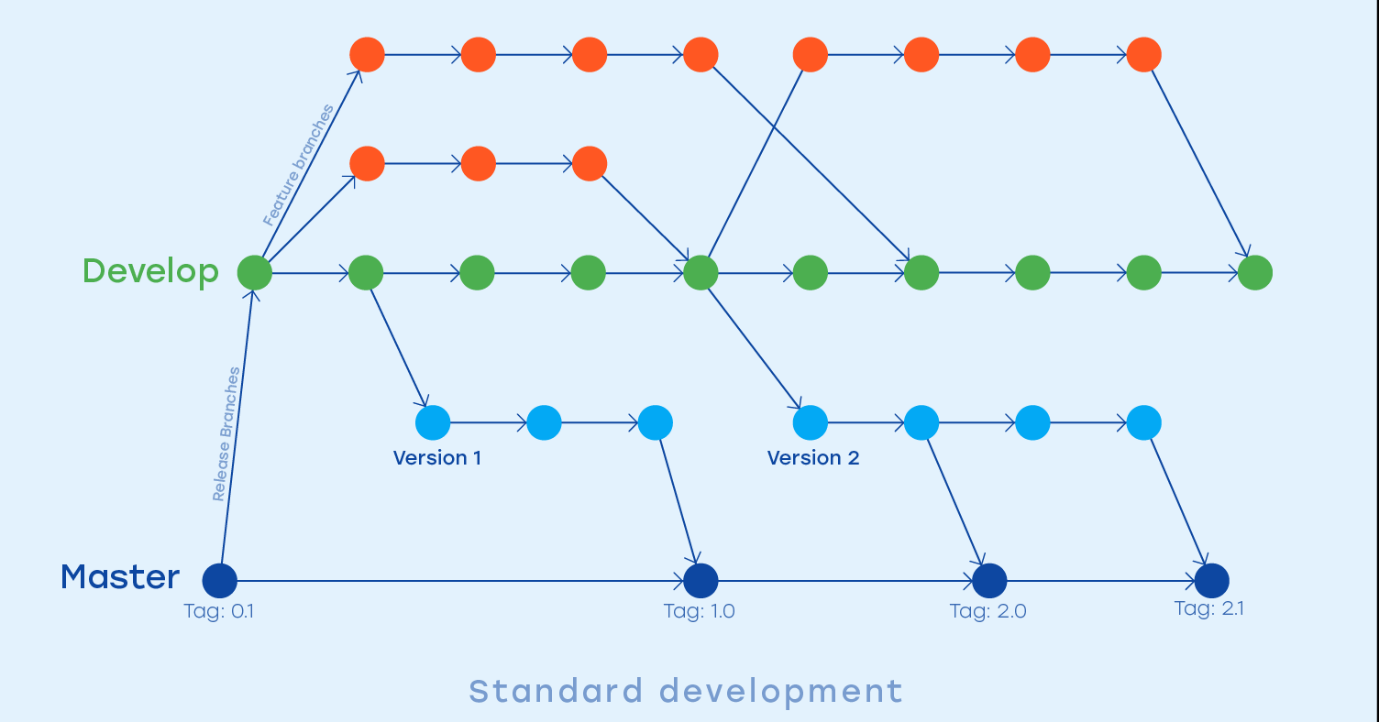
Git: git-use

A guide to development with a healthy git-use



Conducted by

Date

Archive title

Glyn Marcus Leine

04-05-2020

A guide to development with a healthy git-use

Table of Contents

[Understanding source control 3](#_Toc39575782)

[Why use source control? 3](#_Toc39575783)

[Data safety 3](#_Toc39575784)

[Collaboration organised 3](#_Toc39575785)

[Maintenance and organisation 3](#_Toc39575786)

[How does source control work? 3](#_Toc39575787)

[Using source control 3](#_Toc39575788)

[Terminology 4](#_Toc39575789)

[Git-use 5](#_Toc39575790)

[Branches 5](#_Toc39575791)

[Working with branches 5](#_Toc39575792)

[Commit messages 5](#_Toc39575793)

[Step-by-step with GitFork 6](#_Toc39575794)

# Understanding source control

Git is one of the many different software solutions that offer a source control service. But what is source control?

