Modding Git Guide

If you are modding as a team, or any large software project, you should be using a Git. While it can be daunting at first, it is well worth it in the long run. This guide is designed to be short and to the point, as well as minimising technical language so that anyone can read it and get started. I hope you find it useful!

- Written by Zankoas, maintained by Alpinia

Index

1. Introduction

If you aren't interested in the how, why or more advanced stuff and just want a simple guide to the basics, skip to 4. Setup Guide and 5. Git Basics.

While I am not the first and will not be the last person to write a guide for using a Git, none I have found are focused around modding. Modding presents unique challenges to using a Git as people are less reliable and less experienced than in normal software development. This does make using a Git harder, but I don't for a moment think that outweighs the benefits.

This won't be a flawless guide, nor explain the most efficient way to use a Git, but it should help you reach the minimum level needed to start working and let you work from there. Modding is very much a learning experience, with or without a Git.

What is a Git you ask? It is a bit of software used to manage multiple people working on the same project. It does this by storing changes (called 'commits') as the difference between the current and the previous version rather than as a whole new version (technically it's more complicated than this, see the Q&A in the appendix if you want the details).

This lets you see who has made a change, switch between versions and merge different versions together without needing to store every single version in full.

If you'd like to give feedback or make suggestions you can message me on discord at: *zankoas*, or message *alpinia* who helps maintain this resource. You can also email the Kaiserreich team - who own this document - at kaiserreichofficial@gmail.com.

As you may or may not have worked out, I am primarily a modder for HOI4, but this guide is equally applicable to other PDX games. In fact, apart from a few parts regarding .mod files, this guide works great for any mod for any game.

2. Dictionary

- Git - Software used to manage multiple people working on the same project
- GitLab/GitHub - Both companies that host your Git remotely, so other people can access it

- GitExt - Short for Git Extensions, a client used manage your Git with an easy to use interface
- KDiff3 - A tool used to manage merges
- Commit - Making changes to the Git, think of it like clicking ‘confirm’ when asked if you are sure you want to make changes
- Push - Taking changes you made locally and applying them to the online version of your Git (the one hosted by GitLab for example)
- Pull - Taking changes someone else made to the online version of the Git and applying them to your local Git

3. Why use a Git?

Let's start with why you wouldn’t want to use a Git:

1. Takes some effort to set up
2. Certain types of mistakes (see merges later on) can be harder to fix
3. Can be intimidating for a new user
4. Takes longer to download the first time

That’s really about it, as you can see 1-3 are just about being careful and having a useful guide (hint hint), while 4 is just a minor inconvenience.

So, reasons to use a Git:

1. Able to automatically merge changes from everyone working on a project
2. Able to almost instantly switch between *any* version of the project that has *ever* existed
3. Can check who made any change, when they made it, what the change was and what they said they did
4. Able to easily and quickly update after it is set up
5. Ensures that no one person is storing the mod on their personal computer (accidents happen after all)

As I’m sure you can see with just a single person a Git is useful, but with four people, seven people, twelve people or more, it becomes a must. Being able to merge changes automatically, see who made what change and updating files easily becomes a lifesaver in a team environment.

4. Setup Guide

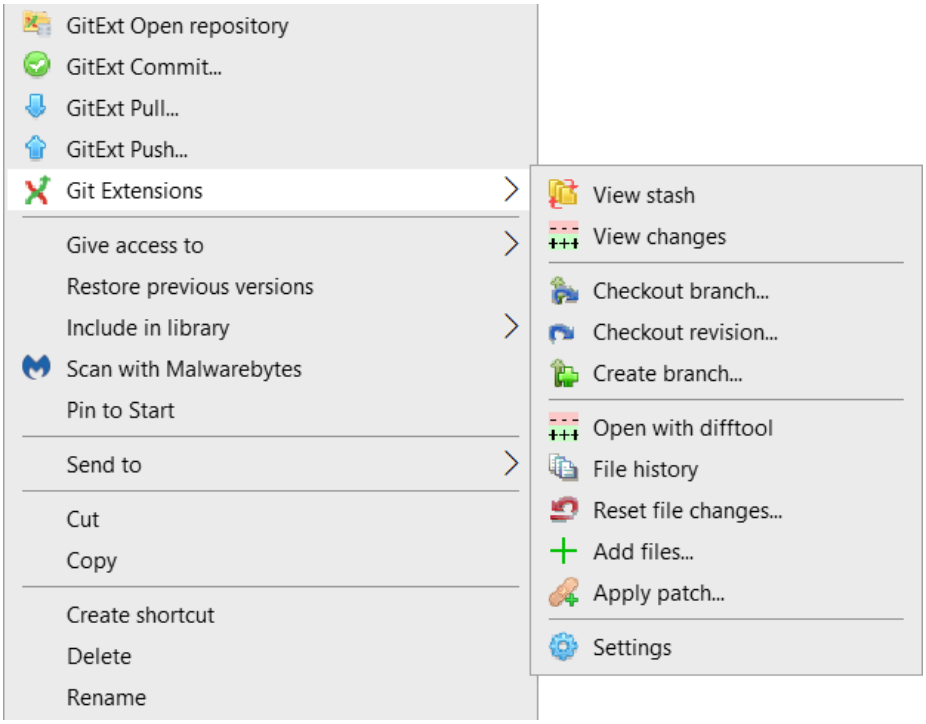
This is my preferred setup for using a Git, it is by no means the only software that you can use but the basic concepts in here will be the same whatever software you use. Note this is a windows setup guide. For more details on other operating systems, the Q&A in the appendix.

1. Download and install Git: https://git-scm.com/downloads
    - If you are unsure of the settings to choose when asked, I suggest the following:
    - Select Components
    - Choosing the default editor used by Git
        i. This requires Notepad++ be installed on your system, if it isn't you can download and install it from here:
        ii. https://notepad-plus-plus.org/downloads/
        iii. Equally, if you'd rather not use Notepad++, feel free to select a different text editor
    - Adjusting the name of the initial branch in new repositories
        i. Note that this will be changing in the future, so it may be worth adjusting now rather than later
    - Adjusting your PATH environment
    - Choosing HTTPS transport backend
    - Configuring the line ending conversions
    - Configuring the terminal emulator to use with Git Bash
    - Choose the default behavior of `git pull`
        i. See 5. Git Basics section 'Pull…' for more information on what this means
    - Choose a credential helper
    - Configuring extra options
    - Configuring experimental options
2. Download and install KDiff3: https://download.kde.org/stable/kdiff3/
    - Sort by 'Last modified' to find the latest version, will be a .exe file
3. Download and install Git Extensions: https://github.com/gitextensions/gitextensions/releases
    - Look under 'Assets' at the bottom of the latest version to get the installer, will be a .msi file
    - If you are unsure of the settings to choose when asked, I suggest the following:

    - Select SSH Client
    - Telemetry privacy policy
4. Go to where you want to have the Git stored on your PC, for HOI4 modding that would likely be: \Documents\Paradox Interactive\Hearts of Iron IV\mod
5. Right click and select ‘GitExt Clone’
6. First line is address of your mod, normally: https://github.com/Kaiserreich/Kaiserreich-4-Development
    - If you are someone joining an existing Git then ask the author for this address, if it is private then ask them to give you access to it
7. Second line should auto fill as the folder you right clicked in
8. Third line should be the folder you want to create in there, unless specifically told otherwise, I would avoid changing this from the suggested name
9. Hit Clone and wait for it to download and setup
    - For HOI4 modding you will need to put a .mod file in \Documents\Paradox Interactive\Hearts of Iron IV\mod in order to tell the game you have added a mod, most teams put one inside the Git folder so just copy (not move) that one, though if in doubt ask
    - Note that accents aren’t supported by .mod files, so if you have an accent in your username you’ll need to clone the Dev Build into a different folder

