# Subject: TortoiseGit client for windows.

Date: 4/28/2020

Author: Asoka Vangala

1. **Purpose**:

This document defines the procedure for tracking project files for the latest changes and creates history for tracking previous changes.

1. **Scope:**

Verify functionality of the TortoiseGit with user defined examples.

1. **Reference documents and Related Web sites for information:**

GitHub for 32-bit windows.

<https://github.com/git-for-windows/git/releases/download/v2.26.2.windows.1/Git-2.26.2-32-bit.exe>

GitHub for 64-bit windows.

<https://github.com/git-for-windows/git/releases/download/v2.26.2.windows.1/Git-2.26.2-64-bit.exe>

GitHub Desktop for windows.

<https://central.github.com/deployments/desktop/desktop/latest/win32>

TortoiseGit for 32-bit windows.

<https://download.tortoisegit.org/tgit/2.10.0.0/TortoiseGit-2.10.0.2-32bit.msi>

TortoiseGit for 32-bit windows.

<https://download.tortoisegit.org/tgit/2.10.0.0/TortoiseGit-2.10.0.2-64bit.msi>

C:\Users\avangala>git --version

git version 2.27.0.windows.1

1. **Audience Requirements:**

This document is written for computer literate folks who want to use Git to manage their data but are uncomfortable using the command line client to do so. Since TortoiseGit is a windows shell extension it is assumed that the user is familiar with the windows explorer and knows how to use it.

1. **System requirements:**

PC with Windows 7 OS or Windows 10 OS with minimum 8GB ram size and ample space for download SW on hard drive.

1. **Terminology used in this document:**

To make reading the docs easier, the names of all the screens and Menus from TortoiseGit are marked up in a different font. The Log Dialog for instance. A menu choice is indicated with an arrow. TortoiseGit → Show Log means: select Show Log from the TortoiseGit context menu. Where a local context menu appears within one of the TortoiseGit dialogs, it is shown like this: Context Menu → Save As ...

User Interface Buttons are indicated like this: Press OK to continue.

User Actions are indicated using a bold font. **Alt+A**: press the **Alt**-Key on your keyboard and while holding it down press the **A**-Key as well. Right-drag: press the right mouse button and while holding it down drag the items to the new location.

System output and keyboard input is indicated with a different font as well.

1. **ICON definitions with symbols:**

 Important notes are marked with an icon(important).

 Tips that make your life easier(tip).

 Places where you must be careful what you are doing(caution).

 Where extreme care must be taken, data corruption or other nasty things may occur if these warnings are ignored(warning).

1. **Version Control:**

Version control is the art of managing changes to information. It has long been a critical tool for programmers, who typically spend their time making small changes to software and then undoing or checking some of those changes the next day. Imagine a team of such developers working concurrently - and perhaps even simultaneously on the very same files! - and you can see why a good system is needed to manage the potential chaos*.*

1. **What is TortoiseGit?**

TortoiseGit is a free open-source client for the Git version control system. That is, TortoiseGit manages files over time. Files are stored in a local repository. The repository is much like an ordinary file server, except that it remembers every change ever made to your files and directories. This allows you to recover older versions of your files and examine the history of how and when your data changed, and who changed it.

**9.1 Definitions for TortoiseGit commands:**

**Add:** A Git command that is used to add a file to your working tree. The new items are added to the repository when you commit.

**BASE revision:** This is the common ancestor's version of a conflicted file.

**Blame:** This command is for text files only, and it annotates every line to show the repository revision in which it was last changed, and the author who made that change. Our GUI implementation is called TortoiseGitBlame and it also shows the commit date/time and the log message when you hover the mouse of the revision number.

**Branch:** A term frequently used in revision control systems to describe what happens when development forks at a point and follows 2 separate paths. You can create a branch off the main development line to develop a new feature without rendering the main line unstable. Or you can branch a stable release to which you make only bug fixes, while new developments take place on the unstable trunk. In Git a branch is implemented as a “pointer to a revision”.

**Cleanup:** Remove untracked files from the working tree.

**Clone:** A Git command which creates a local working tree in an empty directory by downloading a remote repository.

**Commit:** This Git command is used to pass the changes in your local working tree back into the repository, creating a new repository revision.

**Conflict:** When changes from the repository are merged with local changes, sometimes those changes occur on the same lines. In this case Git cannot automatically decide which version to use and the file is said to be in conflict. You must edit the file manually and resolve the conflict before you can commit any further changes.

**Copy:** In a Git repository you can manually create a copy of a single file or an entire tree w/o problem.

**Delete:** When you delete a versioned item (and commit the change) the item no longer exists in the repository after the committed revision. But of course, it still exists in earlier repository revisions, so you can still access it. If necessary, you can copy a deleted item and “resurrect” it complete with history.