Source control is the practice of backing up work on a unified location and tracking the history of all the files. This unified location could be the computer of one of the developers or a remote server like the cloud. GitHub is one of the most popular publicly available git hosts.

## Why use source control?

### Data safety

There are several reasons why a team would choose to use source control. One big reason is because of data safety.

Recording all history means you can easily travel back in time if anything goes wrong. You can either continue to work from that part in history or even just change a few things and merge it back with the rest of the “future” work.

Another data safety feature source control offers is that everything can be stored off-site. Meaning that in case all developers machines fail, the off-site server will still have a backup of the data.

### Collaboration organised

One more service source control provides that is crucial to development with more than one developer is that everything is stored in that unified location. This means that there is always one current version that is kept track off. The latest version is no longer spread out over multiple machines and synchronised by sending each other files through download links.

### Maintenance and organisation

Due to the history recording feature, we also get another extra benefit: versions.

Throughout the development time, we can appoint specific points in the history as a stable version. Meaning that if we ever need to test anything or supply something stable to someone we could fetch the latest version off of the server and hand that over without having to worry about last-minute patches for those newly added features. Developers can just continue developing without fear of breaking the build.

## How does source control work?

As mentioned before source control work by recording the full history of files. Storing a copy of each file after they have changed however takes a lot of storage space and is also a lot of data to upload and download. So instead source control works with changes.

At every upload to the server/unified location, the source control service will scan your files for the changes it can recognise. These changes are then uploaded, and the source control can figure out how to merge all changes into a single file by itself. These changes can easily be mixed and matched, so making a new file from changes from multiple users is a non-issue for the source control as long as changes don’t overlap.

# Using source control

Source control can be implemented in several different ways depending on the back-end software. In this case, we will focus on one specific back end: Git.

## Terminology

Before we start getting into how to use git we should start with some minor terminology:

* **Repository**: a repository is a source control aided project.
* **Clone**: cloning means you download the data on a repository to your local machine so you can work with it.
* **Remote**: the remote is another word for the unified location.
* **Fetch**: update your local view to see any changes made on the remote.
* **Pull**: pulling from the remote means downloading the latest version from the remote to update your local version.
* **Change**: a change is a certain difference between the file on your local version and the remote
* **Stage**: staging changes means that they are ready to be committed. This is mostly used to differentiate between changes that aren’t ready to be uploaded and changes that are ready.
* **Commit**: when you commit changes that means that those changes are ready to be uploaded to the server. Committed changes can no longer be edited unless committed again. A commit also appoints a certain point of time in the history of the repository.
* **Push**: pushing your commits is the action of taking all local commits and uploading them to the remote, forcing the remote version to update.
* **Conflict**: when creating changes and committing them you can get into conflicts. This means both you and another team member have made a change in the same location, the source control does not know how to merge your two versions of the file into 1 version. You now need to decide and tell it what to do.
* **Branch**: branches are a sperate version of the project developed alongside the remote. In fact, when you clone a repository you also create your own local branch. You can create separate branches yourself as well.
* **Merge**: merging is the act of taking the changes in two branches and creating one unified version from the two branches. This could trigger conflicts if both branches had changes in the same locations.
* **Fast-forward**: fast-forwarding is the ability to merge branches without the need for human interaction. This is only possible if there are not any conflicts.
* **Tag**: a tag is well... a tag... it is a tag you can attach to a certain commit flagging it as a stable version of the project.
* **Master Branch**: the master branch is the main branch that holds all the stable versions of the project.

## Git-use

### Branches

For a healthy git-use we keep a set of default branches:

* The master branch: as explained above
* The develop branch: here we can add changes that might not be stable allowing us to further the project

Next to those two default branches, we can have other branches as well:

* A version branch: this is a branch that just holds a specific stable version.
* A feature branch: this is a temporary branch that can be used to develop a feature without having to constantly be synchronised with all the changed on the develop branch.