5. Git Basics

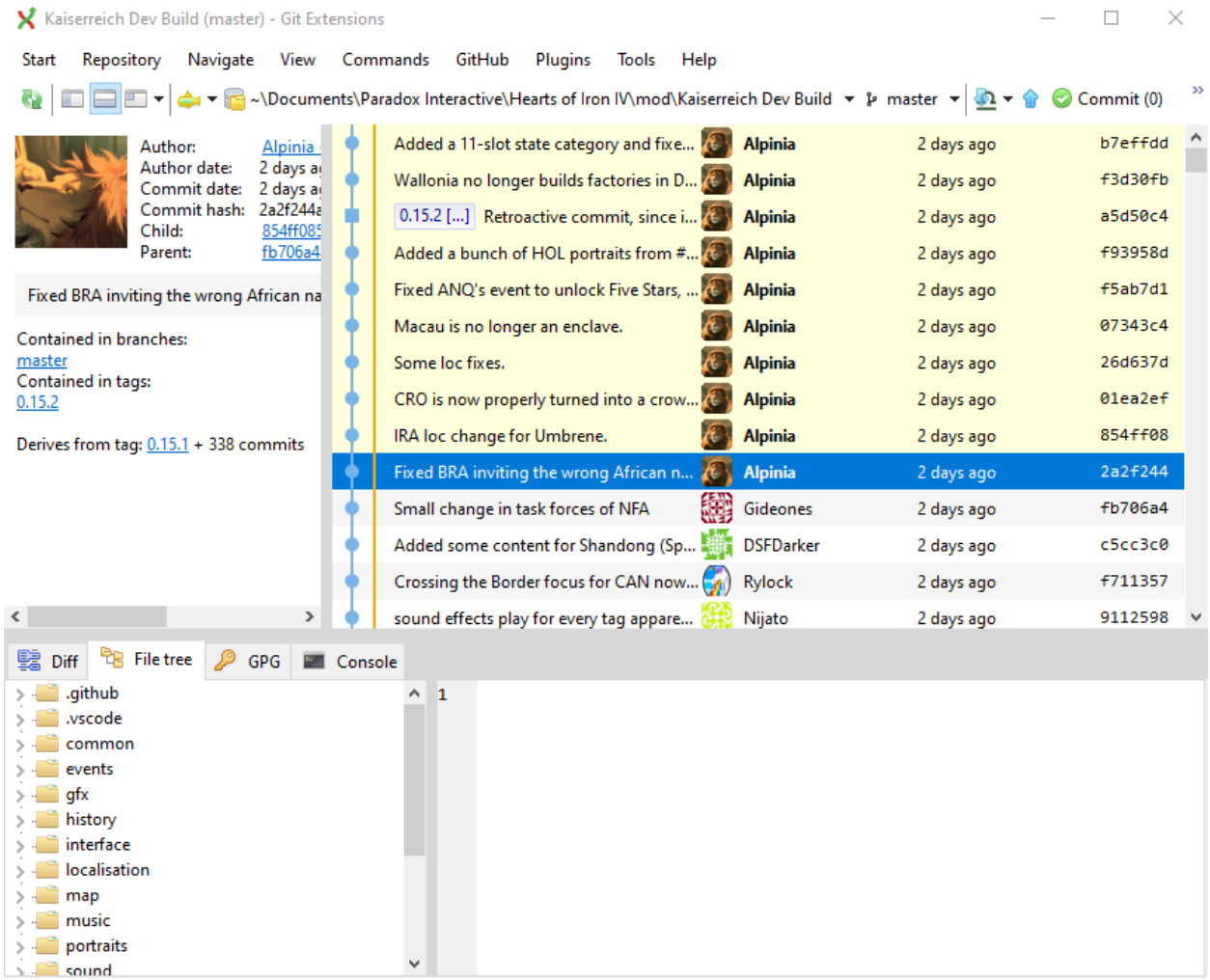
There are a few basic windows you are going to need to get familiar with. First is the right click menu, available by right clicking on the folder containing your Git.



Here you can see several useful commands. The four main ones you are going to be using are ‘GitExt Open repository’, ‘GitExt Commit…’, ‘Pull…’ and ‘Push…’. I’ll go through them one by one.

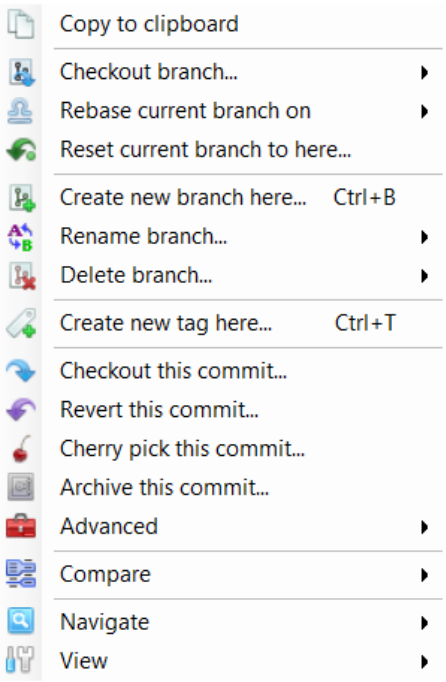
‘GitExt Open repository’

‘GitExt Open repository’ will take you to the menu of your project, something like this:



From here you can see a timeline of the project from commit to commit. Each one can be clicked on to view details about what was changed, who made the change and so on.

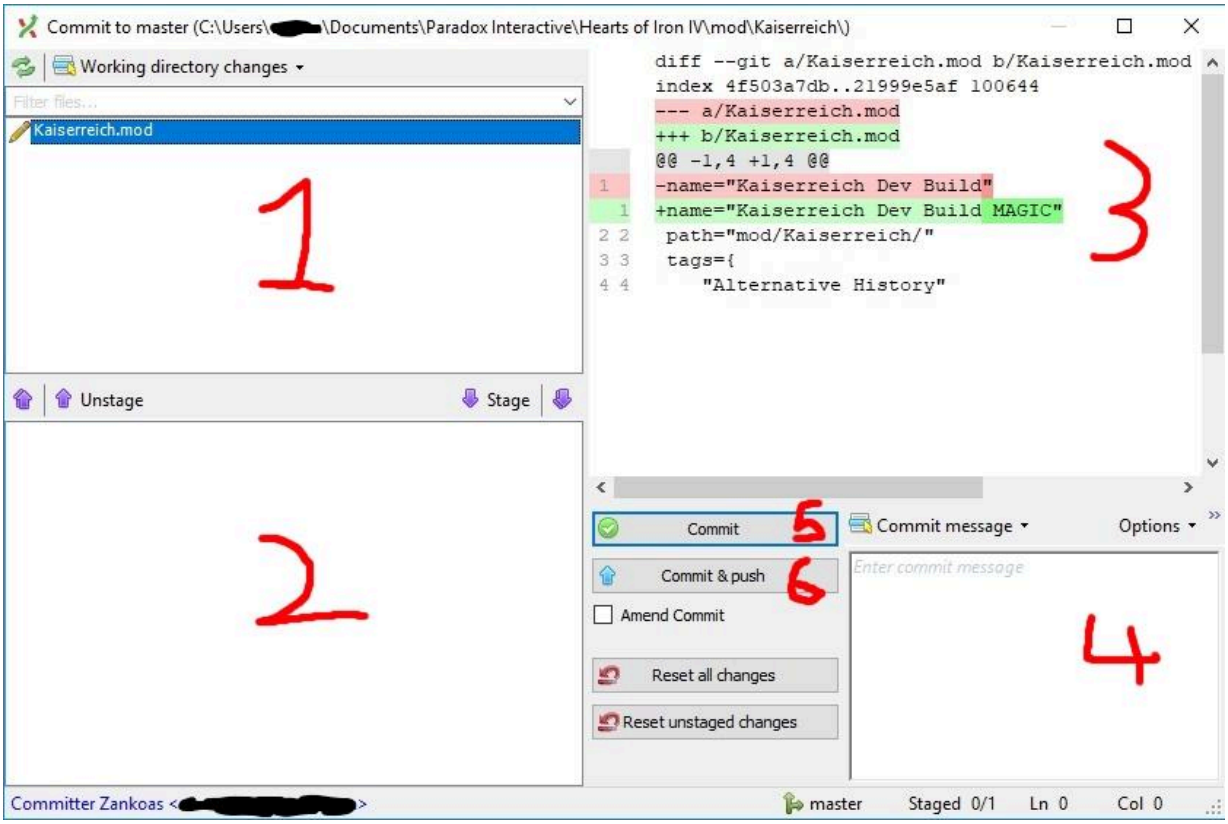
Right clicking on a commit brings up a new menu:



From here it is possible to switch between branches, merge branches and more, but details on that later in 8. Resetting and 10. Branches.

‘GitExt Commit…’

‘GitExt Commit…’ will take you to this window where it is possible to look over the changes you are about to make.



