

---

# **Git Extensions Documentation**

*Release 3.5*

**Contributors**

**Mar 11, 2021**

---

# Contents

---

<b>1</b>	<b>Git Extensions</b>	<b>1</b>
1.1	Features . . . . .	1
1.2	Video tutorials . . . . .	1
1.3	Links . . . . .	2
<b>2</b>	<b>Getting Started</b>	<b>3</b>
2.1	Installation . . . . .	3
2.2	Portable . . . . .	7
2.3	Settings . . . . .	7
2.4	Dashboard . . . . .	9
2.5	Create new repository . . . . .	10
2.6	Open repository . . . . .	11
2.7	Clone repository . . . . .	11
2.8	Clone Github repository . . . . .	12
<b>3</b>	<b>Settings</b>	<b>14</b>
3.1	Git Extensions . . . . .	15
3.1.1	General . . . . .	16
3.1.2	Appearance . . . . .	17
3.1.2.1	Colors . . . . .	19
3.1.2.2	Fonts . . . . .	20
3.1.3	Revision Links . . . . .	20
3.1.4	Build server integration . . . . .	23
3.1.5	Scripts . . . . .	24
3.1.6	Hotkeys . . . . .	25
3.1.7	Shell Extension . . . . .	26
3.1.8	Advanced . . . . .	26
3.1.8.1	Confirmations . . . . .	27
3.1.9	Detailed . . . . .	28
3.1.9.1	Browse repository window . . . . .	29
3.1.9.2	Commit dialog . . . . .	29
3.1.9.3	Diff Viewer . . . . .	30
3.1.10	SSH . . . . .	30
3.2	Git . . . . .	31
3.2.1	Paths . . . . .	31
3.2.2	Config . . . . .	32
3.2.3	Advanced . . . . .	33

3.3	Plugins	33
<b>4</b>	<b>Browse Repository</b>	<b>34</b>
4.1	View revision graph	34
4.2	Search or filter the commit history	36
4.2.1	Quick search in history	36
4.2.2	Go to a specific commit	37
4.2.3	Filter history	37
<b>5</b>	<b>File history</b>	<b>40</b>
5.1	Commit	41
5.2	Diff	41
5.3	View	42
5.4	Blame	43
<b>6</b>	<b>Commit</b>	<b>44</b>
6.1	Commit changes	44
6.1.1	Staging changes	46
6.1.2	Staging selected lines	46
6.1.3	Undoing or resetting changes	47
6.1.4	Making the commit	48
6.2	Amend commit	50
<b>7</b>	<b>Stash</b>	<b>51</b>
7.1	Revision graph	52
<b>8</b>	<b>Tag</b>	<b>53</b>
8.1	Create tag	53
8.2	Delete tag	54
<b>9</b>	<b>Branches</b>	<b>56</b>
9.1	Create branch	57
9.1.1	Orphan branches	58
9.2	Checkout branch	58
9.3	Merge branches	58
9.4	Rebase branch	61
9.5	Interactive rebase	62
9.6	Delete branch	63
<b>10</b>	<b>Patches</b>	<b>64</b>
10.1	Create patch	65
10.2	Apply patches	65
<b>11</b>	<b>Remotes</b>	<b>67</b>
11.1	Manage remote repositories	67
11.2	Git Credential Manager	69
11.3	Create SSH key	69
11.3.1	PuTTY and github	69
11.3.2	OpenSSH and github	71
11.4	Pull changes	72
11.5	Push changes	74
<b>12</b>	<b>Merge Conflicts</b>	<b>76</b>
12.1	Handle merge conflicts	76

<b>13</b>	<b>Modify Git history</b>	<b>80</b>
13.1	Cherry pick commit . . . . .	80
13.2	Revert commit . . . . .	82
13.3	Modify the last commit . . . . .	84
13.4	Modify an older commit . . . . .	84
13.4.1	Interactive rebase . . . . .	85
13.4.2	Using autosquash rebase feature . . . . .	86
13.4.3	Edit/reword commit . . . . .	88
<b>14</b>	<b>Notes</b>	<b>89</b>
<b>15</b>	<b>Submodules</b>	<b>91</b>
15.1	Manage submodules . . . . .	91
15.2	Add submodule . . . . .	92
<b>16</b>	<b>Worktrees</b>	<b>93</b>
<b>17</b>	<b>Maintenance</b>	<b>94</b>
17.1	Compress Git database . . . . .	94
17.2	Recover lost objects . . . . .	95
17.3	Fix user names . . . . .	97
17.4	Ignore files . . . . .	98
<b>18</b>	<b>Translations</b>	<b>99</b>
18.1	Change language . . . . .	99
18.2	Translate Git Extensions . . . . .	100
<b>19</b>	<b>Windows Explorer</b>	<b>101</b>
<b>20</b>	<b>Visual Studio</b>	<b>103</b>
20.1	Menu . . . . .	103
20.2	Toolbar . . . . .	104
20.3	Context menu . . . . .	105
<b>21</b>	<b>Command line</b>	<b>106</b>
21.1	Git Extensions command line . . . . .	106
<b>22</b>	<b>Appendix</b>	<b>109</b>
22.1	Git Cheat Sheet . . . . .	110
<b>23</b>	<b>Plugins</b>	<b>112</b>
23.1	Auto compile SubModules . . . . .	112
23.2	Bitbucket Server . . . . .	112
23.3	Create local tracking branches . . . . .	113
23.4	Delete obsolete branches . . . . .	113
23.5	Find large files . . . . .	113
23.6	Gerrit Code Review . . . . .	113
23.7	GitHub . . . . .	113
23.8	GitFlow . . . . .	114
23.9	Gource . . . . .	114
23.10	Impact Graph . . . . .	114
23.11	Jira Commit Hint . . . . .	115
23.12	Periodic background fetch . . . . .	116
23.13	Proxy Switcher . . . . .	116
23.14	Release Notes Generator . . . . .	116
23.15	Statistics . . . . .	116

Git Extensions is a toolkit aimed at making working with Git under Windows more intuitive. The shell extension will integrate in Windows Explorer and presents a context menu on files and directories. There is also a Visual Studio extension to use Git from the Visual Studio IDE.

### 1.1 Features

- Windows Explorer integration for Git
- Feature rich user interface for Git
- 32bit and 64bit support
- Visual Studio extension (2015-2017)

Specific in 2.5x releases:

- Visual Studio (2010 - 2015) add-in
- Runs under Linux or Mac OS X using [Mono](#)
- Basic SVN functionality

For description of 2.x specific features, see the [2.x documentation](#)

### 1.2 Video tutorials

There are video tutorials for some basic functions on YouTube (made for older Git Extensions versions).

1. [Clone](#)
2. [Commit changes](#)
3. [Push changes](#)
4. [Pull changes](#)

5. Handle merge conflicts

## 1.3 Links

See the following links for the Git Extensions download page, source code and documentation.

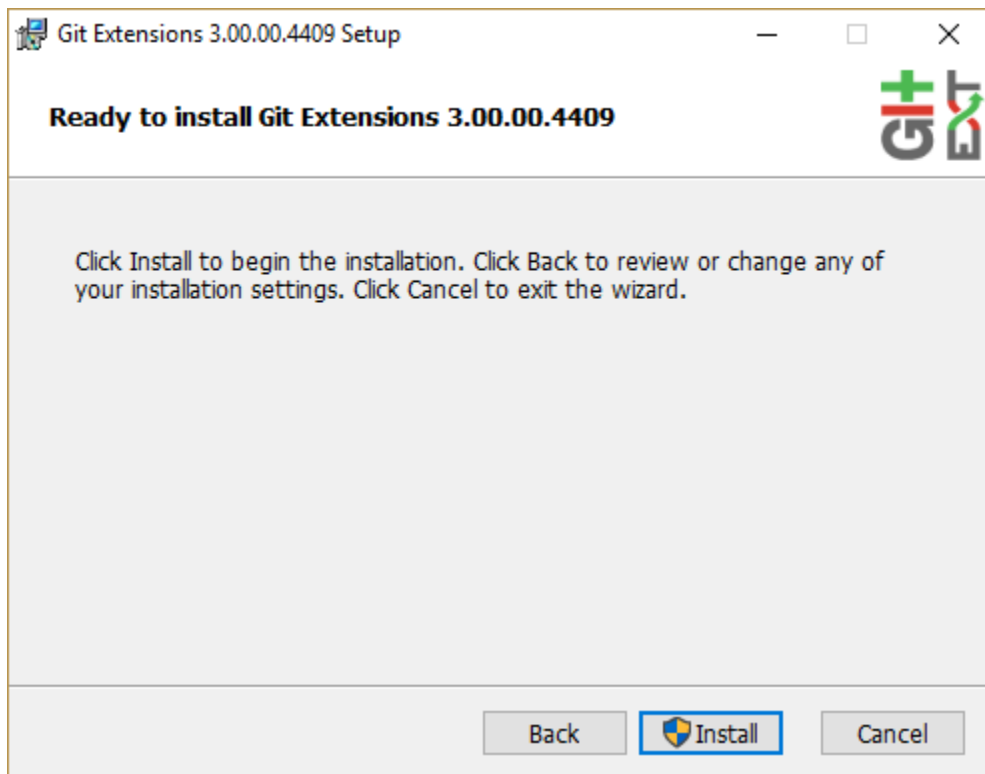
- Download page: <https://sourceforge.net/projects/gitextensions/>
- Source Code: <https://github.com/gitextensions/gitextensions>
- Source Code Issue tracker: <https://github.com/gitextensions/gitextensions/issues>
- Documentation: <https://github.com/gitextensions/GitExtensionsDoc>
- Documentation Issue tracker: <https://github.com/gitextensions/GitExtensionsDoc/issues>
- Wiki: <https://github.com/gitextensions/gitextensions/wiki>

Please feel free to raise any issues with Git Extensions or its documentation at the appropriate Issue tracker link as shown above.