### Working with branches

The stable branches (master and version branches) are not to be committed to, the only way to add differences to these branches is by merging unstable branches into them.

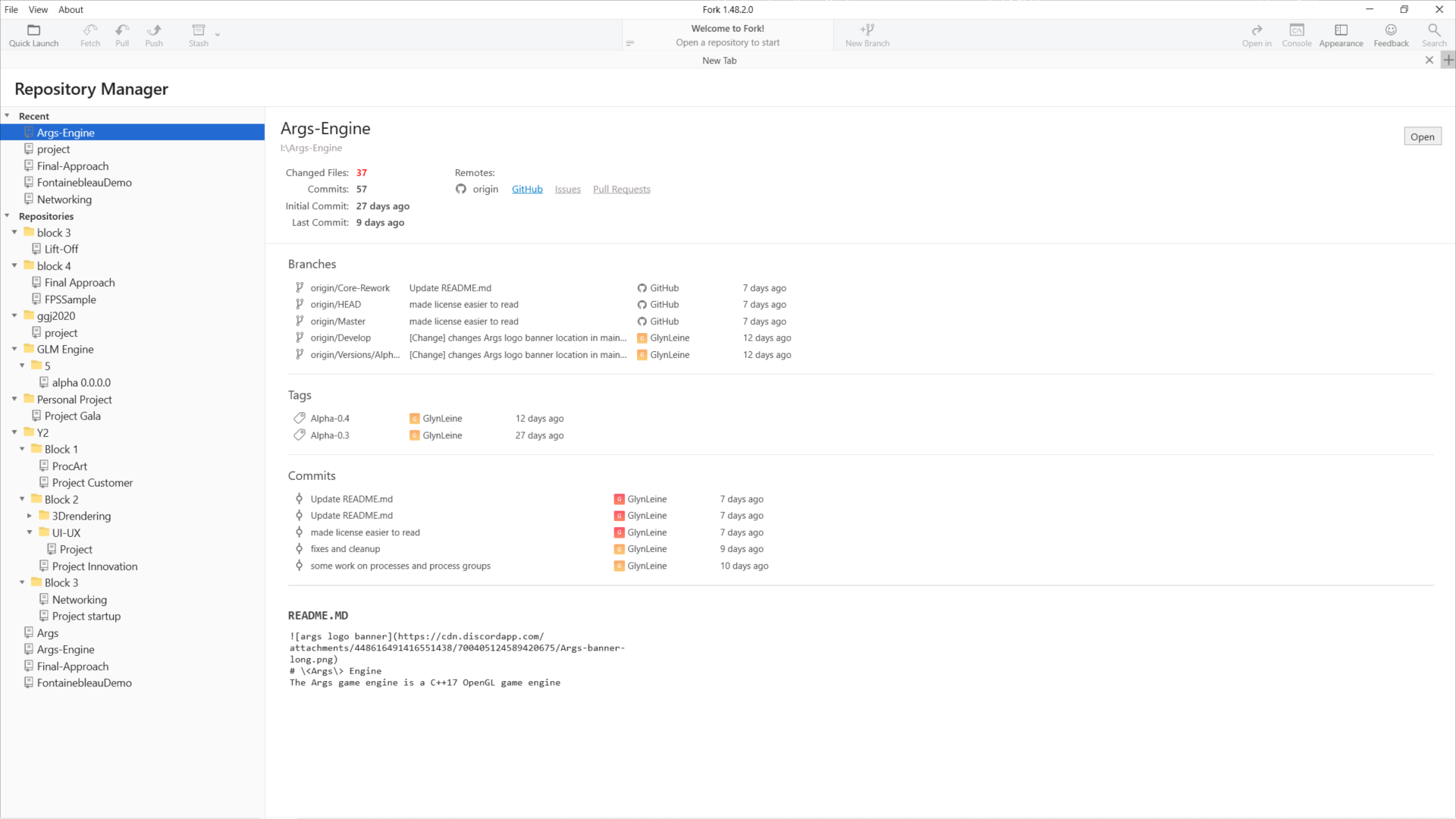
The unstable branches (develop and feature branches) can be freely pushed and pulled from and committed to.