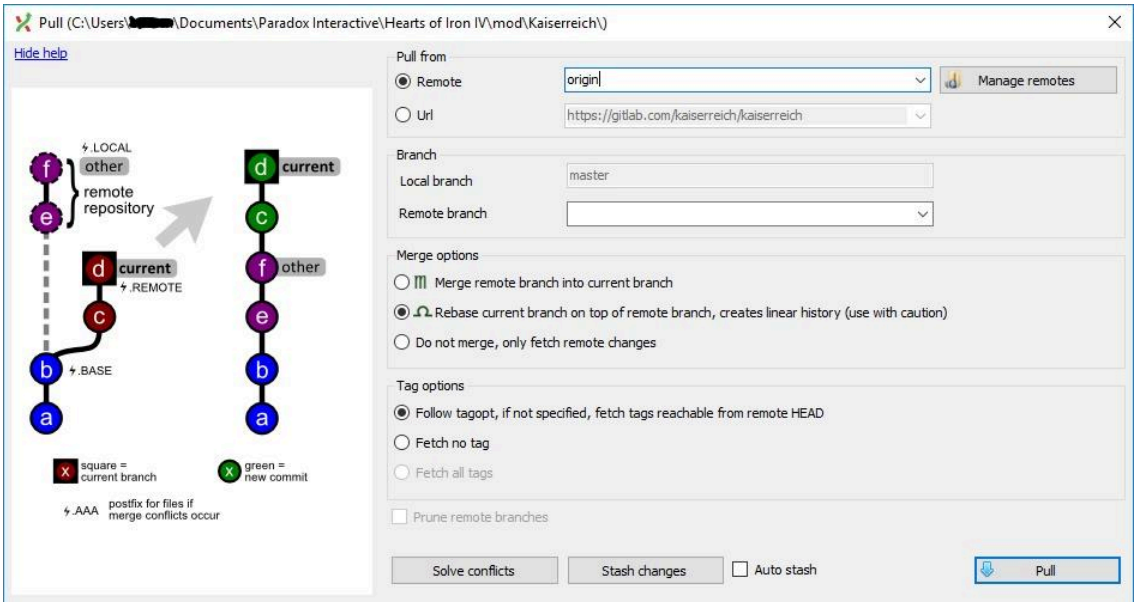
On the top left (1) you can see a list of all files that have been changed within the mod. Selecting one will show a line by line breakdown of the changes on the top right (3), here you can see I am adding ‘MAGIC’ to a line in the file called ‘Kaiserreich.mod’.

The bottom left window (2) is the ‘staged’ files, these are the changes you are about to commit, right now there are none, but double clicking on a file in the top right will stage it. Equally using the arrows in between the two windows (1 and 2) can be used to stage files.

When you have checked your changes and have staged them, it is time to commit them. Type a short message in the box to the bottom right (4) explaining what you have done, then press either commit (5) to make the changes, or commit and push (6) to make the changes and then push them, more details on pushing in a moment.

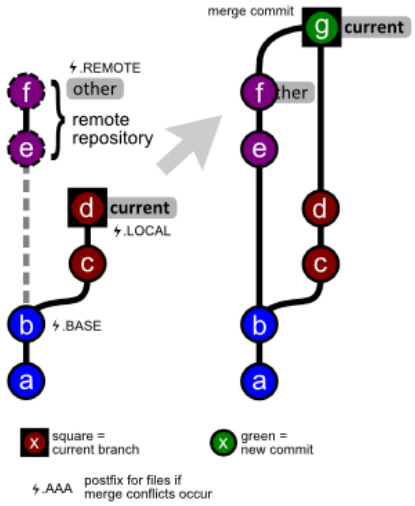
‘Pull…’

‘Pull…’ will take you to this window:



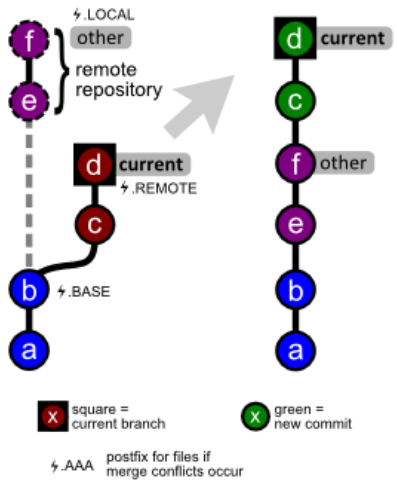
Here you are able to pull changes, while it looks all very complicated, the important setting to look at is ‘Merge options’. Here you can ‘Merge’ or ‘Rebase’.

Merging takes remote changes and local changes, and then attempts to fuse them together in a special ‘Merge Commit’, as shown by this handy diagram:



The left part is before the merge, with commits (a) and (b) at the bottom. The user has made two commits locally, (c) and (d), while two commits have been made by someone else remotely, (e) and (f). On the right we see the merge; a special merge commit, (g), is created, fusing those two lines together.

‘Rebase’ is the alternative method:

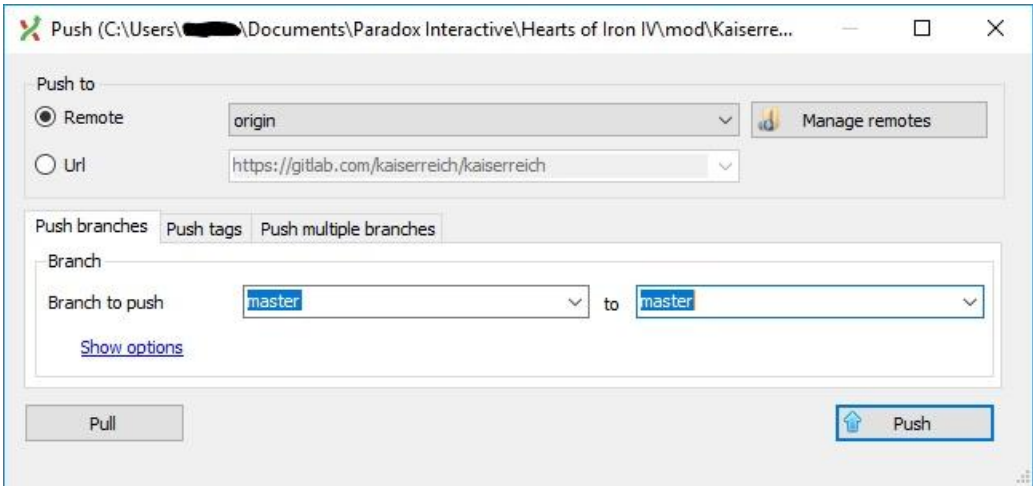


The left part is before the rebase, with commits (a) and (b) at the bottom. The user has made two commits locally, (c) and (d), while two commits have been made by someone else remotely, (e) and (f). On the right we see the rebase; the two local commits, (c) and (d), are temporarily undone, the remote commits applied, (e) and (f), then the two local commits, (c) and (d), are reapplied at the end, creating a nice clean line.

Some teams prefer merge, some prefer rebase, so if you are joining a team it is worth asking which one to use. If you aren't sure or don't care, rebase is *generally* neater so if in doubt I'd suggest using that. If you are setting up a team and want to know why I suggest rebase over merge, I've explained my reasoning in the Q&A in the appendix.

'Push…'

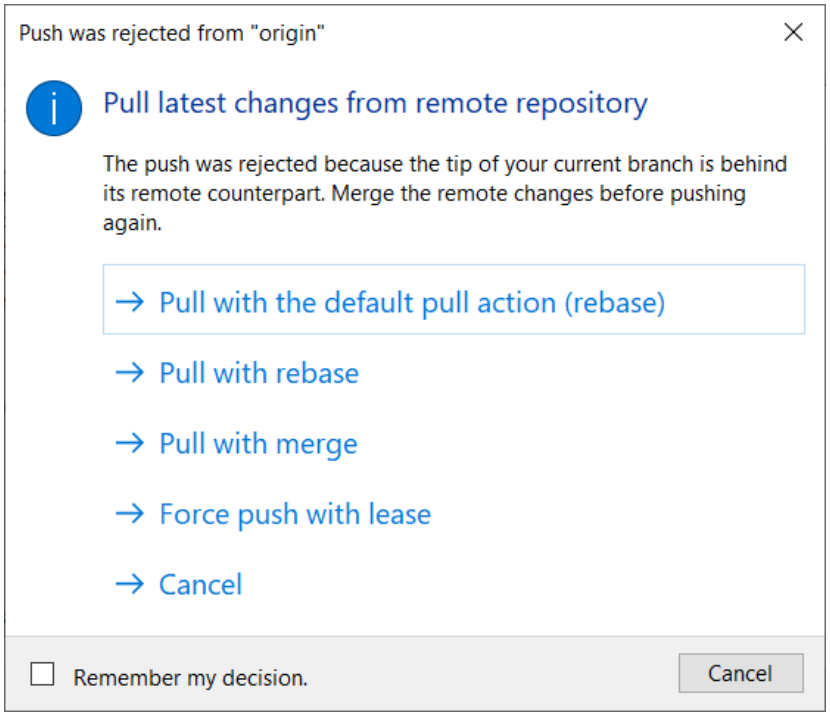
'Push…' is the final command, one which can be done by right clicking on your Git folder or from within the commit window under 'Commit and push'. Clicking it will bring up this window:



Here you can see which local branch you are going to push to which remote branch, more detail in 10. Branches, but for now all you need to do is click ‘Push’ and your local changes will be given to the remote Git.

Help My Push Was Rejected From ‘Origin’

Scary as I’m sure it is, it is actually very simple. Here is the popup:



This means your local Git is out of date, almost always due to someone else making changes since you last pulled. Thus, you can’t push your changes since that would overwrite the other person’s changes.