### 2.1 Installation

This section only covers Git Extensions installation, you will need to have Git for Windows installed: <https://git-scm.com/download/win>

The Git Extensions installer can be found on [GitHub](#).



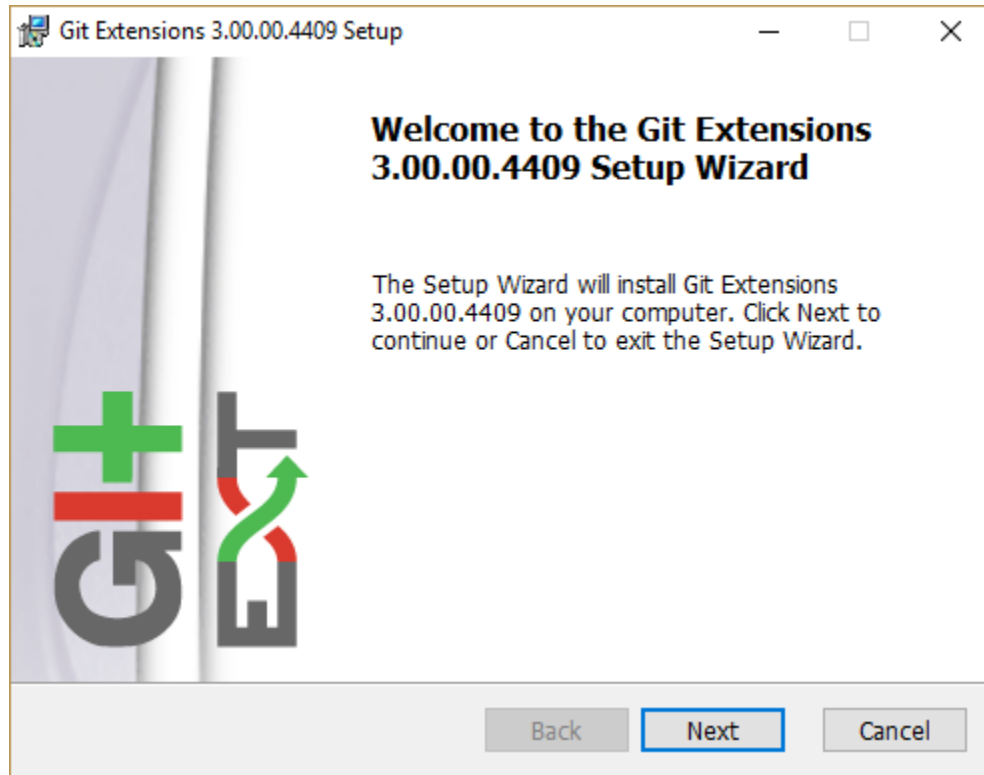


Fig. 1: Start installation.

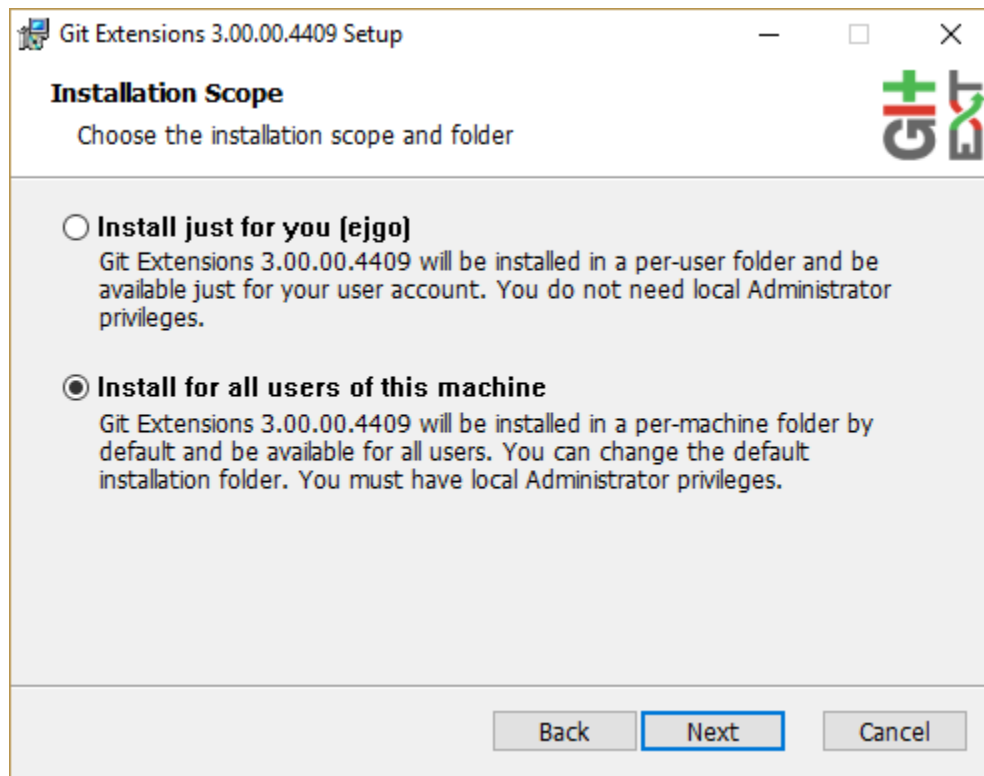


Fig. 2: Installation scope.



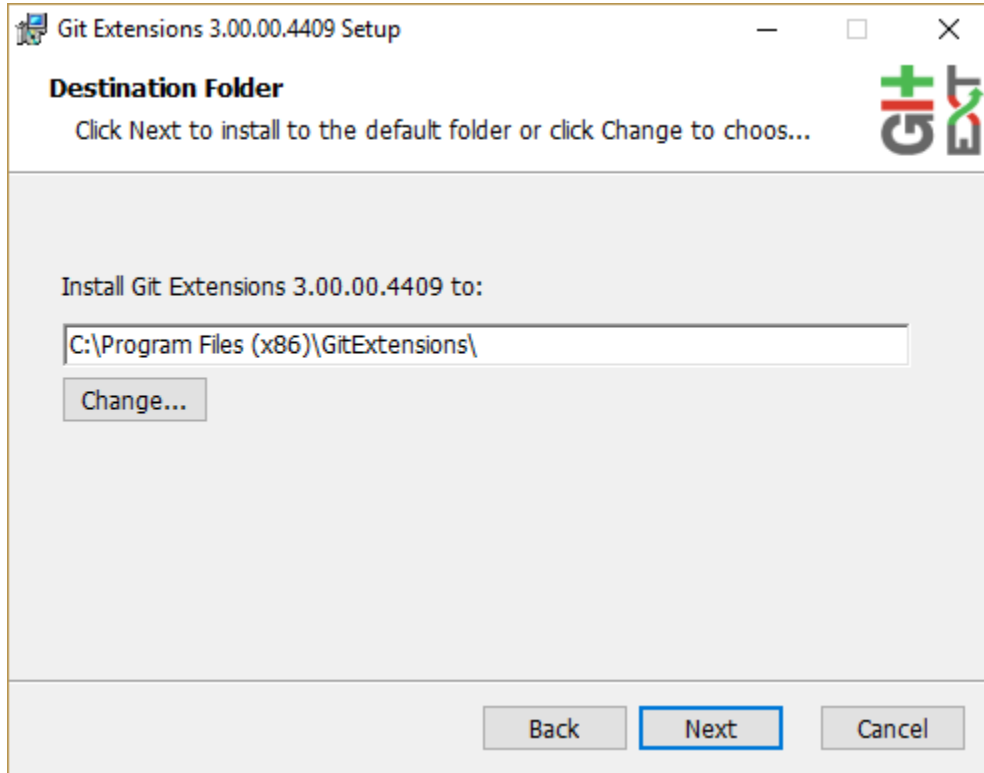


Fig. 3: Destination folder.

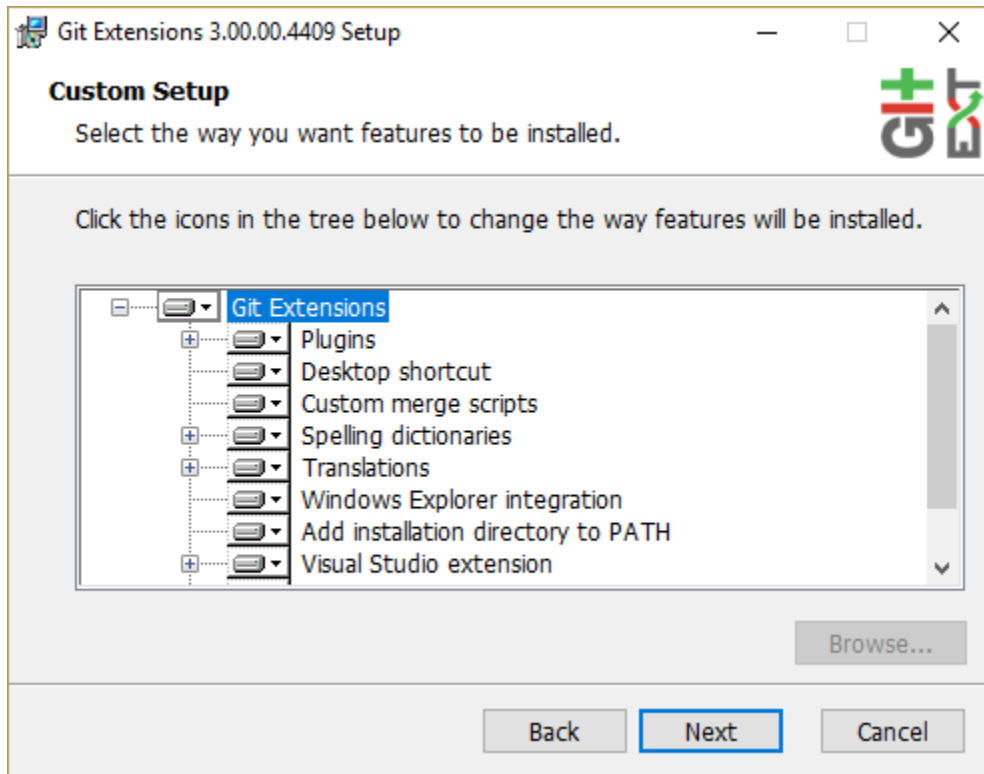


Fig. 4: Choose the options to install.