### Commit messages

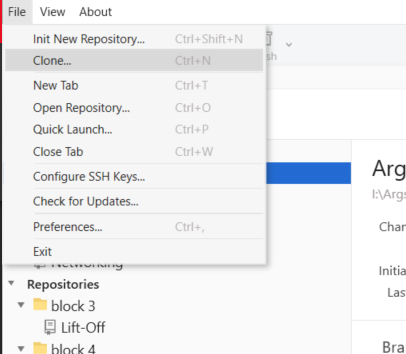
When you commit changes, you can add a commit message. This message exists out of a title and a description. For the title, it is advised to write a short title that describes the changes you made. For the description adhere to the following flags:

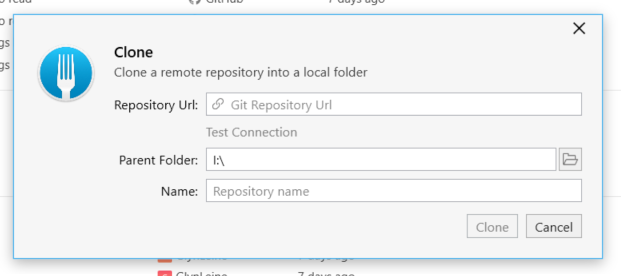
* [Add] this is how you flag a new file or feature
* [Change] this is how you flag a feature that has changed
* [Fix] this is how you flag a fix for a bug
* [Remove] this is how you flag a feature or file has been removed
* [Rename] this is how you flag a feature or file has been renamed
* [WIP] this flag can be added in combination with any of the above flags to communicate that the change is either instable or bound to change in the near future.

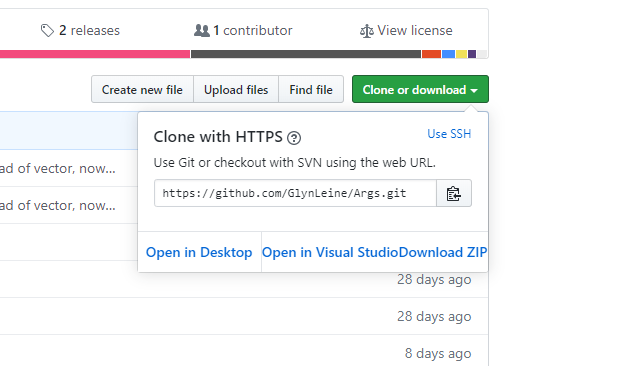
# Step-by-step with GitFork

When first opening GitFork you will be greeted with the following screen:

In order to clone your first repository please go to File>Clone…



This brings up the following screen: 

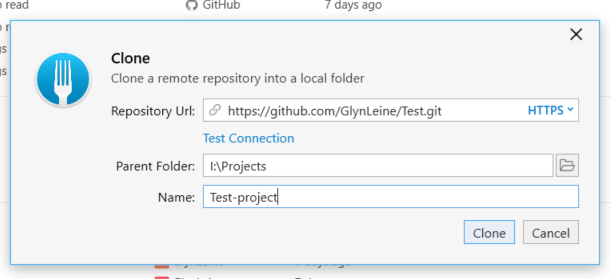
The Repository URL is a URL key that can be found on GitHub

Copy the key into fork.

By selecting the parent folder, you can decide where the project will be downloaded to, this will be your new work location.

The name that you can decide on is the name your local copy will be named; the entire project will be downloaded into a folder of that name in the parent folder.