The first option is pull with the default action, typically (though it can be changed in the settings) merge. Pull with rebase and pull with merge are the same as pulling above, so follow the advice there. Force push is a special kind of push where you overwrite all changes on the server with your own. You must be very careful about doing this as it can, and will, delete other people’s work.

Unless you are absolutely sure what you are doing, never force push.

Cancel aborts your push, though leaves your local commit intact.

And that is it, all the basics of Git. You now know what a Git is, how to set one up remotely and locally, how to make changes locally (committing) and how to move changes between the remote and local Git (pushing and pulling). Congratulations!

But there is more to learn if you are up for it...

6. Creating a New Git

First, you will need to pick your host, that is the company that will store the online version of your Git.

The two main options are Github and GitLab. You can make your own mind up as to which to use, but I am going to assume you use GitLab and use that as the basis for the guide from this point onwards, though your experience won't be that different either way.

First you will need to create the online Git, to do this first create an account with GitLab at: https://gitlab.com/users/sign_in

If you are making a Git for a team it is a good idea to set up a group here: https://gitlab.com/dashboard/groups

This will let several people administer the Git to avoid the trouble of having only one admin.

Once you are done you will want to start a new project by clicking the green 'New project' button. From here you will want to name the project as well as give it a description. You also have the chance to select its visibility level.

For ease of setting up, make sure you tick 'Initialize repository with a README'.

Once you click 'Create project' your new Git will exist on GitLab's servers, and from there you can follow the normal setup guide in 5. Git Basics.

7. Manual Merges

We have already talked about the basic types of merging; Rebasing and Merging, but sometimes they just don't work. Why is this?

First, it is worth saying that Git has built in tools to automatically merge changes and so most of the time it will seamlessly handle this all for you. Sadly, sometimes that isn't possible. The most common reasons are:

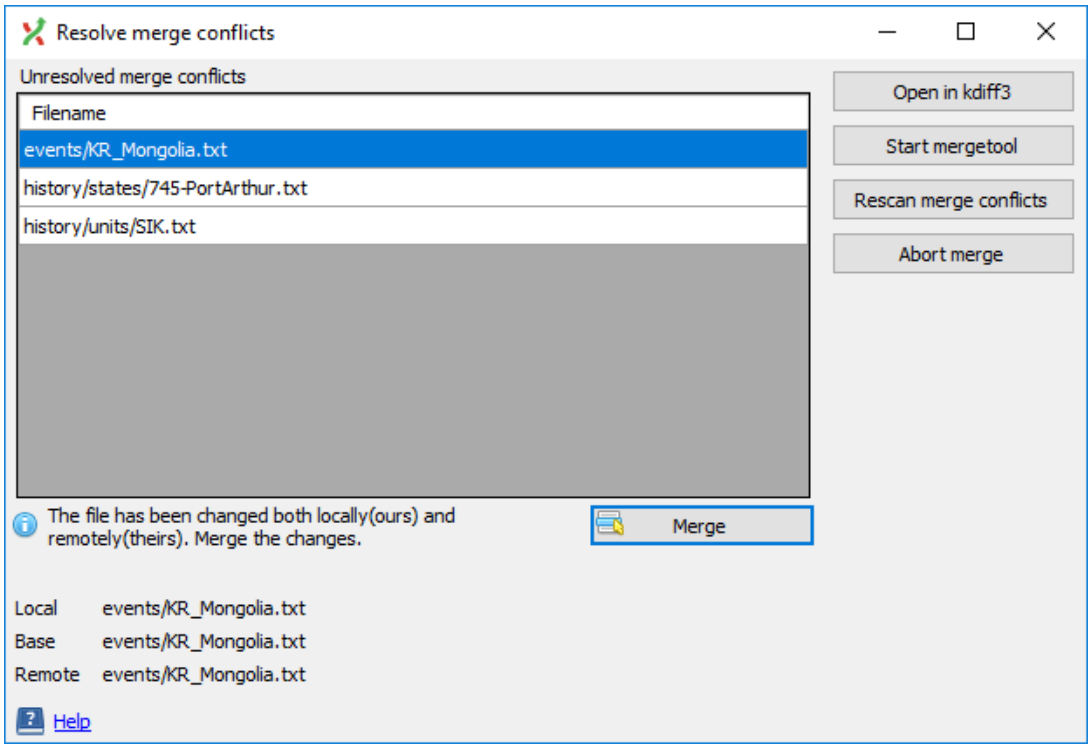
1. Someone has changed the same line of code as you have and Git isn't sure which change to use
2. Someone has changed a binary file (a file that can't be opened in a text editor, such as a graphics file like a .png) and Git can't merge the files together (you know, since there aren't lines of code to merge)

To avoid the mess of manual merges and the difficulty of doing them, as you are about to see, avoid 1. and 2. as best you can will minimise how often you have to do this, saving you much effort.

When Git can't merge automatically happens you have three options:

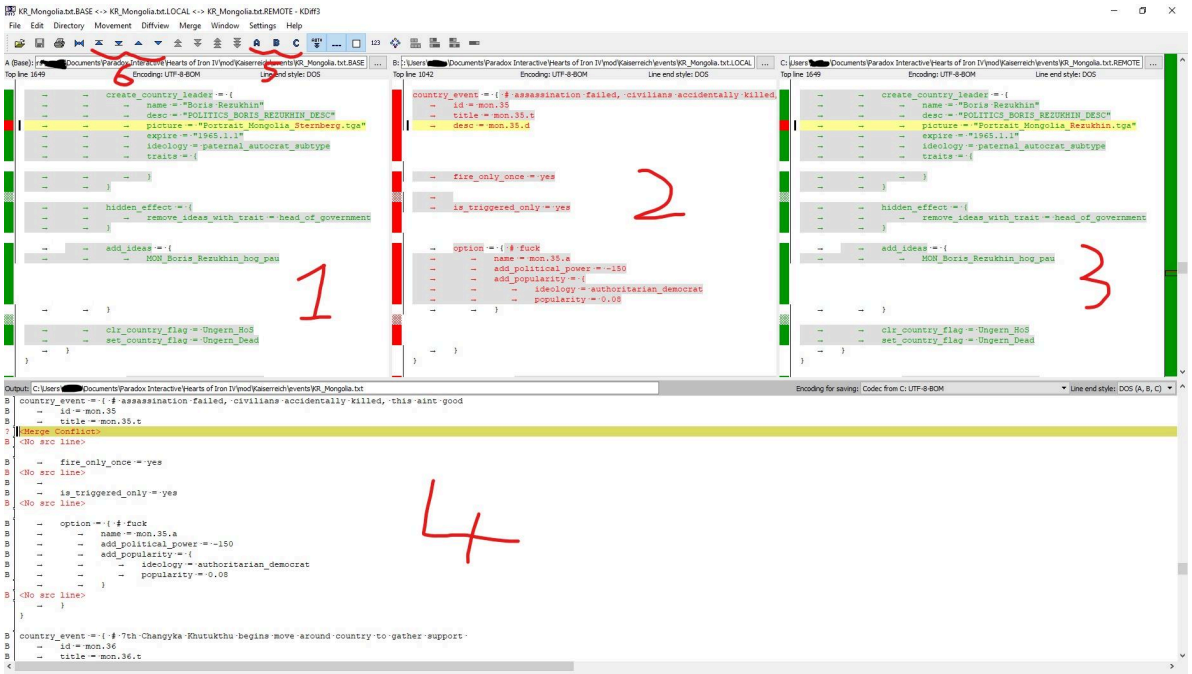
1. Delete your changes. This will let Git take their changes over yours and resolve the issue, the downside is you lose all your changes
2. Copy your file out of the mod folder, pull, then put them back in. This will overwrite all their changes with yours, the downside being they lose all their changes
3. Use a diff tool, this will let you compare line by line the issue and pick which version to keep, it gives you the most control and also shows you exactly which lines are an issue, what their change is, what your change is and what the unchanged version is. The only downside here is the tool isn't the most friendly thing to use, fortunately, I have a handy guide for you below.

If you have unresolved merge conflicts, are asked if you would like to solve them now and click yes, you will be taken to this window:



Here you can see the files that can't be automatically merged by Git. Selecting 'Merge' will bring up KDiff3, the specialist merge tool we installed earlier.

Straight away KDiff3 will try to merge and, if you are lucky, will be able to and say there are zero 'Nr of unsolved conflicts'. If so you can close KDiff3, save the result and move onto the next file. If not then things get more complicated.



Here you can see a breakdown of what KDiff3 is trying to do. (1) is the base file, (2) is the local file and (3) is the remote, the output is shown below at (4).

The ‘base’ file (1) is the file before any changes were done to, in other words, it is the file at the last shared moment of history between the two branches.
The ‘local’ file (2) is the file as it is on the branch you are currently on, at the time of the commit in question where there is the conflict.
The ‘remote’ file (3) is the file as it is on the branch you are trying to fuse into, at the time of the commit in question where there is the conflict.