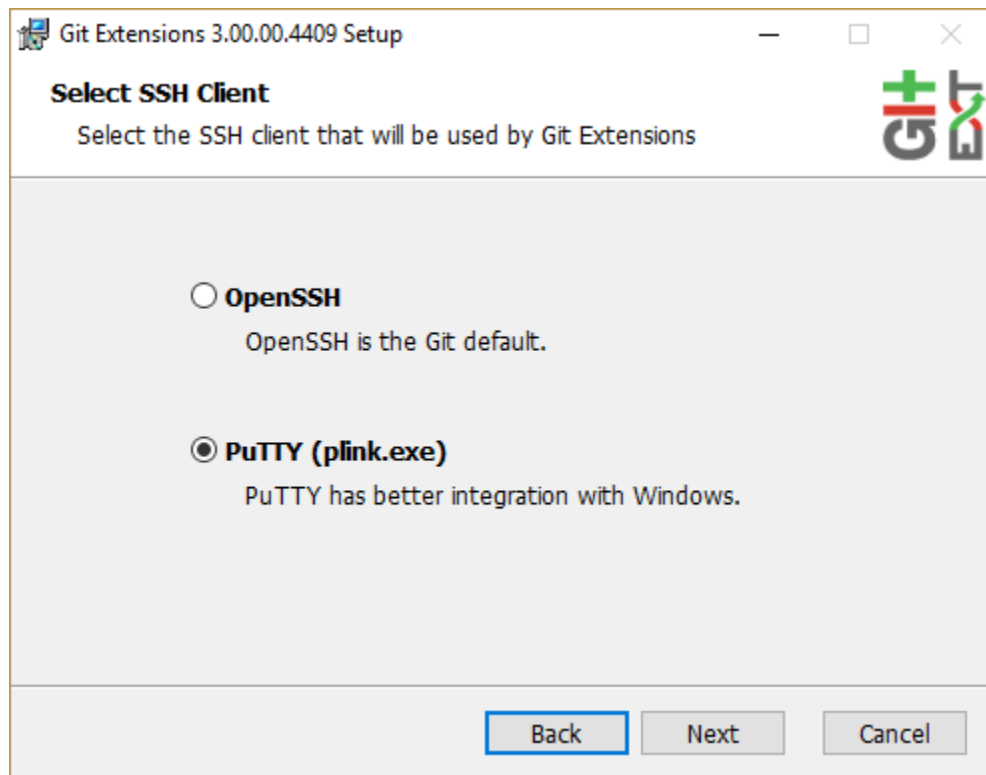


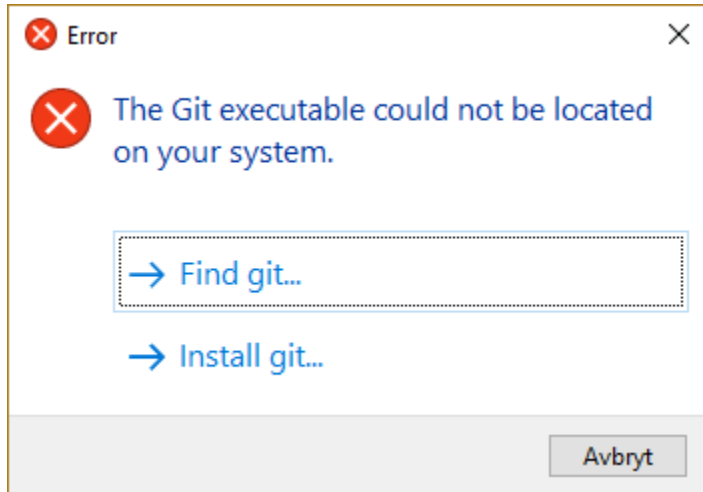
Fig. 5: Choose the SSH client to use. PuTTY is the default because it has better Windows integration, but Pageant must be running.

## 2.2 Portable

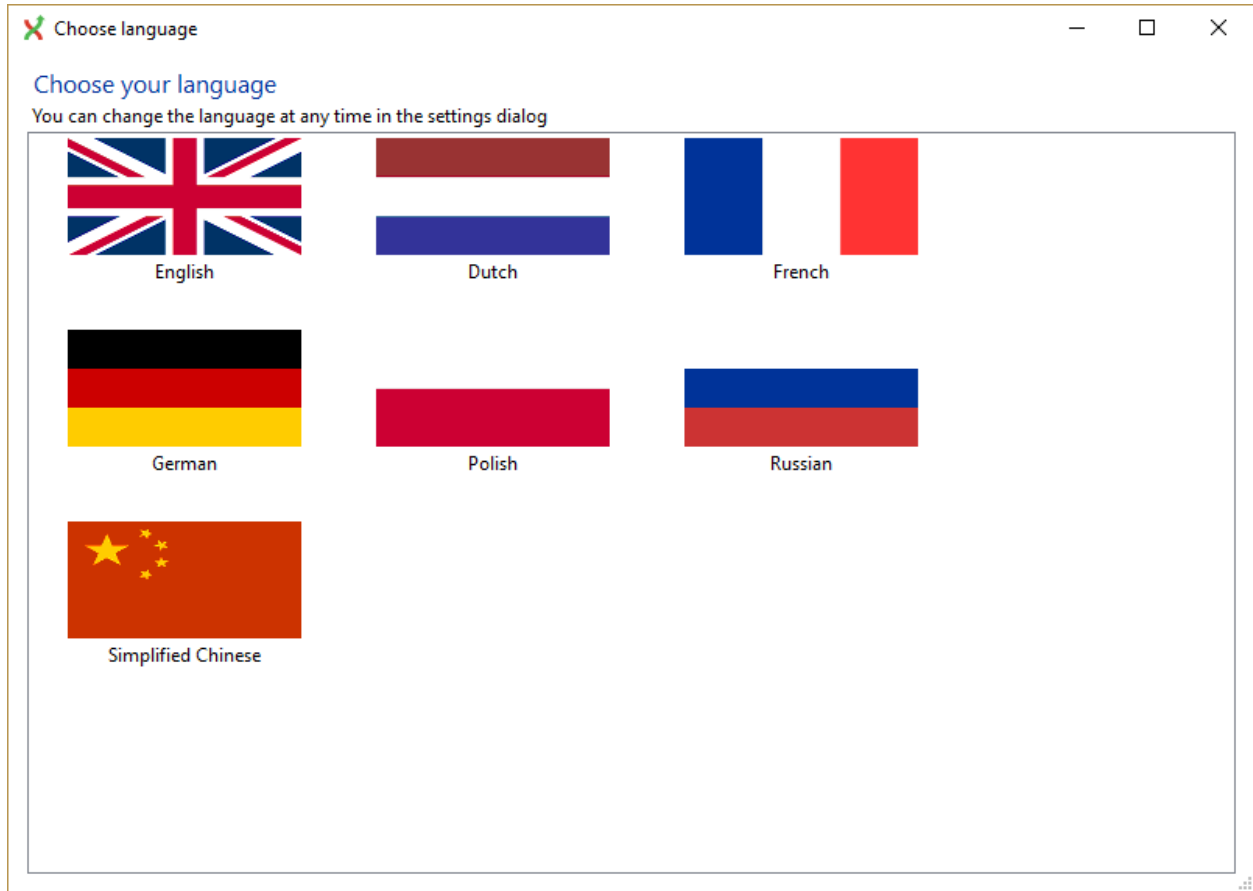
Git Extensions is also distributed as a portable .zip file, that only require unpacking. Some features like Windows shell integration and Visual Studio integration is not available with this package.