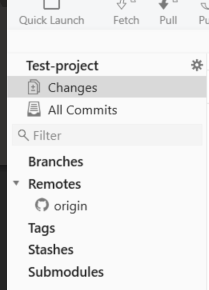
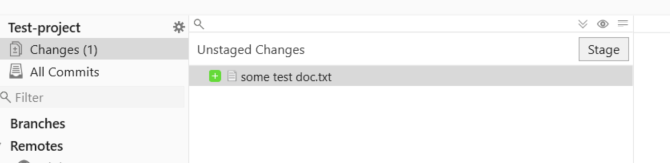
After everything has been setup click Clone and fork will create your own local copy of the repository.



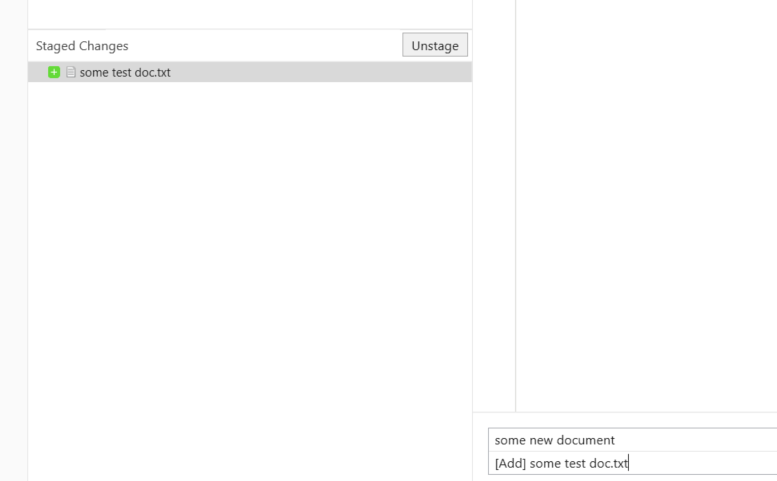
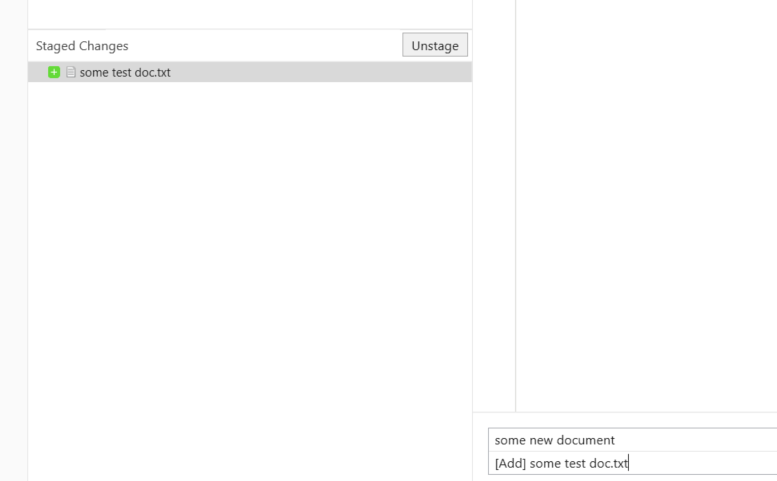
You can now start working on the project.

As you make changes, they will show up in the changes tab.

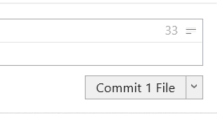
The changes tab can be found here:

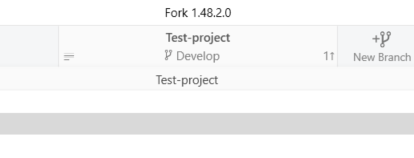
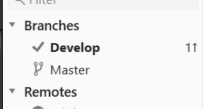
By clicking stage you can flag the selected file/files to be ready to be committed.  
(you can also click the stage all button to stage everything, the stage all button is the arrow above the stage button)

Once staged you can add a commit title and description: 