The highlighted yellow line is one where KDiff3 can’t work out which of (1, 2 or 3) to use in (4). Clicking one of the three letters at (5) will select which line to use. The buttons at (6) let you move to the next yellow line. Once you are done click close, save, and move onto the next file.

Once you are done with all the files you can move forward as normal. Note that this is by far the most complicated part of using a Git and only rarely comes up, so don’t worry if you are struggling here.

One thing to watch out for is .orig files. These are record files that give detailed information on what happened during a manual merge. They can be saved as an easy to access record if need to see that information in one place, though all the information is contained, albeit in a less easy way to find in the File Tree (see chapter 12. History and the File Tree).

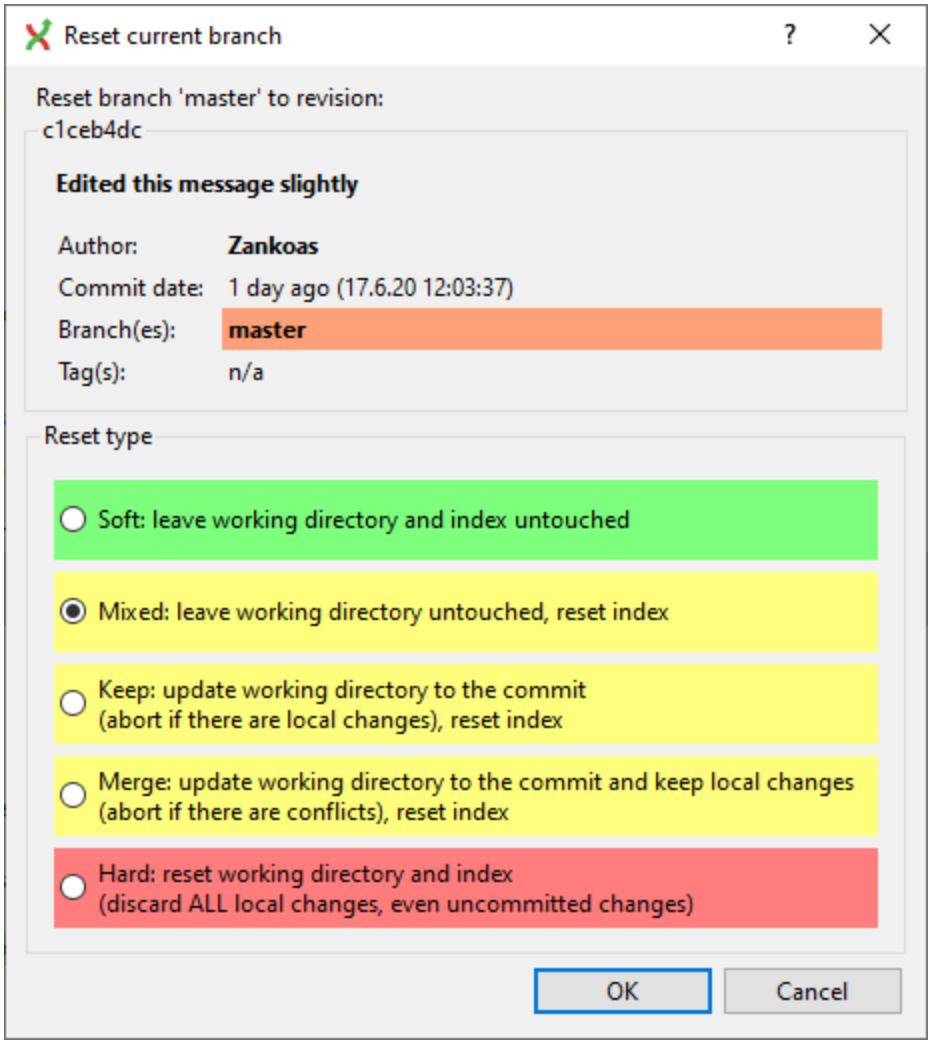
They are created by default when you manually merge a file, so make sure to either save them outside of the mod or delete them, as HOI4 will try to read them as code, which won’t go well. If you don’t use them, this automatic creation can be turned off via Git bash using the command ‘git config --global mergetool.keepBackup false’, and then also in KDiff3 settings under ‘Directory’, ‘Backup files (.orig)’.

8. Resetting

It happens to all of us, it has all gone wrong, you don’t know where or why, you aren’t sure what is happening and you just want to reset. Now of course, you could delete the Git folder and redownload it, but that takes time, effort and is just unnecessary (most of the time anyway).

Note that doing the following steps will delete all local changes that haven't been stashed or committed.

By going into the 'GitExt Open repository' window and right clicking on a commit you can see an option called 'Reset current branch to here...'. Selecting it will bring up this window:



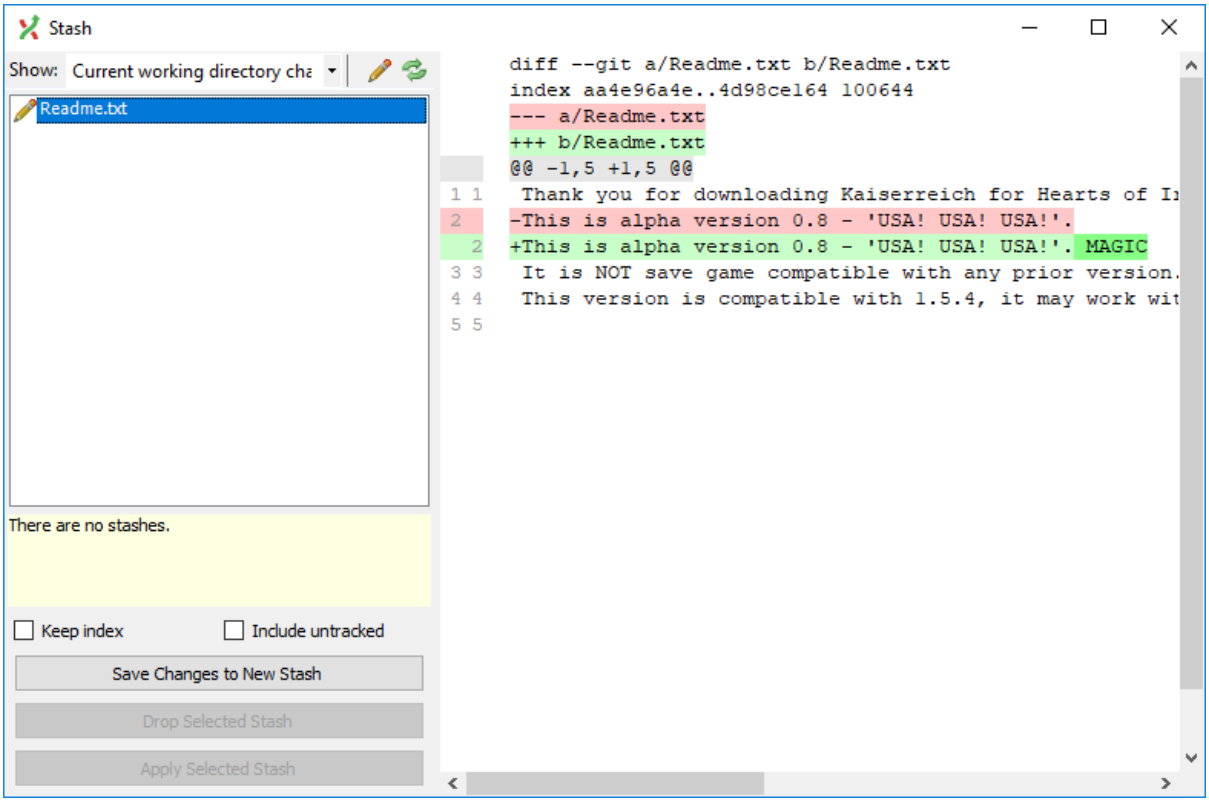
Select 'Hard' and click okay. You will now be moved to that commit exactly. If that commit was the latest commit (labeled as origin/master) you will now have your Git the same as if you have just deleted it and copied but in a fraction of the time.

In very rare cases, often involving a merge of a rebase, this won't fix things. At that point the easiest cause of action is just to copy out your changes, delete the whole folder and reclone.

9. Stashes

Say you have some work you want to save while you focus on some other work. You could copy the file out of your Git, save it somewhere else and then copy back in when done. While simple, it doesn't let you see what the changes are or merge back in nicely if someone else worked on the same file. What you need is a stash!

To manage your stash right click on your Git folder and select 'View stash'. This will bring up the following window:



Here you can see I have made a single change; adding 'MAGIC' to the Readme.txt file. If I wanted to save this I would click the pencil icon to give my stash a name, then simply click 'Save Changes to New Stash'.