## 2.3 Settings

Git must be installed prior to starting Git Extensions:

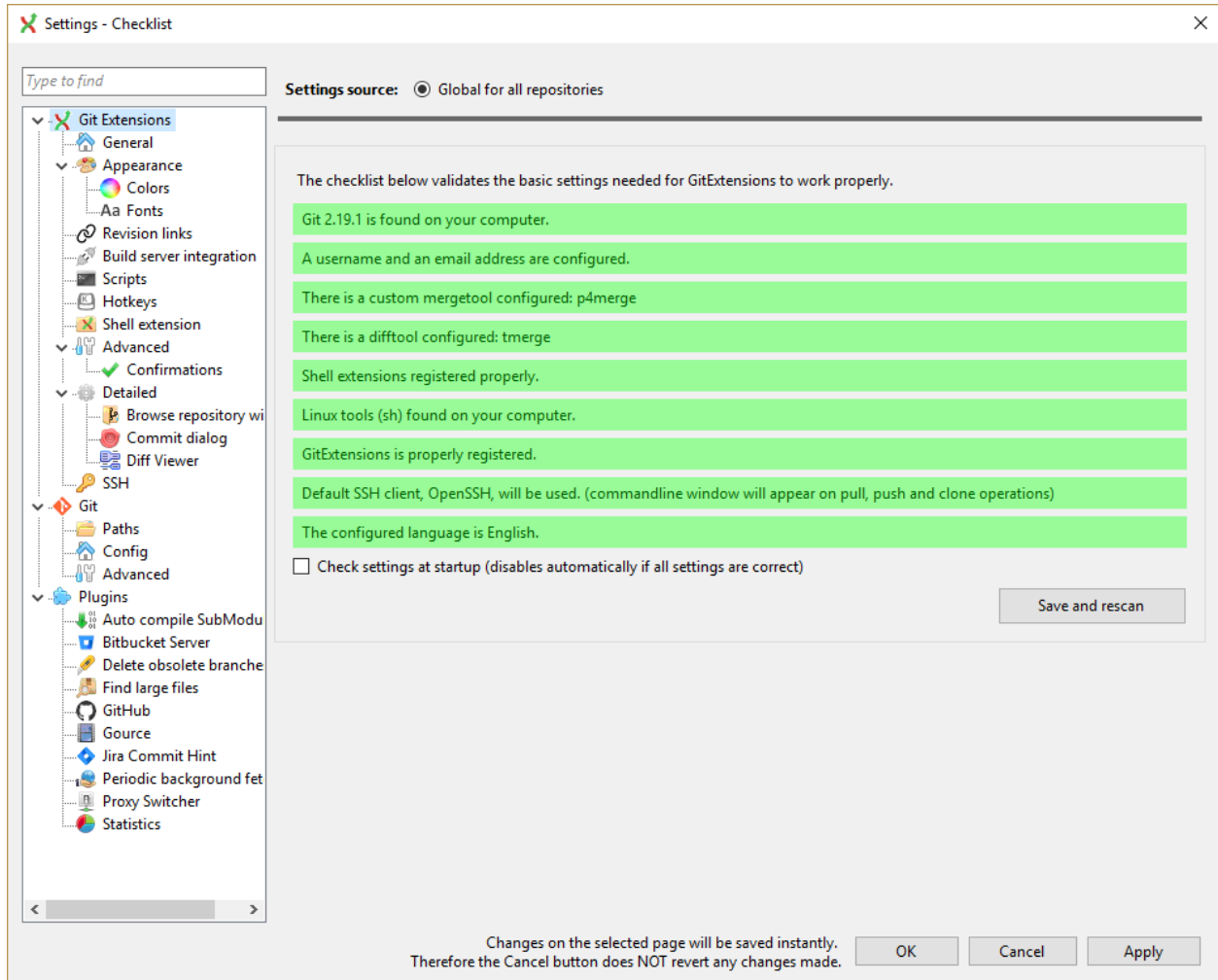


First selection is language (depends on the installed languages):



All settings will be verified when Git Extensions is started for the first time. If Git Extensions requires any settings to be changed, the Settings dialog will be shown. All incorrect settings will be marked in red (for instance if the Git version is unsupported) and orange for not recommended setting (like that Git version is older than recommended). You can ask Git Extensions to try to fix the setting for you by clicking on it. When installing Git Extensions for the first time, you will normally be required to configure your username and email address.

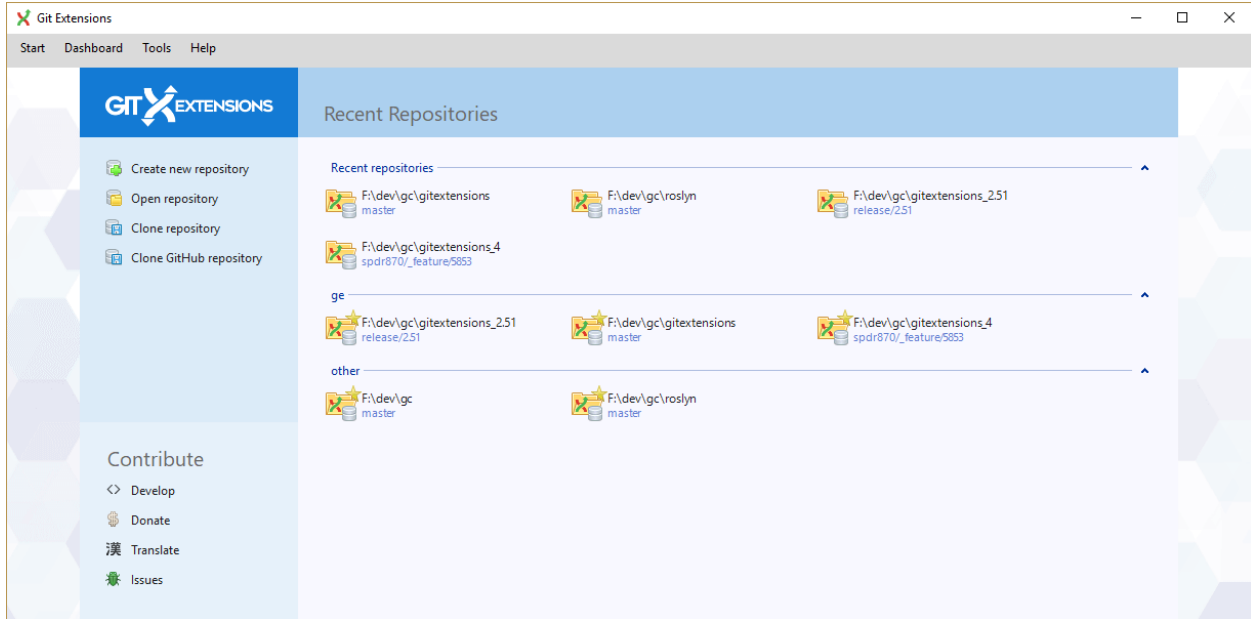
The settings dialog can be invoked at any time by selecting `Settings` from the `Tools` menu option.



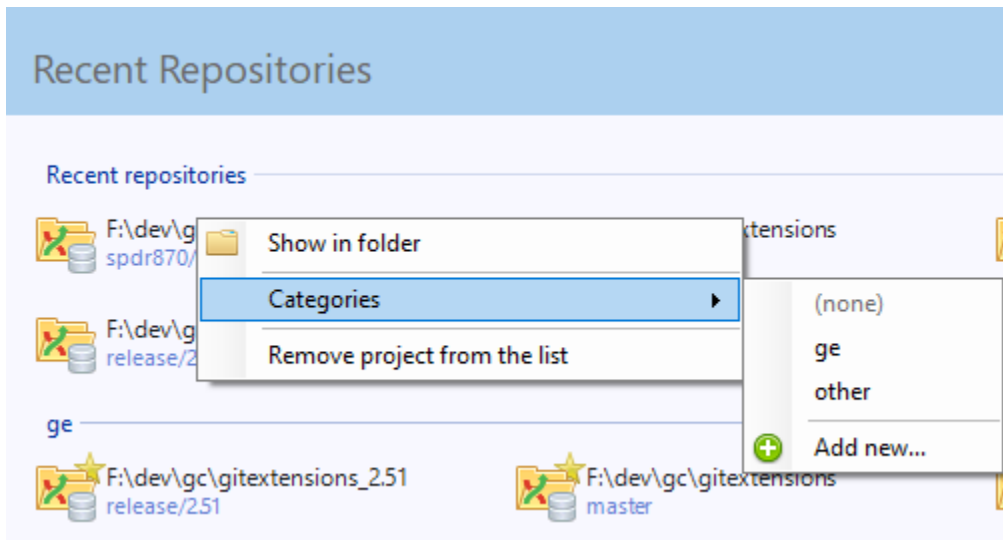
For further information see [Settings](#).

## 2.4 Dashboard

The dashboard contains the most common tasks, recently opened repositories and favourites. Favourite repositories can be added, grouped under Category headings in the right panel.



Recent Repositories can be moved to favourites using the repository context menu. Choose *Categories* / *Add new* to create a new category and add the repository to it, or you can add the repository to an existing category (e.g. 'Currents' as shown below).

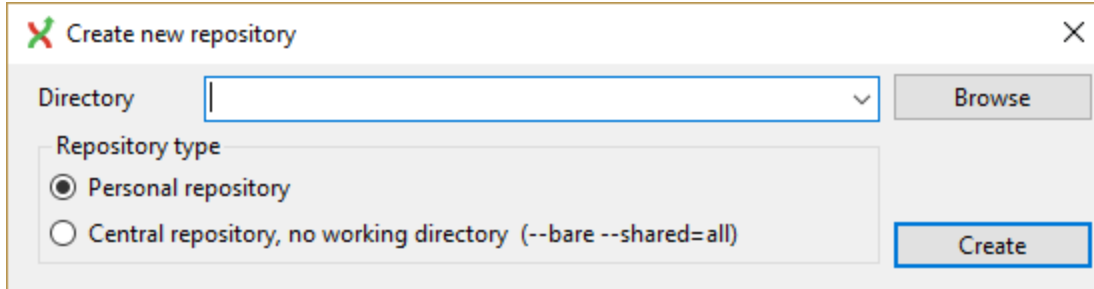


To open an existing repository, simply click the link to the repository, or select *Open repository* (from where you can select a repository to open from your local file system).

To create a new repository, one of the following options under *Common Actions* can be selected.

## 2.5 Create new repository

When you do not want to work on an existing project, you can create your own repository using this option.