**Diff:** Shorthand for “Show Differences”. Very useful when you want to see exactly what changes have been made.

**Export:** This command produces an compressed archive of all versioned files (of a specific revision).

**GPO:** Group policy object.

**HEAD:** HEAD is a synonym for the currently active branch (to be more precise in Git HEAD can also be so-called "detached" and directly pointing to a commit instead of a branch).

**History:** Show the revision history of a file or folder. Also known as “Log”.

**Log: Show** the revision history of a file or folder. Also known as “History”.

**Merge:** The process by which changes from the repository are added to your working tree without disrupting any changes you have already made locally. Sometimes these changes cannot be reconciled automatically, and the working tree is said to be in conflict. Merging happens automatically when you pull changes, cherry-pick, or rebase. You can also merge specific changes from another branch using TortoiseGit's Merge command.

**Patch:** If a working tree has changes to text files only, it is possible to use Git's Diff command to generate a single file summary of those changes in Unified Diff format. A file of this type is often referred to as a “Patch”, and it can be emailed to someone else (or to a mailing list) and applied to another working tree. Someone without commit access can make changes and submit a patch file for an authorized committer to apply. Or if you are unsure about a change you can submit a patch for others to review.

**Pull:** This Git command pulls down the latest changes from the repository into your working tree, merging any changes made by others with local changes in the working tree.

**Repository:** A repository is a place where data is stored and maintained. A repository can be a place where multiple databases or files are located for distribution over a network, or a repository can be a location that is directly accessible to the user without having to travel across a network. Git is a distributed version control system - each working tree contains its own repository (in the. git folder). A Git repository does not require network to work with most operations. Network is required only when you need to synchronize changes with remote repositories.

**Resolve:** When files in a working tree are left in a conflicted state following a merge, those conflicts must be sorted out by a human using an editor (or perhaps TortoiseGitMerge). This process is referred to as “Resolving Conflicts”. When this is complete you can mark the conflicted files as being resolved, which allows them to be committed.

**Revert:** If you have made changes and decide you want to undo them, you can use the “revert” command to go back to the version from HEAD.

**Revision:** Every time you commit a set of changes, you create one new “revision” in the repository. Each revision represents the state of the repository tree at a certain point in its history. If you want to go back in time you can examine the repository as it was at a specific revision. In another sense, a revision can refer to the set of changes that were made when that revision was created.

**SVN:** A frequently used abbreviation for Subversion. TortoiseGit provides git-svn interoperability. You can fetch partial or whole history from an SVN remote and store as a local git repository. This allows you to browse the history and create commits locally. You can finally commit your changes to an SVN remote.

**Switch/Checkout:** Updates all files in the working tree to a specific version. This is normally used for switching/checking out branches.

**Update:** The corresponding command for the SVN update command is *Pull*.

**Working Tree:** This is your local “sandbox”, the area where you work on the versioned files, and it normally resides on your local hard disk. You create a working tree by doing a “Clone” of a repository, and you feed your changes back into the repository using “Commit”.

1. **TortoiseGit's Features:** 
   1. **Shell integration:**

integrates seamlessly into the Windows shell (i.e. the explorer). This means you can keep working with the tools you're already familiar with. And you do not have to change into a different application each time you need functions of the version control!

**10.2 Icon overlays:**

The status of every versioned file and folder is indicated by small overlay icons. That way you can see right away what the status of your working tree is.

**10.3 Easy access to Git commands:**

All Git commands are available from the explorer context menu. TortoiseGit adds its own submenu there.

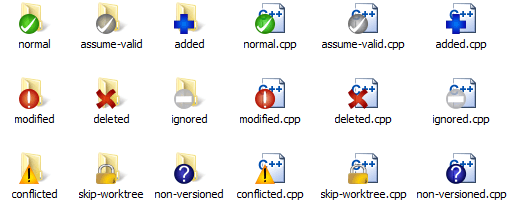
1. **Installing TortoiseGit:**

You need Administrator privileges to install TortoiseGit.

1. **Daily Use Guide:**

* You should have installed TortoiseGit already.
* You should be familiar with version control systems.
* You should know the basics of Git.
* You should have set up a server and/or have access to a Git repository.