The changes are now saved in a stash and I can continue working as if nothing happened.

To see a list of all my stashes I simply open the drop down menu at the top. To apply a stash, that is to say put the changes back, I select the stash from the drop down menu and click 'Apply Selected Stash' at the bottom. To delete a stash I again select it from the drop down menu but this time click 'Drop Selected Stash'.

10. Branches

Sometimes you want people to be working on different versions of a project at the same time, say two people are working on a hotfix while the rest of the team are working on the next big release. No need to set up two different Gits, what you need are branches!

Branches let you separate out different versions of a project and have them exist alongside one another, able to be switched between easily.

There are two branches that will (unless you have picked a different name for them) exist already; ‘master' and ‘origin/master’. ‘master’ is where your local Git is currently at while ‘origin/master’ is where your local Git *thinks* the remote Git is currently at. If you haven’t pulled in a while, ‘origin/master’ may well be out of date, but it is a useful indicator of what the latest change you have locally, as well as being useful for telling you where to switch to if you want to leave another branch.

The end game for a branch is normally merging it back into master, so to ensure that is nice and easy, follow the rules laid out in 7. Manual Merges.

Creating a Branch

Creating a branch is very simple. You can either right-click on any commit and select “**Create new branch here…**”, or use the shortcut **Ctrl+B**. A prompt will appear asking you to enter a name for your new branch.

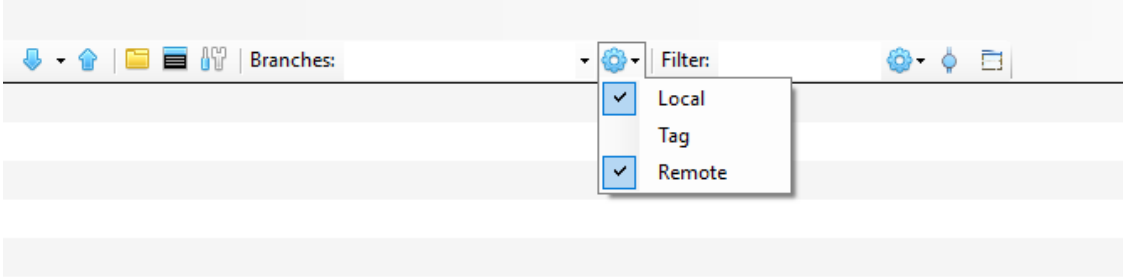
When you push a new branch to the remote repo, GitExt will ask you to confirm what you are doing. Make sure you are pushing the correct branch!

Moving Between Branches

There are several ways of moving between branches, but my personal favorite is via resetting. This isn’t the cleanest way but it is visually pretty simple. Note that doing this will delete all local changes that haven’t been stashed or committed.

Bring up the ‘GitExt Open repository’ window as normal and find the commit of the branch you want to switch to, normally labeled as ‘origin/NAME_OF_BRANCH’. Then follow section 8. Resetting to switch that branch.

How do I find the branch you ask? Well if it was recently changed you can just look for it in the ‘GitExt Open repository’ window, but if you can’t find it you can select it from the branch list here:



First click on the little cog and make sure ‘Remote’ is selected. Then select the drop down menu to the left and it will give you a list of all the branches in the Git. Click the one you want, hit enter, and you will be taken to it.

Once you are done with the branch use the same method to reset back to master.

Making Changes on a Branch

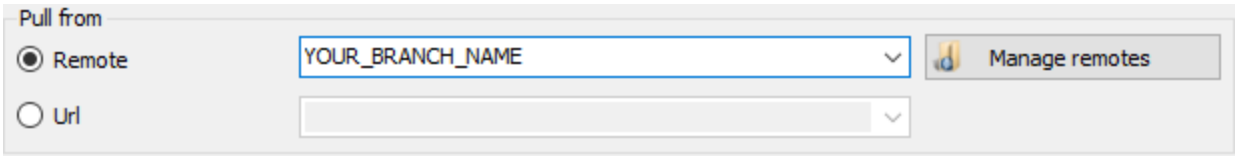
To make a change on a branch simply:

1. Move to it as described above
2. Commit as normal
3. Then as with creating a branch, change the branch you are pushing to to that branch

Simple as pie.

Pulling on a Branch

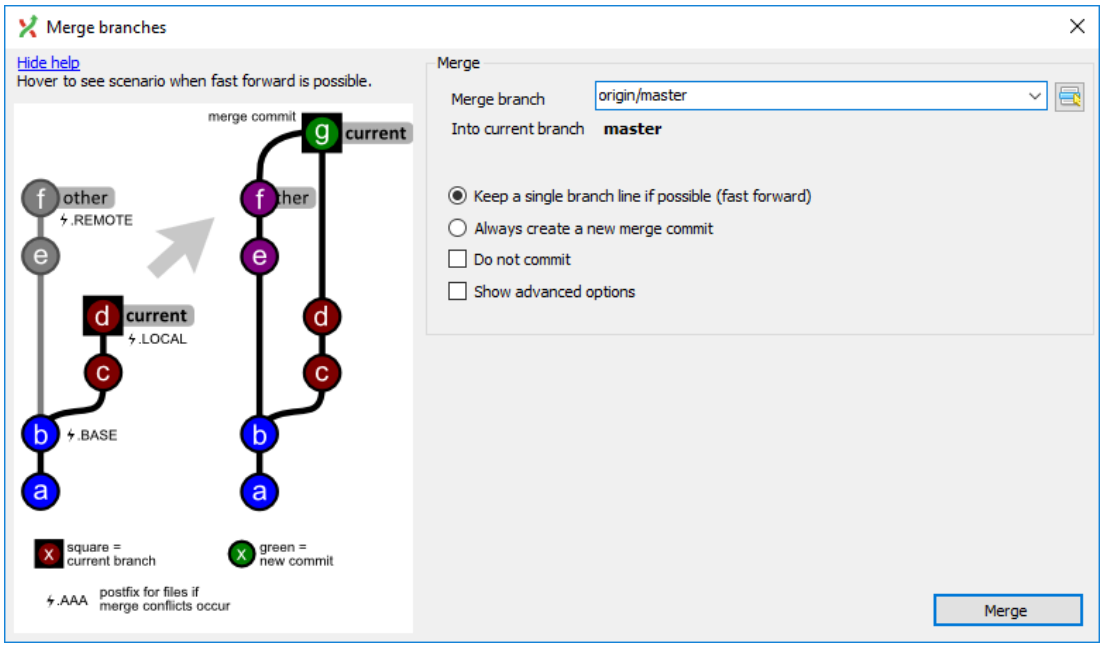
If several people are working on a branch at the same time and you want to pull changes, you can either switch out of it to master, pull as normal and switch back, or just directly pull their changes. To do this select Pull as normal, but change the remote from master to your branch name like so:



Merging Branches

Finally, how to merge a branch in. First you want to move to the branch in question. Then right click on the commit labeled ‘origin/master’ and select either ‘Merge into

current branch' or 'Rebase current branch on'. While rebasing is neater, for larger branches it is very difficult and time consuming, so merging is normally preferred. Once you click 'Merge into current branch' you will be greeted by this window:



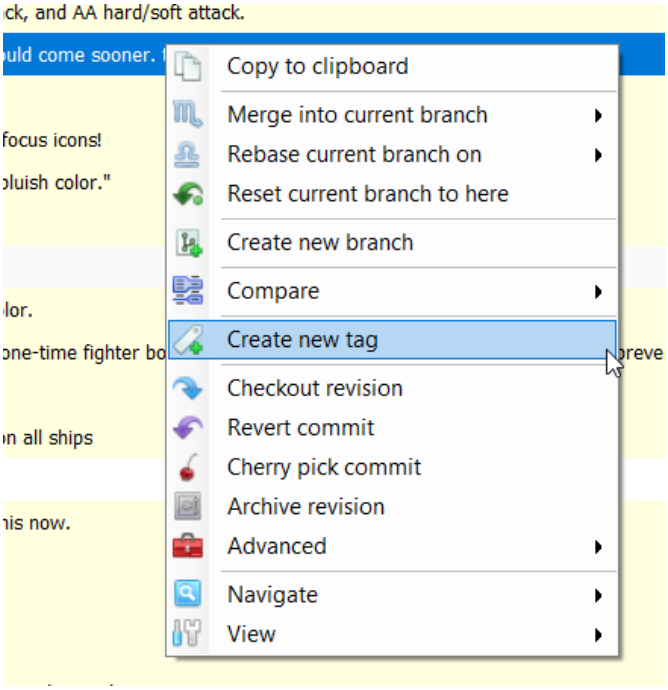
Here you can confirm that you are merging the right branches together. Once you click merge Git will try to auto merge. If it works, great, if not, as with 7. Manual Merges, you will be taken to the Manual Merging window.