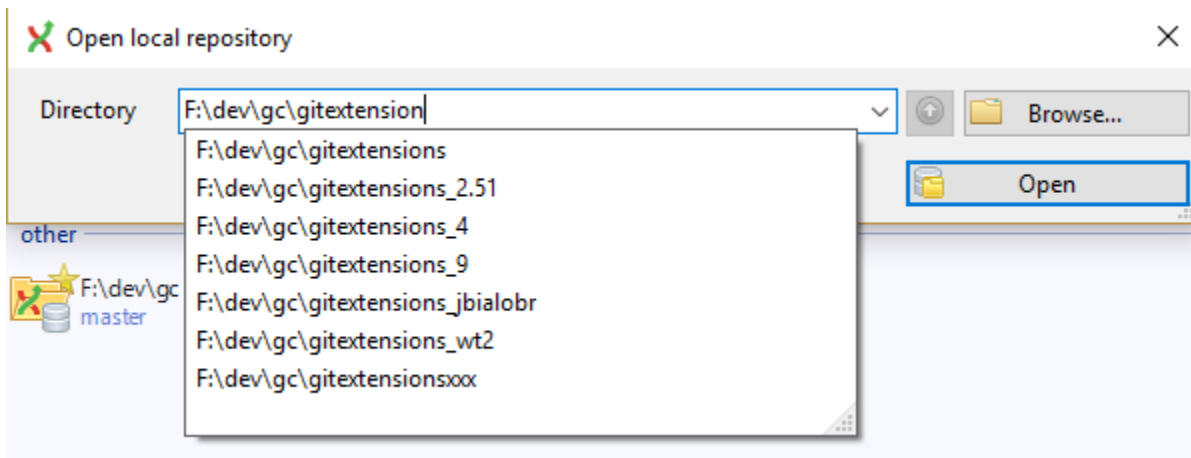
Select a directory where the repository is to be created. You can choose to create a Personal repository or a Central repository.

A personal repository looks the same as a normal working directory but has a directory named `.git` at the root level containing the version history. This is the most common repository.

Central repositories only contain the version history. Because a central repository has no working directory you cannot checkout a revision in a central repository. It is also impossible to merge or pull changes in a central repository. This repository type can be used as a public repository where developers can push changes to or pull changes from.

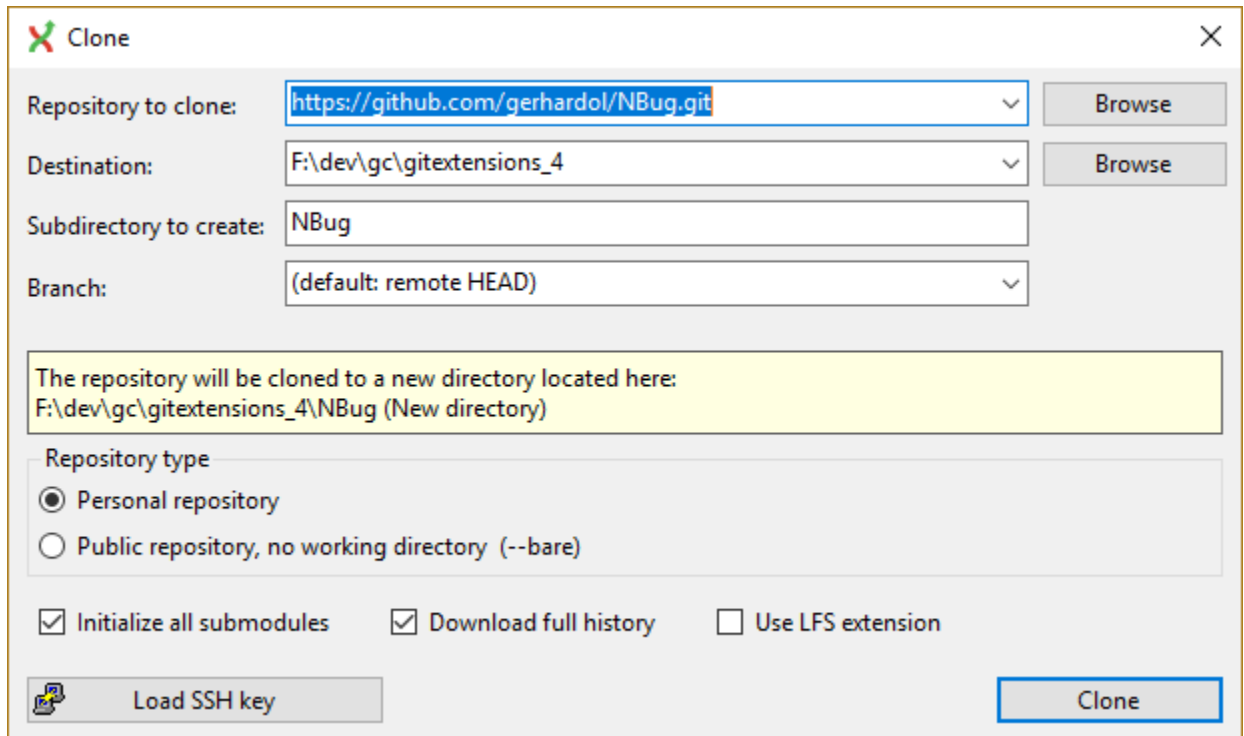
## 2.6 Open repository

Opens a Git repo already existing on the file system.



## 2.7 Clone repository

You can clone an existing repository using this option.



The repository you want to clone could be on a network share or could be a repository that is accessed through an internet or intranet connection. Depending on the protocol (http or ssh) you might need to load a SSH key into PuTTY. You also need to specify where the cloned repository will be created and the initial branch that is checked out. If the cloned repository contains submodules, then these can be initialised using their default settings if required.

There are two different types of repositories you can create when making a clone. A personal repository contains the complete history and also contains a working copy of the source tree. A central (bare) repository is used as a public repository where developers push the changes they want to share with others to. A central repository contains the complete history but does not have a working directory like personal repositories.

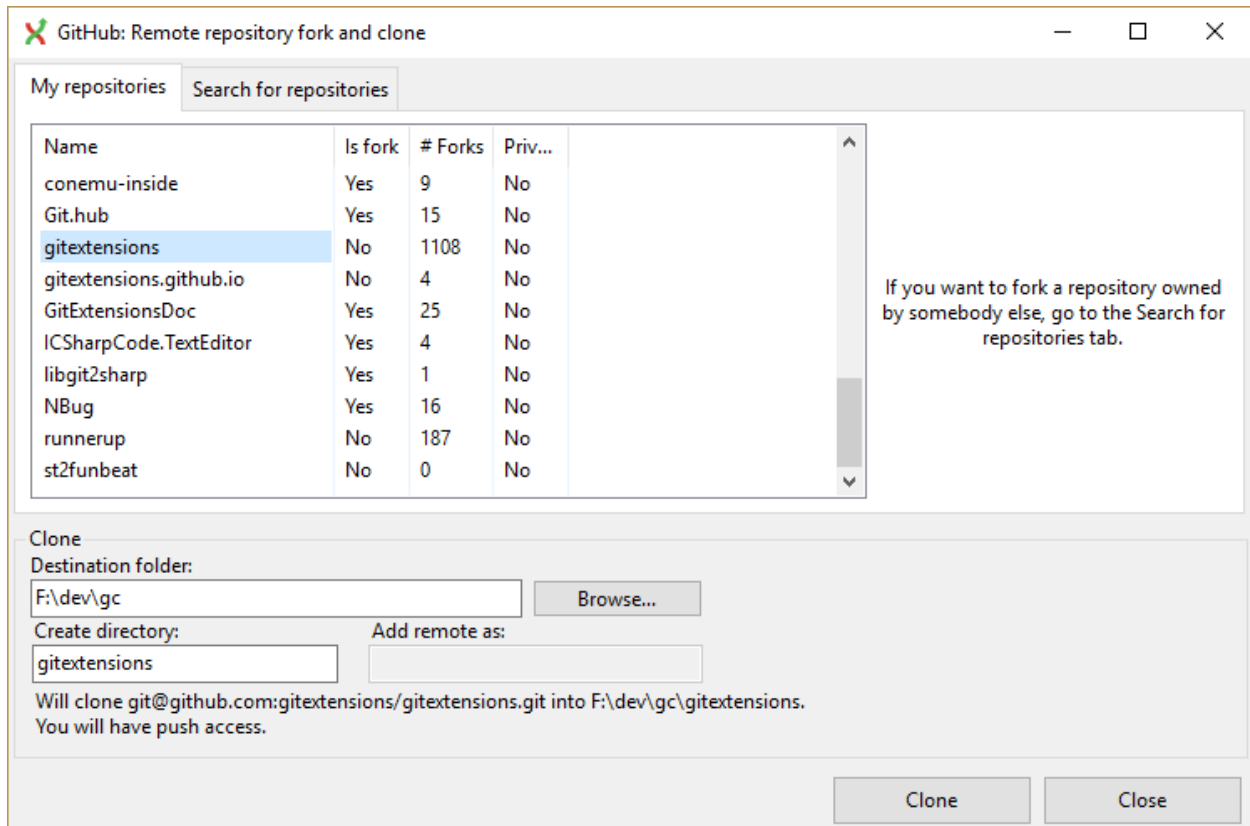
## 2.8 Clone Github repository

This option allows you to

- 1) Fork a repository on GitHub so it is created in your personal space on GitHub.
- 2) Clone any repositories on your personal space on GitHub so that it becomes a local repository on your machine.

You can see your own personal repositories on GitHub, and also search for repositories using the `Search for repositories` tab.