1. **Getting Started:**
   1. **Icon Overlays:**

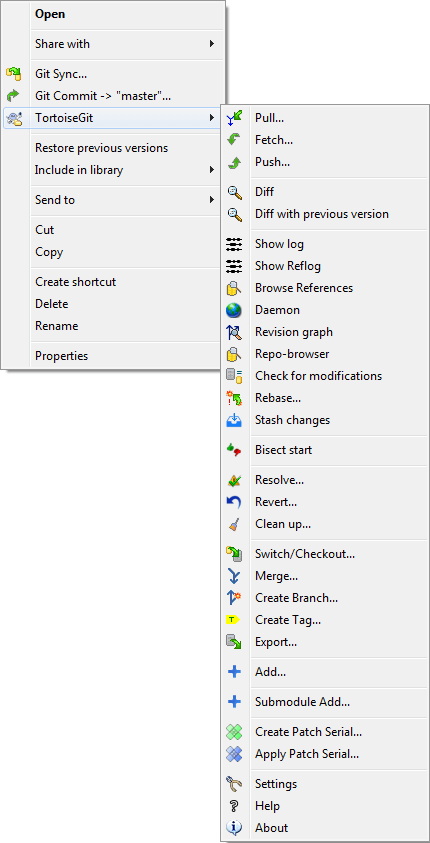


One of the most visible features of TortoiseGit is the icon overlays which appear on files in your working tree. These show you at a glance which of your files have been modified. This is only an example file type C++.File type may be different for different users.

* 1. **Context Menus:**

All TortoiseGit commands are invoked from the context menu of the windows explorer. Most are directly visible when you right click on a file or folder. The commands that are available depend on whether the file or folder or its parent folder is under version control or not. You can also see the TortoiseGit menu as part of the Explorer file menu.

Some commands which are very rarely used are only available in the extended context menu. To bring up the extended context menu, hold down the **Shift** key when you right-click.



* 1. **Drag and Drop:**

Other commands are available as drag handlers when you right drag files or folders to a new location inside working trees or when you right drag a non-versioned file or folder into a directory which is under version control.

* 1. **Common Shortcuts:**

Some common operations have well-known Windows shortcuts, but do not appear on buttons or in menus. If you cannot work out how to do something obvious, like refreshing a view, check here.

F1 for Help, of course and F5 for Refresh the current view.

Ctrl-A for Select all.

Ctrl-C for Copy the selected text.

Ctrl-F for Search

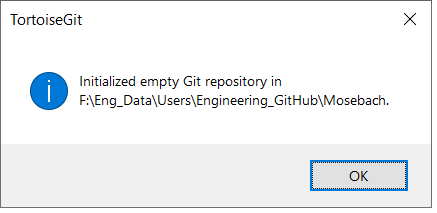
1. **Create Repository:**

This section talks about how to create a git repository. Creating an empty git repository is very simple. At an empty directory, just use the explorer context menu and select Git Create Repository here.

**A screenshot of a cell phone

Description automatically generated**

You can choose here between a bare and normal git repository. A normal repository has a working tree attached to which files can be checkout out and committed whereas a bare repository only can be pushed to and pulled from. After a (non-bare) repository is created a message box will be shown:



1. **Clone Repository:**

This section talks about how to clone a git repository from an existing repository. This operation is used to get a full copy of a remote repository. Cloning a git repository is very simple. At an empty directory, just use the explorer context menu and select Git Clone....

The Clone Dialog will show

**A screenshot of a cell phone

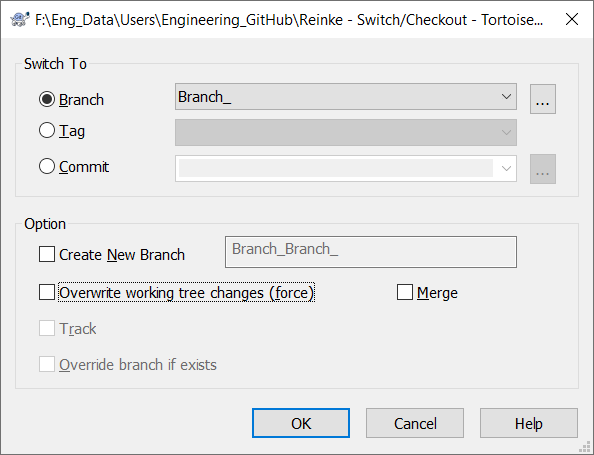
Description automatically generated**

* URL: Input repository URL address, which you will clone from. You can click Browse to browse directory.
* Directory: Input your local directory, which you will clone to. You can click Browse to browse directory.
* If you check the Load Putty Key checkbox, clone will auto load putty key file with Pageant. You can click ... to browse for a putty key file.
* Clone will checkout current HEAD to workspace automatically.