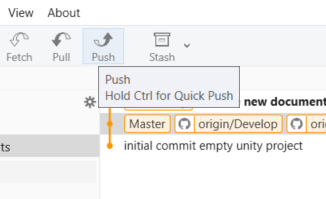
Press the commit button to commit the files



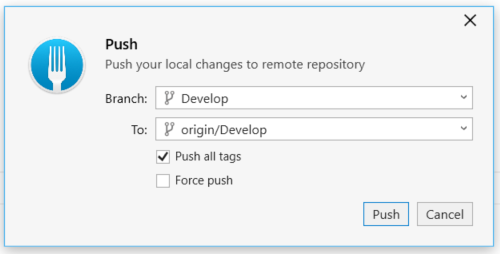
As you commit files fork will show you your current local version is in front of the remote version by showing an arrow



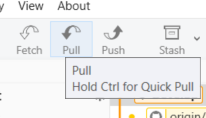
By pressing the push button, you can push your commits to the remote, forcing it to take your changes.



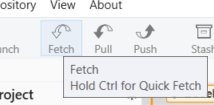
When met with the following dialogue box click push again to complete the push



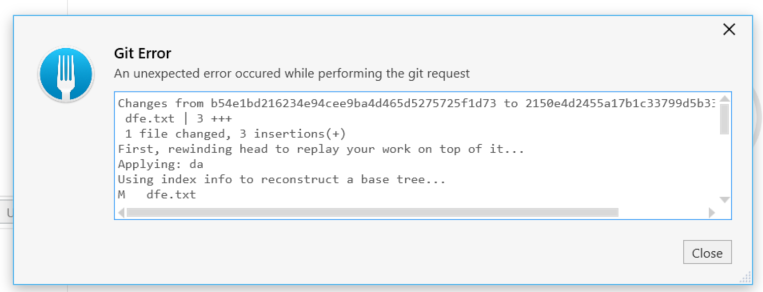
If instead there is a downward arrow as well as an upward arrow, you should press the pull button first before pushing



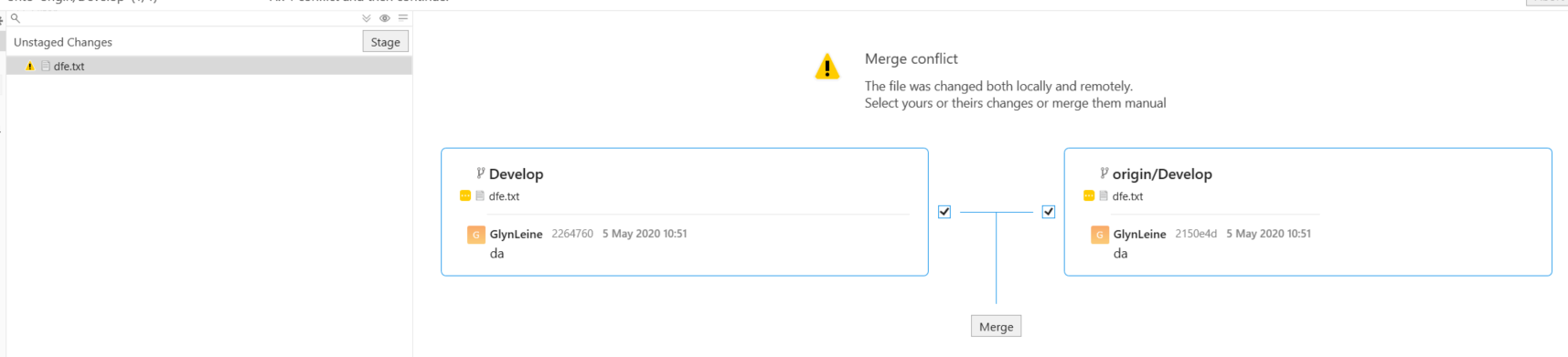
If there are any changes that were made on the remote that you do not see on fork yet, you can click fetch to update your view. It is generally advised to always press fetch before pushing.