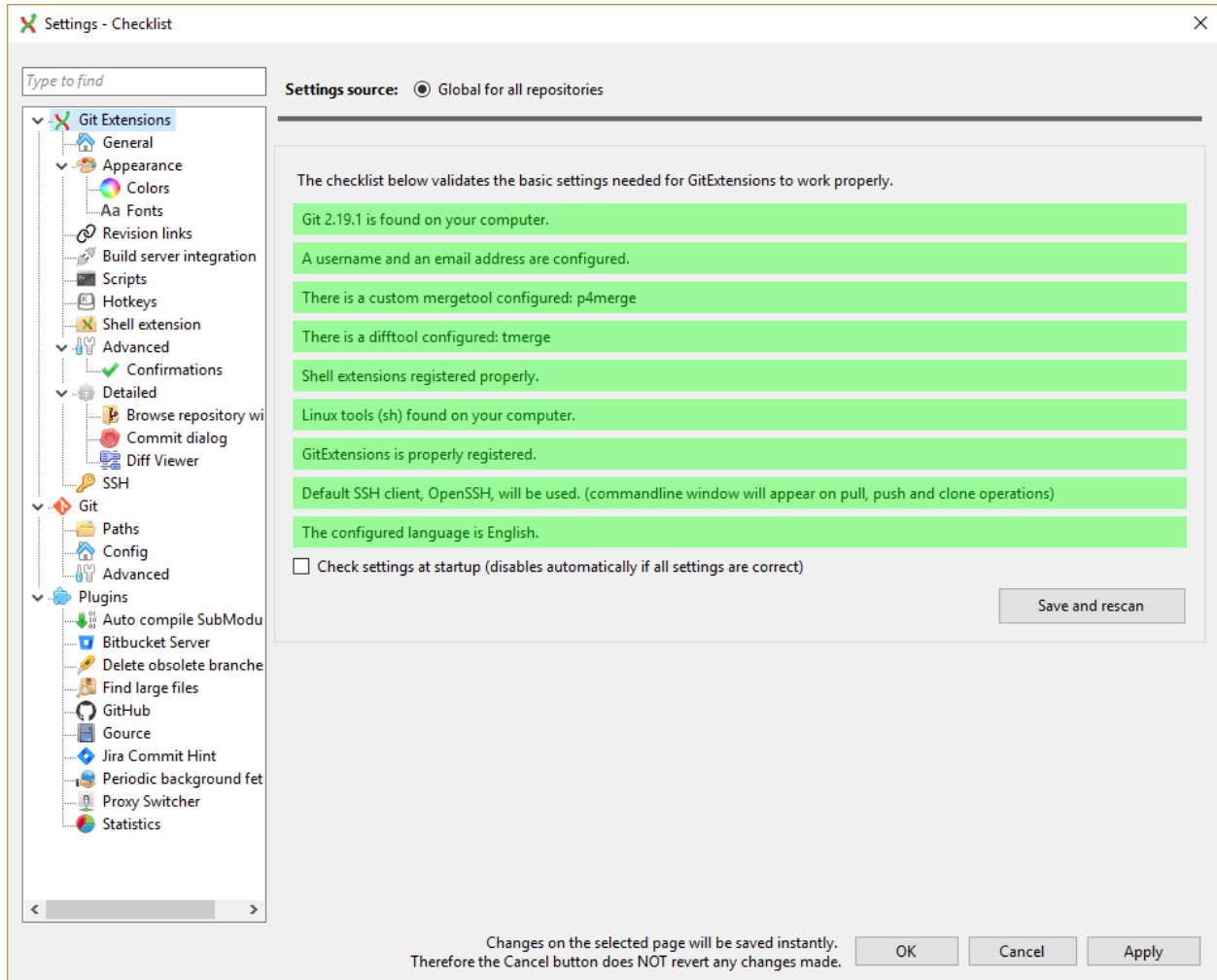
## CHAPTER 3

---

### Settings

---

The settings dialog can be invoked at any time by selecting `Settings` from the `Tools` menu option.



The following buttons are always available on any page of the Settings dialog. Sometimes the `Cancel` button has no effect for the page - this will be noted on the page in the area next to the buttons.

Button	Description
OK	Save any entered changes made in <i>any</i> settings page and close the Settings dialog.
Cancel	Any entered changes in <i>any</i> settings page are <i>not</i> saved. The Settings dialog is closed.
Apply	Any entered changes in <i>any</i> settings page are saved.

Settings that are specific to Git Extensions and apply globally will be stored in a file called `GitExtensions.settings` either in the user's application data path or with the program. The location is dependent on the `Is-Portable` setting in the `GitExtensions.exe.config` file that is with the program. Settings that are specific to Git Extensions but apply to only the current repository will be stored in a file of the same name, `GitExtensions.settings`, but in either the root folder of the repository or the `.git` folder of the repository, depending on whether or not they are distributed with that repository.

### 3.1 Git Extensions

This page is a visual overview of the minimal settings that Git Extensions requires to work properly. Any items highlighted in red should be configured by clicking on the highlighted item.

This page contains the following settings and buttons.

**Check settings at startup**

Forces Git Extensions to re-check the minimal set of required settings the next time Git Extensions is started. If all settings are 'green' this will be automatically unchecked.

**Save and rescan**

Saves any setting changes made and re-checks the settings to see if the minimal requirements are now met.

### 3.1.1 General

This page contains general settings for Git Extensions.

**Performance****Show number of changed files on commit button**

When enabled, the number of pending commits are shown on the toolbar as a figure in parentheses on the Commit button. Git Extensions must be stopped and restarted to activate changes to this option. Turn this off if you experience slowdowns.

**Show number of changed files for artificial commits**

If artificial commits are enabled in the revision graph, show the pending commits as well as a tool tip with a summary of changes.

**Show current working directory changes as an artificial commit.**

When enabled, two artificial revisions are added to the revision graph. The first shows the worktree (current working directory) status. The second shows the commit index (staged).

**Use FileSystemWatcher to check if index is changed**

Monitor if the Git index is changed due to changes outside of Git Extensions, if so show this in the Refresh button in Browse.

**Show stash count on status bar in browse window**

When you use the stash a lot, it can be useful to show the number of stashed items on the toolbar. This option causes serious slowdowns in large repositories and is turned off by default.

**Show submodule status in browse window**

Show the status for submodules (as well as supermodules) in the dropdown menu in Browse. The status is updated if *Show number of changed files for artificial commits* is enabled and the number of artificial commits is updated. (Changes in supermodules is not monitored). This option causes serious slowdowns in large repositories and is turned off by default.

**Check for uncommitted changes in checkout branch dialog**

Git Extensions will not allow you to checkout a branch if you have uncommitted changes on the current branch. If you select this option, Git Extensions will display a dialog where you can decide what to do with uncommitted changes before swapping branches.

**Limit number of commits that will be loaded at start-up**

This number specifies the maximum number of commits that Git Extensions will load when it is started. These commits are shown in the Revision Graph window. To see more commits, then this setting will need to be adjusted and Git Extensions restarted.

**Behaviour****Close Process dialog when process succeeds**

When a process is finished, close the process dialog automatically. Leave this option off if you want to see the result of processes. When a process has failed, the dialog will automatically remain open.

**Show console window when executing git process**

Git Extensions uses command line tools to access the git repository. In some environments it might be useful to see the command line dialog when a process is executed. An option on the command line dialog window displayed allows this setting to be turned off.

**Use patience diff algorithm**

Use the Git 'patience diff' algorithm instead of the default. This algorithm is useful in situations where two files have diverged significantly and the default algorithm may become 'misaligned', resulting in a totally unusable conflict file.

**Include untracked files in stash**

If checked, when a stash is performed as a result of any action except a manual stash request, e.g. checking out a new branch and requesting a stash then any files not tracked by git will also be saved to the stash.

**Follow renames in file history (experimental)**

Try to follow file renames in the file history.

**Follow exact renames and copies only**

Follow file renames and copies for which similarity index is 100%. That is when a file is renamed or copied and is committed with no changes made to its content.

**Open last working dir on startup**

When starting Git Extensions, open the last used repository (bypassing the Dashboard).

**Default clone destination**

Git Extensions will pre-fill destination directory input with value of this setting on any form used to perform repository clone.

**Revision grid quick search timeout [ms]**

The timeout (milliseconds) used for the quick search feature in the revision graph. The quick search will be enabled when you start typing and the revision graph has the focus.

**Email settings for sending patches****SMTP server name**

SMTP server to use for sending patches.

**Port**

SMTP port number to use.

**Use SSL/TLS**

Check this box if the SMTP server uses SSL or TLS.

## 3.1.2 Appearance

This page contains settings that affect the appearance of the application.

**General****Show relative date instead of full date**

Show relative date, e.g. 2 weeks ago, instead of full date. Displayed on the `commit` tab on the main Revision Graph window.

**Show current branch in Visual Studio**

Determines whether or not the currently checked out branch is displayed on the Git Extensions toolbar within Visual Studio.

**Auto scale user interface when high DPI is used**

Automatically resize controls and their contents according to the current system resolution of the display, measured in dots per inch (DPI).

**Truncate long filenames**

This setting affects the display of filenames in a component of a window e.g. in the Diff tab of the Revision Graph window. The options that can be selected are:

- `None` - no truncation occurs; a horizontal scroll bar is used to see the whole filename.
- `Compact` - no horizontal scroll bar. Filenames are truncated at both start and end to fit into the width of the display component.
- `Trimstart` - no horizontal scroll bar. Filenames are truncated at the start only.
- `FileNameOnly` - the path is always removed, leaving only the name of the file, even if there is space for the path.

**Sort by author date**

This setting causes commits to be sorted by author date (rather than commit date) in the revision grid. Sorting by author date may delay rendering of the revision graph.

**Author images****Show author's avatar column in the commit graph**

If checked, avatar images are downloaded for commit authors and shown in the revision grid.

**Show author's avatar in the commit info view**

If checked, avatar images are downloaded for commit authors and shown in the commit info view.

**Avatar provider**

The avatar provider setting determines the source from which avatar images are requested.