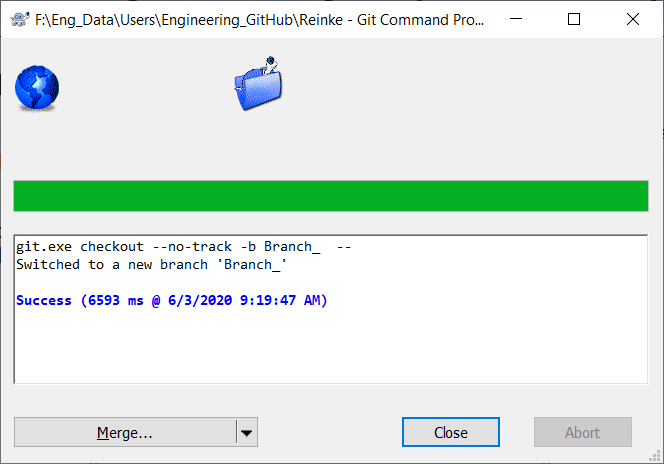
1. **Checking Out A Working Tree (Switch to commit):**

The Switch/Checkout dialog can be used to checkout a specific version to the working tree (i.e., all files are updated to match their state of the selected version). Normally, a specific version will be represented by a (local) branch which is set as the current branch.

Select a git repository directory in windows explorer Right click to pop up the context menu and select the command TortoiseGit → Switch/Checkout..., which brings up the following dialog box:



Click on Ok button to complete the process.



**If you enter a branch name at Create New Branch, a new branch will be created. Also, the new branch will be set as the current branch (HEAD).**

**You can click on the ... to browse the references/branches/log to choose a branch to checkout.**

**Check Overwrite working tree changes (force) will overwrite uncommitted changes in the working tree with the selected version.**

**When you selected a remote branch, you can check Track in order to track the remote branch. When you open the** [**push**](mk:@MSITStore:C:\Program%20Files\TortoiseGit\bin\TortoiseGit_en.chm::/tgit-dug-push.html)**,** [**pull**](mk:@MSITStore:C:\Program%20Files\TortoiseGit\bin\TortoiseGit_en.chm::/tgit-dug-pull.html) **or** [**sync**](mk:@MSITStore:C:\Program%20Files\TortoiseGit\bin\TortoiseGit_en.chm::/tgit-dug-sync.html) **dialog, the remote branch will be pre-selected automatically.**

|  |
| --- |
| Important |
| If you checkout/switch to a Tag or Commit, you should create a new branch. Otherwise you will work at "no branch" (detached HEAD state, i.e., there is no current branch. |

 **Export**

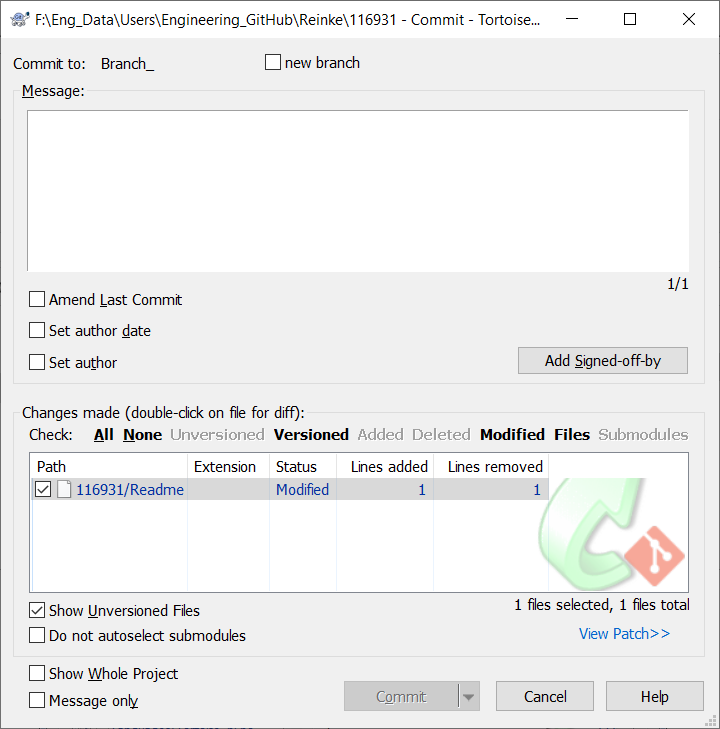
Sometimes you may want to create a local copy without any of those. git directories, e.g. to create a zipped tarball of your source.

1. **Committing Your Changes to The Repository**

Storing the changes, you made to your working tree is known as committing the changes. you can use TortoiseGit → Check for Modifications first, to see which files have changed locally.