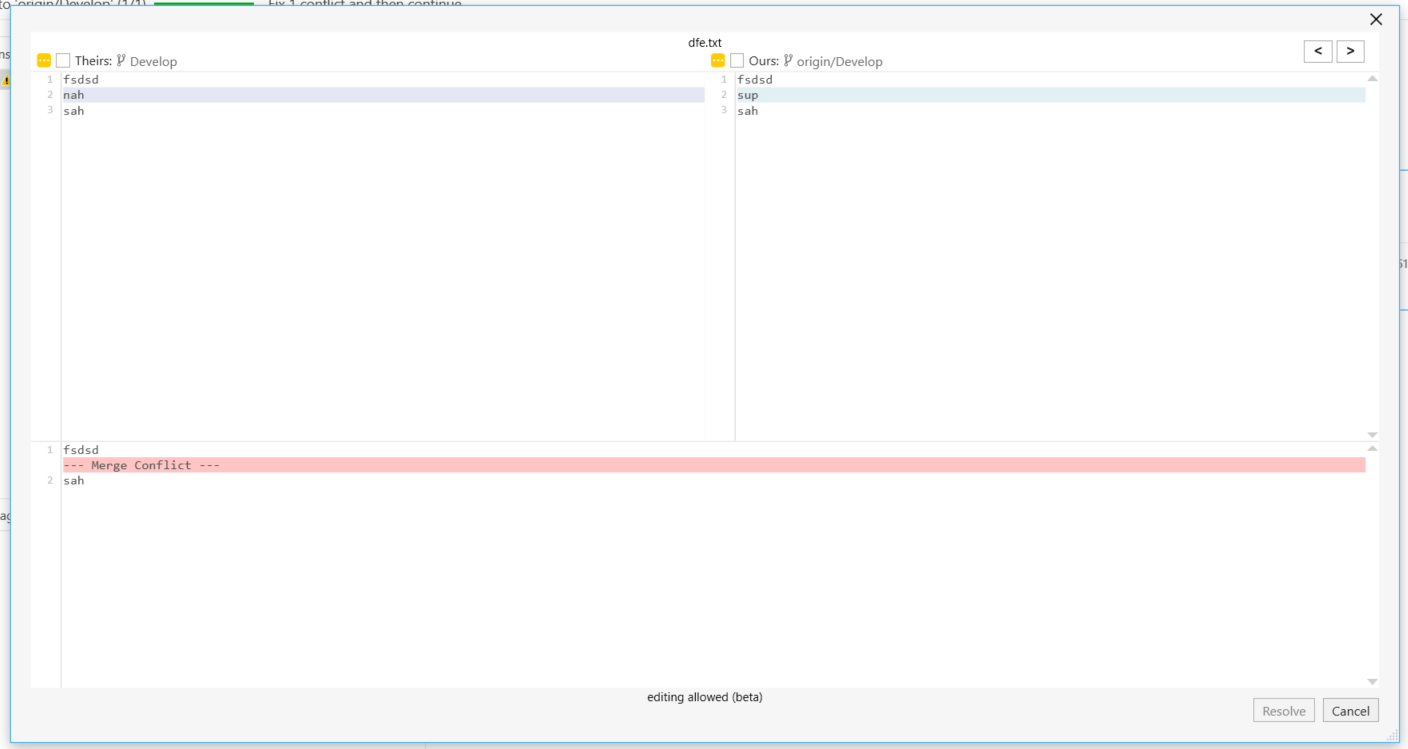
Pulling after having made changes that are in similar locations as changes one of your team members may have pushed can result in conflicts. You will be greeted with this error box:



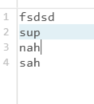
When closing the box, you will see this



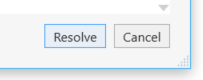
By clicking merge you can get to see both changes and decide how the changes should be merged by hand.



You can select either one of the top two to be marked as the right version going forward, or you could edit the version below to be what it should have been if merged properly:



Then click the resolve button to resolve the conflict:



After this you can commit these new changes and pull and push as normal again.