- `Default` - The default avatar provider loads a user defined avatar images, depending on the email address, from GitHub or Gravatar. If no user defined image could be found, a fallback images is used.
- `None` - If selected, no user-defined images are loaded and the fallback is evaluated immediately.
- `Custom` - An advanced mode that allows you to set one or more custom avatar provider services (e.g. Libravatar) by providing URL templates.

**URL Template Syntax** The URL template syntax consists of regular URLs to avatar images, that can be enriched with variables, which are substituted before evaluation. Those variables are encoded using curly brackets `{}` and can be used like this: `https://example.avatar.service/u/{email}/avatar.png`. If a request fails (http 400 and 500 errors) or does not provide a valid image, the next URL is used. More URLs can be specified by chaining them together with semicolons ("`;`") like so: `http://provider1.com/{sha1}.png;http://provider2.com/{sha1}.png`. If all custom URLs fail to provide an avatar image, the applications internal fallback mechanism will provide one for that user.

The variable names are case insensitive. If a variable is not found (for example because of typo or it does not exist), it is substituted with an empty string, so the resulting URL never contains the curly brackets.

The following variables are currently supported:

- `name` - The name of the commit author (git config `user.name`). Special characters are URL encoded.
- `email` - The email address of the commit author (git config `user.email`). Special characters are URL encoded.

- md5 - A lowercase hex representation of the MD5 hash of the normalized (all characters lowercase) email address (without URL encoding). This hash is compatible with Gravatar and thus compatible with a lot of similar services.
- sha1 - Like the md5 variable but with SHA1 as hash algorithm.
- sha256 - Like the md5 variable but with SHA256 as hash algorithm.
- imagesize - Represents the requested avatar size in pixels.

A complete working configuration might look something like this: `https://www.libravatar.org/avatar/{md5}?s={imageSize}&default=404;https://avatar.tobi.sh/{md5}?size={imageSize}`

#### **Fallback generated avatar style**

The configured fallback determines how authors without a user-defined avatar are presented. Besides `Author Initials` all other options are provided by Gravatar. Details about their fallback modes can be found here <https://en.gravatar.com/site/implement/images/> in the section “Default Image”. `Author Initials` are generated by the application internally and require no network connection to be displayed.

#### **Cache images (days)**

The number of days to elapse before the avatar image source is checked for any changes to an authors image.

#### **Clear image cache**

Clear the cached avatars.

### **Language**

#### **Language (restart required)**

Choose the language for the Git Extensions interface.

#### **Dictionary for spelling checker**

Choose the dictionary to use for the spelling checker in the Commit dialog.

### **3.1.2.1 Colors**

This page contains settings to define the colors used in the application.

#### **Revision graph**

##### **Multicolor branches**

Displays branch commits in different colors if checked. If unchecked, all branches are shown in the same color. This color can be selected.

##### **Draw alternate background**

Alternate background colour for revision rows.

##### **Draw non relatives graph gray**

Show commit history in gray for branches not related to the current branch.

##### **Draw non relatives text gray**

Show commit text in gray for branches not related to the current branch.

##### **Highlight authored revisions**

Highlight revisions committed by the same author as the selected revision.

##### **Color authored revisions**

Color to highlight authored revisions.

**Color tag**

Color to show tags in.

**Color branch**

Color to show branch names in.

**Color remote branch**

Color to show remote branch names in.

**Color other label**

Color to show other labels in.

**Difference View**

**Color removed line**

Highlight color for lines that have been removed.

**Color added line**

Highlight color for lines that have been added.

**Color removed line highlighting**

Highlight color for characters that have been removed in lines.

**Color added line highlighting**

Highlight color for characters that have been added in lines.

**Color section**

Highlight color for a section.

### 3.1.2.2 Fonts

**Fonts**

**Code font**

The font used for the display of file contents.

**Application font**

The font used on Git Extensions windows and dialogs.

**Commit font**

The font used for entering a commit message in the Commit dialog.

**Monospace font**

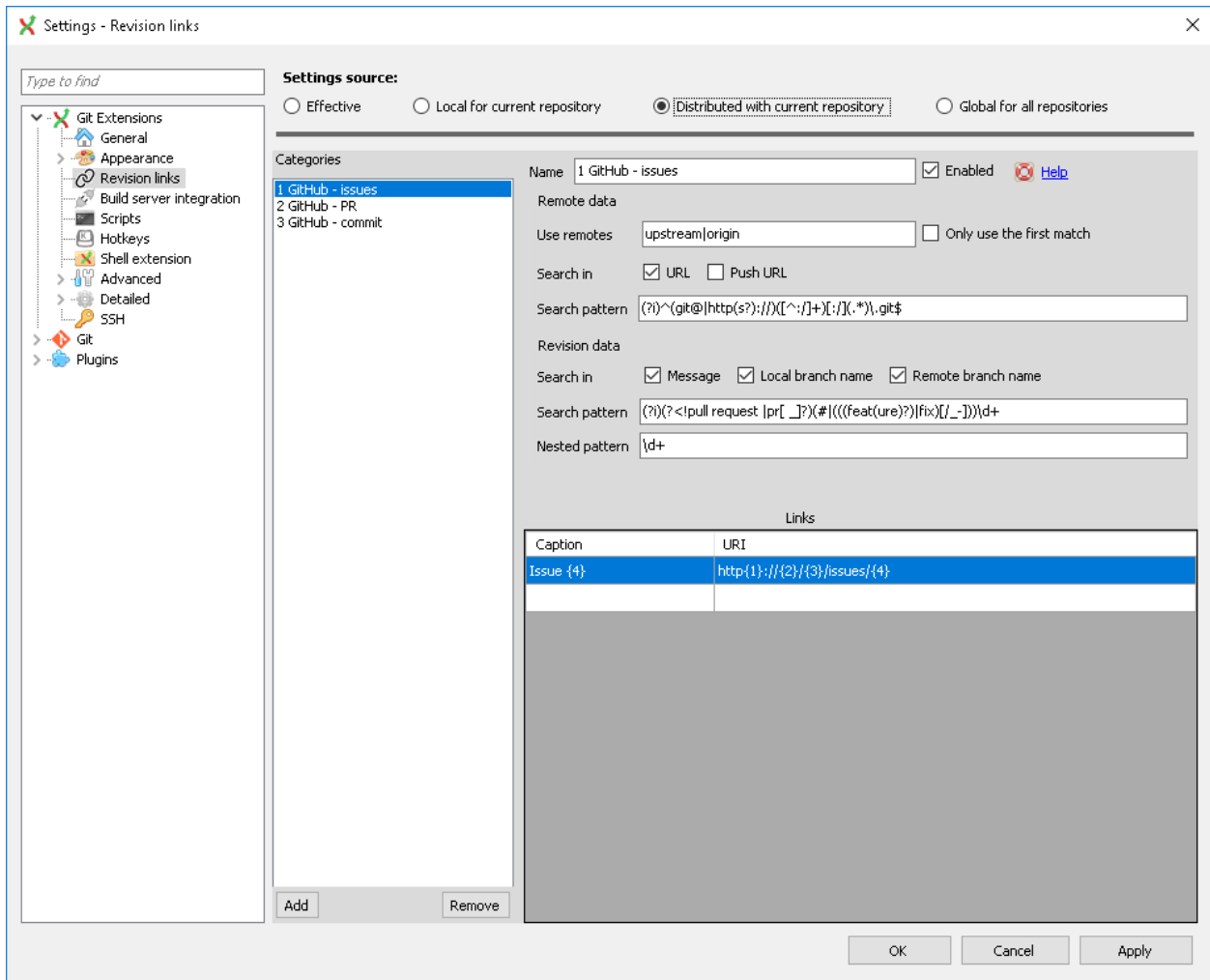
The font used for the commit id in the revision graph.

### 3.1.3 Revision Links

You can configure here how to convert parts of a revision data into clickable links. These links will be located under the commit message on the `Commit` tab in the `Related links` section.







**Categories**

Lists all the currently defined Categories. Click the **Add** button to add a new empty Category. The default name is 'new'. To remove a Category select it and click the **Remove** button.

**Name**

This is the Category name used to match the same categories defined on different levels of the Settings.

**Enabled**

Indicates whether the Category is enabled or not. Disabled categories are skipped while creating links.

**Remote data**

It is possible to use data from remote's URL to build a link. This way, links can be defined globally for all repositories sharing the same URL schema.

**Use remotes**

Regex to filter which remotes to use. Leave blank to create links not depending on remotes. If full names of remotes are given then matching remotes are sorted by its position in the given Regex.

**Only use the first match**

Check if you want to create links only for the first matching remote.

**Search in**

Define whether to search in URL, Push URL or both.

## Revision data

### Search in

Define which parts of the revision should be searched for matches.

**id** search-pattern

### Search pattern

Regular expression used for matching text in the chosen revision parts. Each matched fragment will be used to create a new link. More than one fragment can be used in a single link by using a capturing group. Matches from the Remote data group go before matches from the Revision data group. A capturing group value can be passed to a link by using zero-based indexed placeholders in a link format definition e.g. {0}.

### Nested pattern

`Nested pattern` can be used when only a part of the text matched by the *Search pattern* should be used to format a link. When the `Nested pattern` is empty, matches found by the *Search pattern* are used to create links.

### Links: Caption/URI

List of links to be created from a single match. Each link consists of the `Caption` to be displayed and the `URI` to be opened when the link is clicked on. In addition to the standard zero-based indexed placeholders, the `%COMMIT_HASH%` placeholder can be used to put the commit's hash into the link. For example:  
`https://github.com/gitextensions/gitextensions/commit/%COMMIT_HASH%`

## 3.1.4 Build server integration

This page allows you to configure the integration with build servers. This allows the build status of each commit to be displayed directly in the revision log, as well as providing a tab for direct access to the Build Server build report for the selected commit.