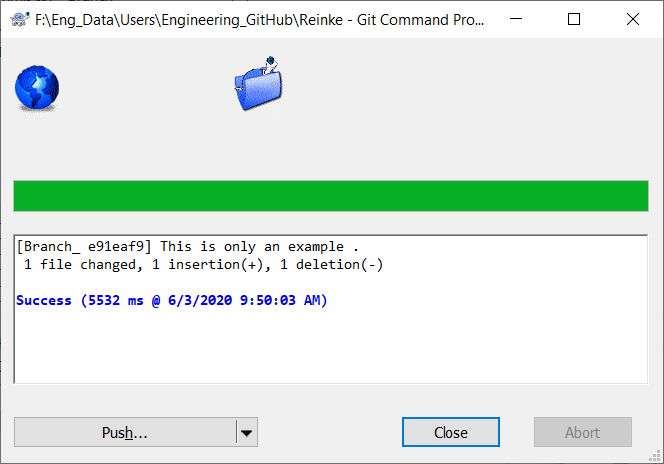
* 1. The Commit Dialog

If there are no conflicts, you are ready to commit your changes. Select any file and/or folders you want to commit, then TortoiseGit → Commit....This is only an example.



The commit dialog will show you every changed file, including added, deleted and unversioned files. If you don't want a changed file to be committed, just uncheck that file. If you want to include an unversioned file, just check that file to add it to the commit. Explain changes in the message window.

Default commit dialog just list select paths and their child directory files. If you want to list all files of project, you can just click Whole Project. See the below picture for commit progress.



Double clicking on any modified file in the commit dialog will launch the external diff tool to show your changes. The context menu will give you more options, as shown in the screenshot. You can also drag files from here into another application such as a text editor or an IDE.

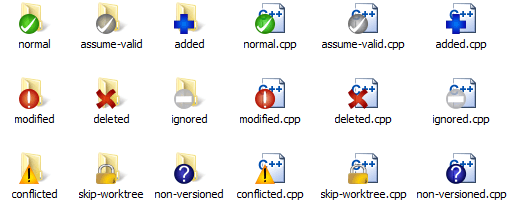
You can select or deselect items by clicking on the checkbox to the left of the item.

The columns displayed in the bottom pane are customizable. If you right click on any column header you will see a context menu allowing you to select which columns are displayed. You can also change column width by using the drag handle which appears when you move the mouse over a column boundary. These customizations are preserved, so you will see the same headings next time.

1. **Getting Status Information**

While you are working on your working tree you often need to know which files you have changed/added/removed or renamed, or even which files got changed and committed by others.

* 1. Icon Overlays description



A fresh checked out working tree has a green checkmark as overlay. That means the Git status is normal.

As soon as you start editing a file, the status changes to modified and the icon overlay then changes to a red exclamation mark. That way you can easily see which files were changed since you last updated your working tree and need to be committed.

If during an update a conflict occurs, then the icon changes to a yellow exclamation mark.

Staged. If you use "git update-index" to tell git this file will be committed, Git makes that file staged.

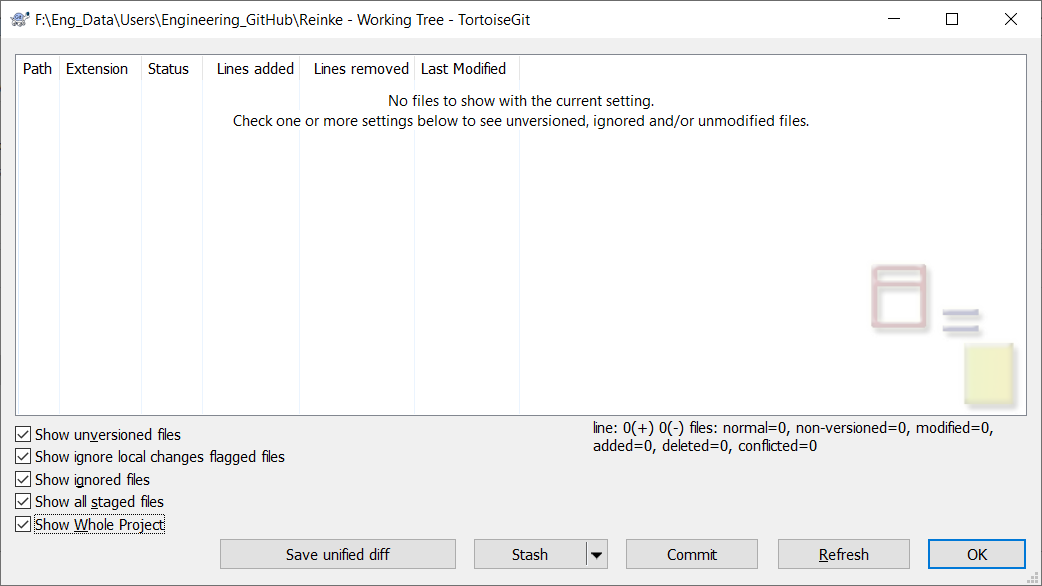
This icon shows you that some files or folders inside the current folder have been scheduled to be deleted from version control or a file under version control is missing in a folder.