Once it is done you will have a local Git that is the fusion of both branches. If you want to update the branch with changes from master you simply commit and push to the branch, while if you want to update master with the changes from the branch you commit and push to master.

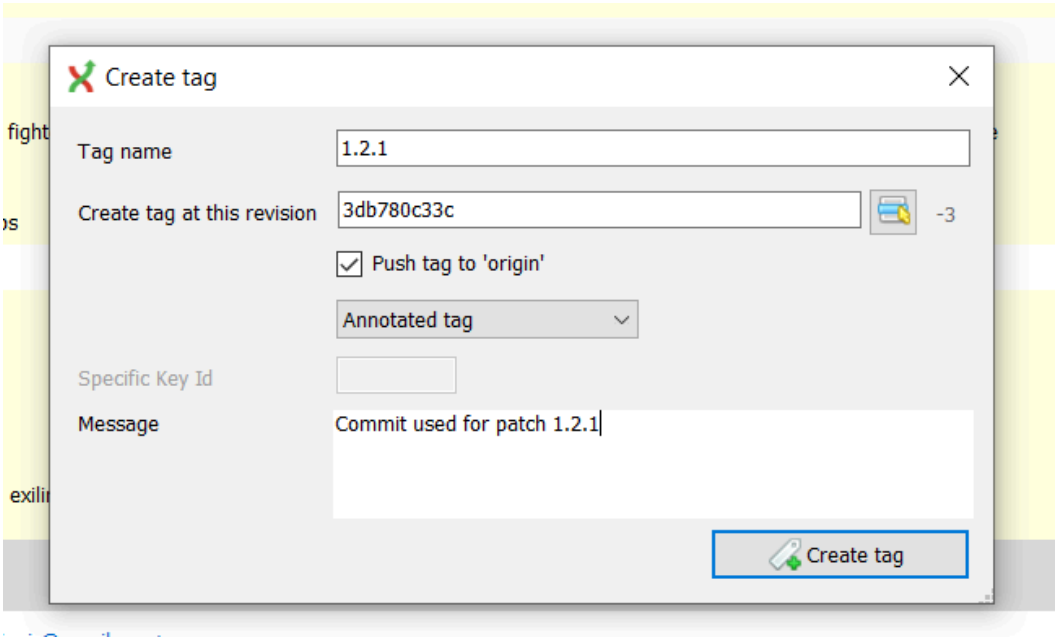
11. Tags

Tags are a wonderful tool enabling you to easily mark important versions of the mod, without needing to have full branches for them. This is often used to mark release versions, though can be used to mark any version you feel is worthy. Just like with branches, you can then easily jump to each tag, saving you scrolling through hundreds of commits to get there.

To create a tag, first find the commit you want to mark and right click on it, then select 'Create new tag':



That will bring up the following window:

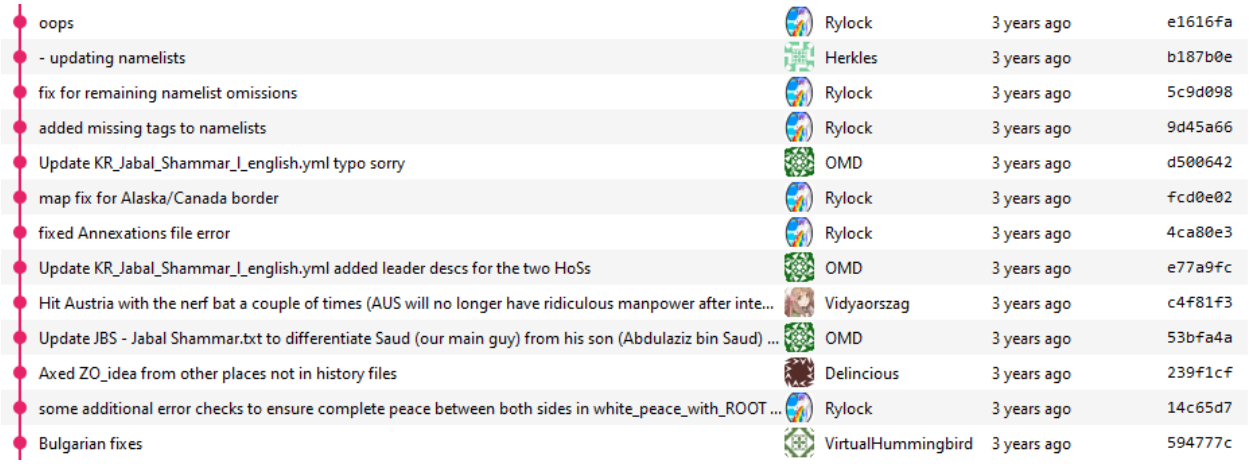


Give your tag a name, such as the release version number you are marking (here ‘1.2.1’), tick 'Push tag to 'origin'' and then select ‘Annotated tag’ from the drop down menu. Fill in the message with what the tag is, then click ‘Create tag’ and you are done!

12. History and the File Tree

One of the headline benefits of a Git is that, by storing changes rather than whole new versions, you are able to see every version of the mod that has ever existed. In other words, the entire history of the project is available to you.

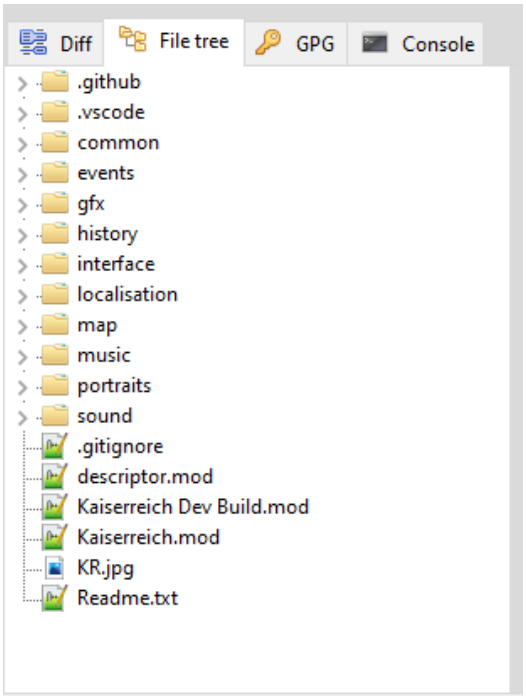
The main way of viewing this history is in the timeline. You are likely familiar with it as it is the main page you land on when opening GitExt. It contains a list of all the commits made in chronological order. You can see what it looks like here:



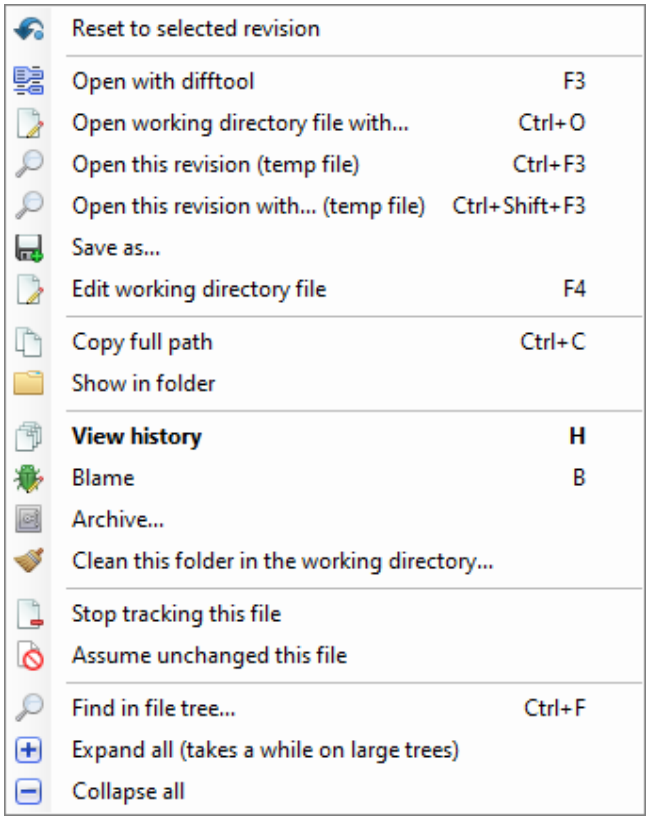
Clicking on any commit will bring up details about it on the left hand side, showing you who made it, what message they gave, when they changed it and so on.

Along the bottom you will be able to see the ‘Diff’ view, showing you the **Diff**erence between this commit and the prior one, or in other words, what they changed.