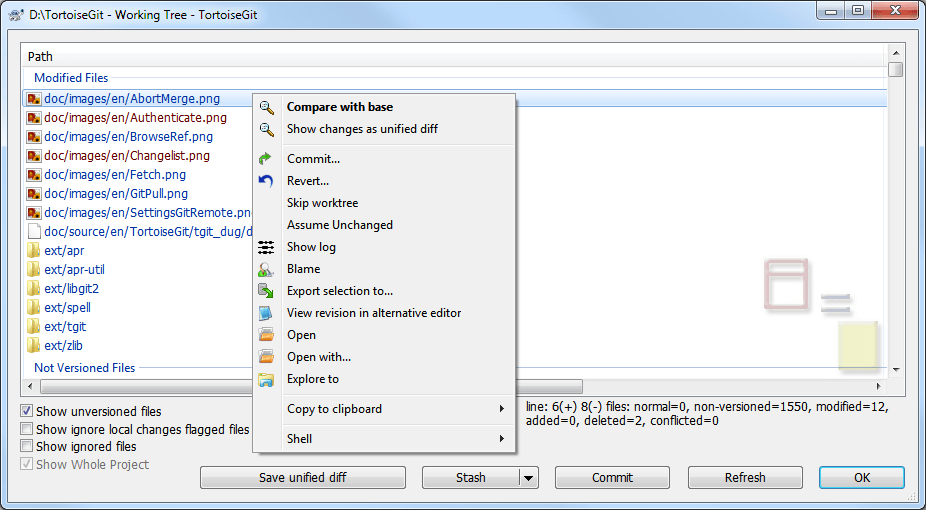
The plus sign tells you that a file or folder has been scheduled to be added to version control.

The bar sign tells you that a file or folder is ignored for version control purposes. This overlay is optional.

This icon shows files and folders which are not under version control, but have not been ignored. This overlay is optional.

In fact, you may find that not all of these icons are used on your system. This is because the number of overlays allowed by Windows is very limited and if you are also using an old version of Tortoise CVS or tools with overlay handlers such as SkyDrive, Drobox or Google Drive, then there are not enough overlay slots available. TortoiseGit tries to be a “Good Citizen (TM)” and limits its use of overlays to give other apps a chance too.

* 1. Status, see figures below with and with out modifications.



It's often very useful to know which files you have changed, and also which files got changed and committed by others. That's where the command TortoiseGit → Check for Modifications... comes in handy. This dialog will show you every file that has changed in any way in your working tree, as well as any unversioned files you may have.

The dialog uses color coding to highlight the status.

Blue: Locally modified items.

Purple: Added items. Items which have been added with history have a + sign in the Text status column, and a tooltip shows where the item was copied from.

Dark red: Deleted or missing items.

Green: Items modified locally and in the repository. The changes will be merged on update. These may produce conflicts on update.

Bright red: Items modified locally and deleted in repository or modified in repository and deleted locally. These willproduce conflicts on update.

Black: Unchanged and unversioned items.

This is the default color scheme, but you can customize those colors using the settings dialog.

From the context menu of the dialog you can show a diff of the changes. Check the local changes you made using Context Menu → Compare with Base. Check the changes in the repository made by others using Context Menu → Show Differences as Unified Diff.

You can also revert changes in individual files. If you have deleted a file accidentally, it will show up as Missing and you can use Revert to recover it.

Unversioned and ignored files can be sent to the recycle bin from here using Context Menu → Delete. If you want to delete files permanently (bypassing the recycle bin) hold the **Shift** key while clicking on Delete.

If you want to examine a file in detail, you can drag it from here into another application such as a text editor or IDE.

The columns are customizable. If you right click on any column header you will see a context menu allowing you to select which columns are displayed. You can also change column width by using the drag handle which appears when you move the mouse over a column boundary. These customizations are preserved, so you will see the same headings next time.

At the bottom of the dialog you have several options to select which entries to show (such as ignored or untracked/unversioned files). It is also possible to view all files which were marked as "Assume valid" or "Skip work tree" here (using Show ignore local changes flagged files). Resetting those flags (it's also possible to edit this flag using file properties in explorer on the Git tab).

* 1. **Viewing Diffs**

Often you want to look inside your files, to have a look at what you have changed. You can accomplish this by selecting a file which has changed and selecting Diff from TortoiseGit's context menu. This starts the external diff-viewer, which will then compare the current file with the pristine copy (BASE revision), which was stored after the last checkout or update

Even when not inside a working tree or when you have multiple versions of the file lying around, you can still display diffs:

Select the two files you want to compare in explorer (e.g. using **Ctrl** and the mouse) and choose Diff from TortoiseGit's context menu. The file clicked last (the one with the focus, i.e. the dotted rectangle) will be regarded as the later one.

1. **Pull and Fetch change**

This section talks about how to fetch or pull (i.e., download) changes from another repository. The difference between pull and fetch is:

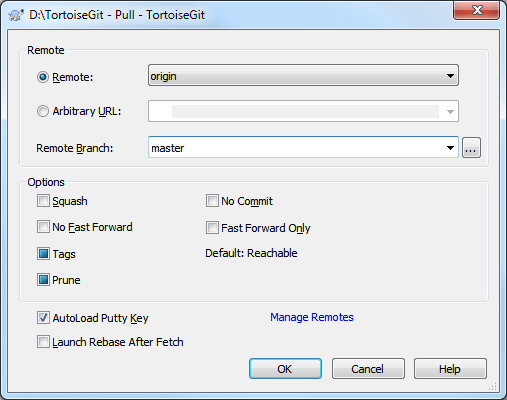
Fetch just downloads the objects and refs from a remote repository and normally updates the remote tracking branches. Pull, however, will not only download the changes, but also merges them - it is the combination of fetch and merge. The configured remote tracking branch is selected automatically.

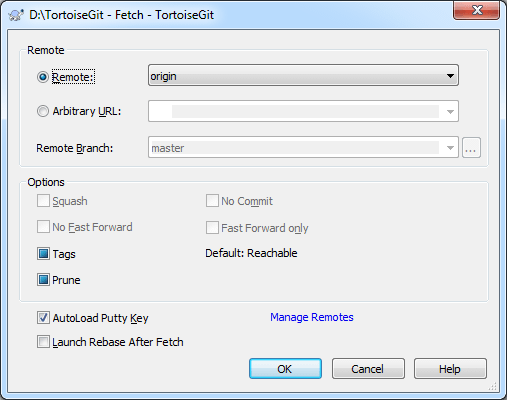
**Important**

Whenever you merge, it is possible a file was changed in both branches and that the changes cannot be merged automatically: This is called a "conflict" and needs to be manually resolved

A pull/fetch can be initiated by using TortoiseGit → Pull... or TortoiseGit → Fetch.... Fetching and pulling changes is also possible using the Sync dialog, however, there you have less options, but the sync dialog allows you to initiate other operations such as pushing and to see diffs and changes.

The fetch and pull dialog will open.





Remote Choose a configured remote repository (these can be changes using the Manage Remotes label). Instead of the configured repositories, you can also put the URL of another repository into the Arbitrary URL textbox.

If the current active branch has a remote tracked branch set, the remote branch and remote repository are automatically selected. A remote tracked branch can be set using the reference browser or using the push dialog.

Other: Input Other URL or local directory. You can click ... to browse directory.

If you check the Autoload Putty Key checkbox, a configured Putty key will be automatically loaded using Pageant.

Tags has three states (git 1.9 and later): Checked: All tags as well as branches are downloaded (--tags is passed to git), unchecked: No tags are downloaded (--no-tags is passed to git), and third state: use default behavior (based on remote.<name>.tagopt setting). Tags has three states (prior to git 1.9): Checked: Only all tags are downloaded but no branches are downloaded (--tags is passed to git), unchecked: No tags are downloaded (--no-tags is passed to git), and third state: use default behavior (based on remote.<name>.tagopt setting).

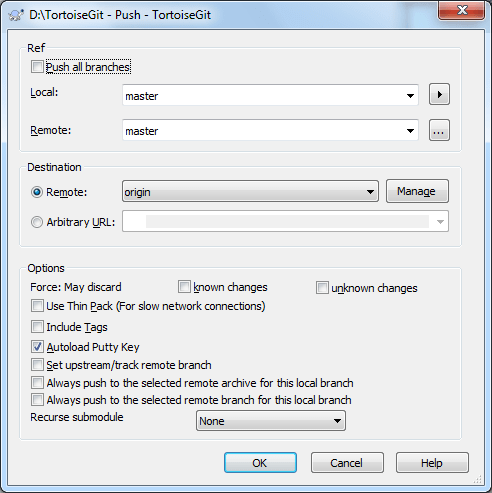
Prune has three states: True to remove remote-tracking branches which no longer exist on the remote, false: not to remove, and third state: use default behavior (based on remote.<name>.prune or fetch. Prune git setting which can be set on [“Remote”](mk:@MSITStore:C:\Program%20Files\TortoiseGit\bin\TortoiseGit_en.chm::/tgit-dug-settings.html#tgit-dug-settings-remote).

|  |
| --- |
| **Tip** |
| You can find more information about PuTTY and using ssh-key on website. There is also explained how you can use several accounts at the same time for a remote. |

|  |
| --- |
| **Conflicts** |
| Although major merge work is done by git automatically while pulling, a conflict may happen during cherry-picking (i.e., a file was modified in your current branch and in the branch, you are pulling.  Please note, that "REMOTE"/"theirs" in the conflict editor refers to the changes you’re on the branch you selected for pulling/merging and "LOCAL"/"mine" to your HEAD version in your working tree. |

* 1. **PUSH**

This section talks about how to push (i.e., send) changes to another repository. In order to perform a push open the push dialog using TortoiseGit → Push... . Pushing changes is also possible using the Sync dialog however, there you have less options, but the sync dialog allows you to initiate other operations such as pulling and to see diffs and changes.