Next to that view is the file tree view. You can access it by clicking ‘File tree’, letting you browse the mod as it existed at the time of that commit. To browse the mod as it currently is simply select the latest commit. You can see the file tree view below:

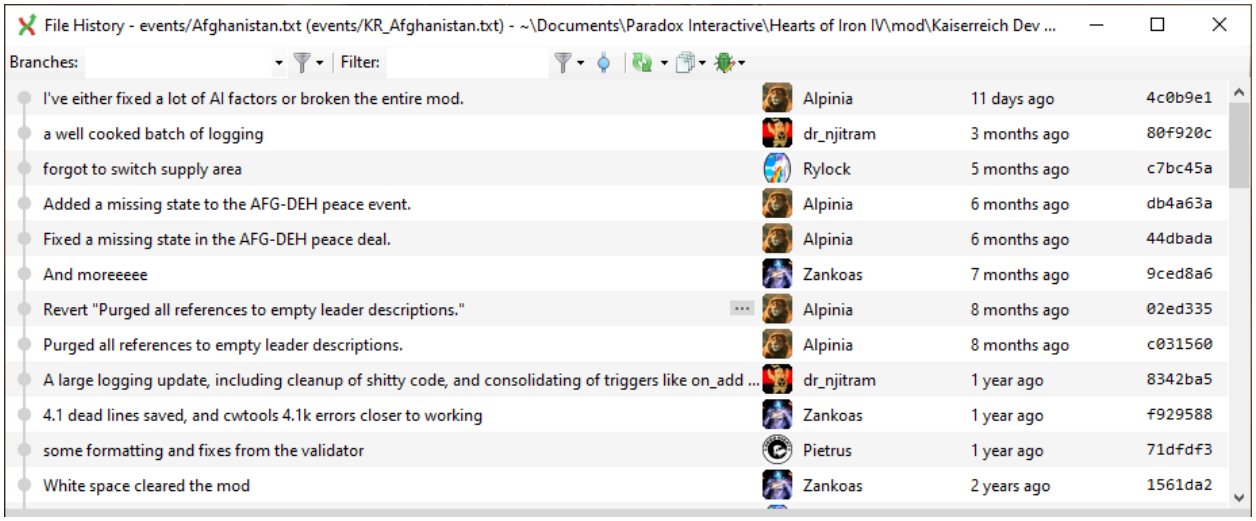


Within this view right clicking on a file will bring up a menu that lets you access several useful functions:



The ‘Save as…’ button lets you save a copy of the file as it existed in that commit, useful for pulling files out of old versions of the mod without having to fully reset back to it.

The ‘View history’ button lets you view the history of that file only. This will bring up a view which looks similar to the main mod history, but crucially it only shows you commits which changed this file. As a result, you can easily see who has changed a given file, when, what they did and so on. You can also access this by going into the mod folder in file explorer, then right clicking it, mousing over ‘Git Extensions’ and then clicking ‘File history’. Either method will lead you to the same result which you can see below.



Finally there is the ‘Blame’ button. The one will, as with ‘View history’, bring up a history view of all the commits that have changed this file, but this time highlighting who last changed each line of the file, giving you even more granular control. You can see an example below:

13. Appendix

And that's it! You should now know enough to get going on your first Git. This guide was designed to be as easy to read as possible so suggestions and feedback are very much welcome. Thanks for reading and I hope it was of some use to you!

- Zankoas

Troubleshooting

While tech support is beyond this guide, and there are a million and one potential issues you might face, there are a few common ones I think it worth putting basic troubleshooting steps here for.

I entered the wrong username or password!

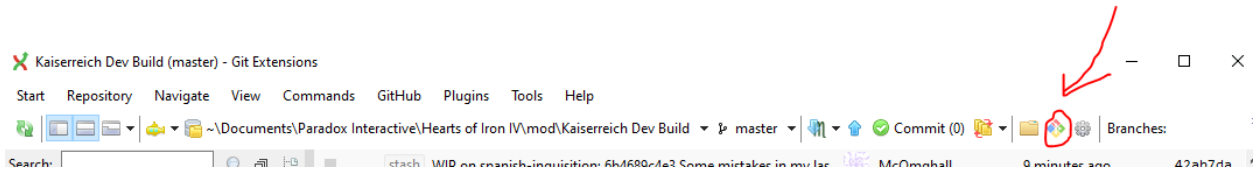
Sadly GitExt can't tell if your username or password is wrong, so it will just give you the generic 'access rejected' message. To prompt GitExt to ask again, head to the windows 'Credential Manager', then 'Windows Credentials' and look for gitlab.com. Select it from the list and delete it; next time GitExt tries to access something it will once again prompt you for your username and password.

GitExt is crashing on me even after reinstalling!

If you are on windows, heading into the temporary GitExt folder in %APPDATA% and deleting it has been known to fix crashes. This will wipe your local settings, but if you are crashing, it's worth a shot.

GitExt asks me for my login details every time I start windows!

1. Make sure you have updated to the latest git version, see 4. Setup Guide on install/updating instructions
2. Open a git Command Line Interface from GitExt clicking this button in the Open Repository Window:



3. In the prompt that will appear type "git config --global credential.helper wincred" and press enter
4. Go to the Windows Credential Manager

    a. In Windows 10, click the start button where all your apps can be found and type “Credential”
    b. In that window, go to Windows Credentials and delete all the ones that include “GitHub”
5. Try pulling again from GitHub
    a. It will try to log you in to GitHub in your browser, input credentials if necessary
    b. Give access to the Git Credential Manager app to your account (should happen immediately after you log in to your GitHub account)
6. Double check you’ve pulled correctly by confirming there were no errors during the pull, then pull again to confirm your credentials are saved (i.e. you don’t get a popup asking for them)

Q&A

Why only talk about GitLab and GitHub? Why only talk about GitExt? Why don’t you mention **insert other thing here**?

In order to keep this guide short, simple and accurate, I have only talked about what I am confident in and with minimal other options. Ultimately if you are at the point of questioning which host or client to use, you are beyond this simple guide.

What if I want to use another client?

Then go for it! While this will make this guide slightly less relevant to you, the fundamental concepts of committing, pushing, pulling and so on are universal between all Git clients.

Why do you suggest rebase over merge when merge is the default?

In my experience modding, teams tend to make lots of small commits rather than singular larger commits. In that environment having many merges can quickly make reading the Git history difficult, so using rebase makes more sense. One of the reasons merge is the default is that rebase technically changes the history of the project. For mods this isn’t something we really care about, but for a large project where accountability and security are primary concerns, this is often a deal breaker.

Why is there no section on Mac or Linux?

Aside from a lack of personal experience using a Git on them, the audience for a GUI client for someone modding on them is tiny so I find it difficult to justify spending time expanding the guide for them. That said, I wouldn’t be opposed to adding a short follow

up section for them; if someone with more experience is interested in writing it please get in contact with me.

Does Git actually store changes as the difference between the current and the previous version?

Technically no, it doesn't. If you really think about it, it can't be that way as that would mean that in order to relocate to a different commit it would need to iterate through millions of diffs from the very start to that new commit, which would take ages, yet Git can do it almost instantly. Git actually stores commits as snapshots, but simply shows you the diff to make it easy for you to understand. I find people tend to 'get it' quicker when thinking about Git in terms of diffs rather than snapshots and so have gone with that approach, even if it is technically false. If you are interested and want to learn more about exactly how it works, Derrick Stolee has a great blog post about it here: Commits are snapshots, not diffs

Changelog

1.15 - Updated Discord usernames to match new format
1.14 - Improved linking between the document, and adjusted wording to reflect that the initial branch may no longer be called 'master'
1.13 - Maintainer changed from Zankoas to Alpinia
1.12 - Added a Q&A explaining that Git technically works via snapshots, not diffs
1.11 - Split the Q&A into the old Q&A and a new troubleshooting section, adding new steps you can take to deal with credentials being forgot
1.10 - Added a new section on history and the file view, updated images throughout, made the install instructions easier to follow
1.9 - Updated the setup guide to reflect new install options, added new intra-document links, changed the KDiff3 install link to the new repo
1.8 - Prettified some of the links along with minor grammar tweaks
1.7 - Updated the resetting guide for the new options, as well as some grammar tweaks to help readability
1.6 - Changed link formatting to be easier to read as well as redoing some images to be clearer
1.5 - Added two new questions to the Q&A, plus minor formatting tweaks
1.4 - Updated images, title names and text to reflect the new GitExt UI, made some additional clarifications to the merging section, added a Q&A regarding other clients, as well as some minor formatting and typo fixes
1.3 - Added a message about the new GitExt UI and added a guide to using tags, along with some minor formatting and grammar tweaks

1.2 - Changed chapter order and expanded the manual merging section
1.1 - Added a guide to stashing, created an appendix and made clarity tweaks all over thanks to feedback
1.0 - Initial version

Credits

Maintained by Zankoas until 1.13, then Alpinia
Written by Zankoas
Stashing guide written with the help of Paul
Credentials being forgot troubleshooting steps written with the help of McOmghall
Improved with initial feedback from Yard1, Gunnar Von Pontius and Jeankedezeehond among many others