* 1. Branch

Local: The source branch which will be pushed to the other repository. If the current branch or the selected local branch has a remote tracked branch set, the remote branch and remote repository are automatically selected. A remote tracked branch can be set using the reference browser or by using Set upstream/track remote branch. This can be overridden in this dialog by using one of the Always push to the selected remote ... options, so that for pushing a different branch is autos elected as for merging and pulling. Remote: The remote branch of the other repository.

* 1. Destination

Remote: Choose an already configured remote repository.

Arbitrary URL: The URL of a remote repository.

You must push change to a bare repository. Pushing changes to repository which has a working tree can lead to unexpected results.

* 1. Options

Force (May discard known changes) This allows remote repository to accept a safer non-fast-forward push. This can cause the remote repository to lose commits; use it with care. This can prevent from losing unknown changes from other people on the remote. It checks if the server branch points to the same commit as the remote-tracking branch (known changes). If yes, a force push will be performed. Otherwise it will be rejected. Since git does not have remote-tracking tags, tags cannot be overwritten using this option. These passes --force-with-lease option of git push command.

Force (May discard unknown changes) This allows remote repository to accept an unsafe non-fast-forward push. This can cause the remote repository to lose commits; use it with care. This does not check any server commits, so it is possible to lose unknown changes on the remote. Use this option with Include Tags to overwrite tags. This passes the traditional --force option of git push command.

Include Tags Also push tags to remote repository.

**Tip**

You can find more information about PuTTY and using ssh-keys in later sections. There is also explained how you can use several accounts at the same time for a remote.

Set upstream/track remote branch: After a successful push, the tracking relationship will be set between the pushed local branch and its remote tracking branch. This will auto select the remote branch automatically for pulling/pushing and merging.

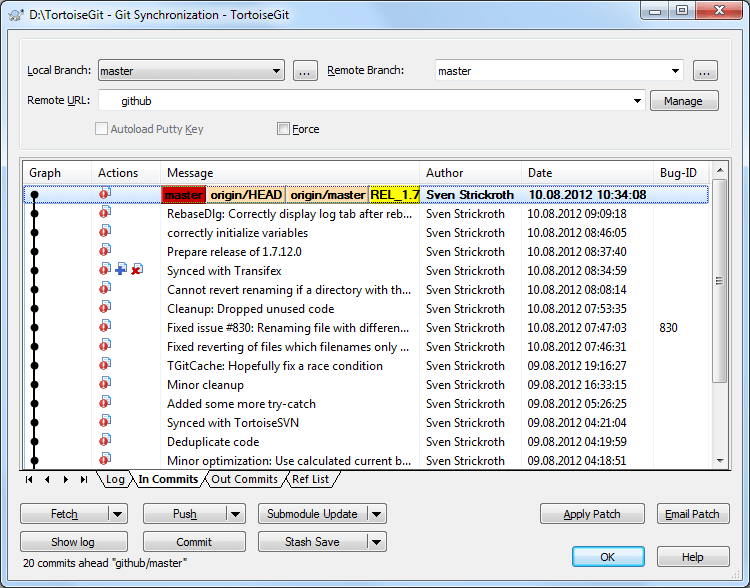
Always push to the selected remote branch for this local branch

Recurse submodule None: No checking. Check: Checks if the bounded commits of all submodules are present on the remote repositories. If any of the submodules are not pushed, the super project push will fail. On-demand: Checks if the bounded commits of all submodules are present on the remote repositories. If the submodules are not pushed yet, it will try to push them.

* 1. Sync

The Sync Dialog provides an interface for all operations related with remote repositories in one dialog. This includes push, pull, fetch, remote update, submodule update, send patch... However, the sync dialog provides less options as the regarding dialogs. The sync dialog can be opened using Sync.

The Sync Dialog will show.



* 1. Branch

Local Branch: The source branch which will push/pull to/from another repository. If the current branch or the selected local branch has a remote tracked branch set, the remote branch and remote repository are automatically selected. Remote Branch: The remote branch of a remote repository.

Remote Branch: The remote branch of a remote repository.

* 1. Destination

URL: Choose remote repository or input remote repository URL. Manage Add new remote name.

* 1. Options

Force Overwrite Existing Branch(May discard changes) .Autoload putty key Autoload putty key when push or pull

* 1. Xxx
  2. xxxx
  3. xx

1. **SSS**
2. **Xxx**