### Enable build server integration

Check to globally enable/disable the integration functionality.

### Build server type

Select an integration target.

### AppVeyor

#### Account name

AppVeyor account name. You don't have to enter it if the projects you want to query for build status are public.

#### API token

AppVeyor API token. Required if the *Account name* is entered. See <https://ci.appveyor.com/api-token>

#### Project(s) name(s)

Projects names separated with '|', e.g. `gitextensions/gitextensions|jbiabobr/gitextensions`

#### Display tests results in build status summary for every build result

Include tests results in the build status summary for every build result.

### Azure DevOps and Team Foundation Server (since TFS2015)

#### Project URL

Enter the URL of the server (and port, if applicable).

#### Build definition name

Limit the builds if desired.

**Rest API token**

Read token for the build server.

**Jenkins**

**Jenkins server URL**

Enter the URL of the server (and port, if applicable).

**Project name**

Enter the name of the project which tracks this repository in Jenkins. Separate project names with “|”. Multi-branch pipeline projects are supported by adding “?m” to the project name.

**Ignore build for branch**

The plugin will normally display the last build for a certain commit. If Jenkins starts several builds for one commit, it is possible to ignore the non interesting builds if all builds are not interesting.

**TeamCity**

**TeamCity server URL**

Enter the URL of the server (and port, if applicable).

**Project name**

Enter the name of the project which tracks this repository in TeamCity. Multiple project names can be entered separated by the | character.

**Build Id Filter**

Enter a regexp filter for which build results you want to retrieve in the case that your build project creates multiple builds. For example, if your project includes both devBuild and docBuild you may wish to apply a filter of “devBuild” to retrieve the results from only the program build.

**Team Foundation Server**

For TFS prior to 2015.

**Tfs server (Name or URL)**

Enter the URL of the server (and port, if applicable).

**Team collection name**

**Project name**

Enter the name of the project which tracks this repository in Tfs.

**Build definition name**

Use first found if left empty.

### 3.1.5 Scripts

This page allows you to configure specific commands to run before/after Git actions or to add a new command to the User Menu. The top half of the page summarises all of the scripts currently defined. If a script is selected from the summary, the bottom half of the page will allow modifications to the script definition.

A hotkey can also be assigned to execute a specific script. See *Hotkeys*.

**Add**

Adds a new script. Complete the details in the bottom half of the screen.

**Remove**

Removes a script.

**Up/Down Arrows**

Changes order of scripts.

**Name**

The name of the script.

**Enabled**

If checked, the script is active and will be performed at the appropriate time (as determined by the On Event setting).

**Ask for confirmation**

If checked, then a popup window is displayed just before the script is run to confirm whether or not the script is to be run. Note that this popup is *not* displayed when the script is added as a command to the User Menu (On Event setting is ShowInUserMenuBar).

**Run in background**

If checked, the script will run in the background and Git Extensions will return to your control without waiting for the script to finish.

**Add to revision grid context menu**

If checked, the script is added to the context menu that is displayed when right-clicking on a line in the Revision Graph page.

**Is PowerShell**

If checked, the command is started through a powershell.exe process. If the *Run in background* is checked, the powershell console is closed after finishing. If not, the powershell console is left for the user to close it manually.

**Command**

Enter the command to be run. This can be any command that your system can run e.g. an executable program, a .bat script, a Python command, etc. Use the `Browse` button to find the command to run.

There are some special prefixes which change the way the script is executed:

- `plugin:<plugin-name>`: Where `<plugin-name>` is the name of a *plugin* (refer *Plugins*). If a plugin with that name is found, it is run.
- `navigateTo:<script-path>`: Where `<script-path>` is the path to a file containing the script to run. That script is expected to return a commit hash as the first line of its output. The UI will navigate to that commit once the script completes.

**Arguments**

Enter any arguments to be passed to the command that is run. The `Help` button displays items that will be resolved by Git Extensions before executing the command e.g. `{cBranch}` will resolve to the currently checked out branch, `{UserInput}` will display a popup where you can enter data to be passed to the command when it is run.

**On Event**

Select when this command will be executed, either before/after certain Git commands, or displayed on the User Menu bar. Since the git pull command includes a fetch, before/after fetch events are triggered on pure fetches as well as on pulls. For the pull command the script execution order is BeforePull - BeforeFetch - git pull - AfterFetch - AfterPull.

**Icon**

Select an icon to be displayed in a menu item when the script is marked to be shown in the user menu bar.

### 3.1.6 Hotkeys

This page allows you to define keyboard shortcuts to actions when specific pages of Git Extensions are displayed. The HotKeyable Items identifies a page within Git Extensions. Selecting a Hotkeyable Item displays the list of commands on that page that can have a hotkey associated with them.

The Hotkeyable Items consist of the following pages

- 1) **Commit**: the page displayed when a Commit is requested via the `Commit User Menu` button or the `Commands/Commit` menu option.
- 2) **Browse**: the Revision Graph page (the page displayed after a repository is selected from the dashboard (Start Page)).
- 3) **RevisionGrid**: the list of commits in Browse and other forms.
- 4) **FileViewer**: the page displayed when viewing the contents of a file.
- 5) **FormMergeConflicts**: the page displayed when merge conflicts are detected that need correcting.
- 6) **BrowseDiff**: Diff tab in Browse.
- 7) **RevisionFileTree**: The FileTree tab in Browse.
- 8) **Scripts**: shows scripts defined in Git Extensions and allows shortcuts to be assigned. Refer *Scripts*.

#### **Hotkey**

After selecting a Hotkeyable Item and the Command, the current keyboard shortcut associated with the command is displayed here. To alter this shortcut, click in the box where the current hotkey is shown and press the new keyboard combination.

#### **Apply**

Click to apply the new keyboard combination to the currently selected Command.

#### **Clear**

Sets the keyboard shortcut for the currently selected Command to 'None'.

#### **Reset all Hotkeys to defaults**

Resets all keyboard shortcuts to the defaults (i.e. the values when Git Extensions was first installed).

### **3.1.7 Shell Extension**

When installed, Git Extensions adds items to the context menu when a file/folder is right-clicked within Windows Explorer. One of these items is `Git Extensions` from which a further (cascaded) menu can be opened. This settings page determines which items will appear on that cascaded menu and which will appear in the main context menu. Items that are checked will appear in the cascaded menu.

To the right side of the list of check boxes is a preview that shows you how the Git Extensions menu items will be arranged with your current choices.

By default, what is displayed in the context menu also depends on what item is right-clicked in Windows Explorer; a file or a folder (and whether the folder is a Git repository or not). If you want Git Extensions to always include all of its context menu items, check the box `Always show all commands`.

### **3.1.8 Advanced**

This page allows advanced settings to be modified. Refer *Confirmations*.

#### **Checkout**

##### **Always show checkout dialog**

Always show the Checkout Branch dialog when swapping branches. This dialog is normally only shown when uncommitted changes exist on the current branch

##### **Use last chosen "local changes" action as default action.**

This setting works in conjunction with the 'Git Extensions/Check for uncommitted changes in checkout branch dialog' setting. If the 'Check for uncommitted changes' setting is checked, then the Checkout

Branch dialog is shown only if this setting is unchecked. If this setting is checked, then no dialog is shown and the last chosen action is used.

## General

### **Don't show help images**

In the Pull, Merge and Rebase dialogs, images are displayed by default to explain what happens with the branches and their commits and the meaning of LOCAL, BASE and REMOTE (for resolving merge conflicts) in different merge or rebase scenarios. If checked, these Help images will not be displayed.

### **Always show advanced options**

In the Push, Merge and Rebase dialogs, advanced options are hidden by default and shown only after you click a link or checkbox. If this setting is checked then these options are always shown on those dialogs.

### **Use Console Emulator for console output in command dialogs**

Using Console Emulator for console output in command dialogs may be useful the running command requires an user input, e.g. push, pull using ssh, confirming gc.

### **Auto normalise branch name**

Controls whether branch name should be automatically normalised as per git branch naming rules. If enabled, any illegal symbols will be replaced with the replacement symbol of your choice.

## Commit

### **Push forced with lease when Commit & Push action is performed with Amend option checked**

In the Commit dialog, users can commit and push changes with one click. However, if changes are meant to amend an already pushed commit, a standard push action will be rejected by the remote server. If this option is enabled, a push action with `--force-with-lease` switch will be performed instead. The `--force-with-lease` switch will be added only when the Amend option is checked.

## Updates

### **Check for updates weekly**

Check for newer version every week.

### **Check for release candidate versions**

Include release candidate versions when checking for a newer version.

### 3.1.8.1 Confirmations

This page allows you to turn off certain confirmation popup windows.

#### **Don't ask to confirm to**

##### **Amend last commit**

If checked, do not display the popup warning about the rewriting of history when you have elected to amend the last committed change.

##### **Commit when no branch is currently checked out**

When committing changes and there is no branch currently being checked out, then GitExtensions warns you and proposes to checkout or create a branch. Enable this option to continue working with no warning.

##### **Apply stashed changes after successful pull**

In the Pull dialog, if `Auto stash` is checked, then any changes will be stashed before the pull is performed. Any stashed changes are then re-applied after the pull is complete. If this setting is checked, the stashed changes are applied with no confirmation popup.

**Apply stashed changes after successful checkout**

In the Checkout Branch dialog, if `Stash` is checked, then any changes will be stashed before the branch is checked out. If this setting is checked, then the stashed changes will be automatically re-applied after successful checkout of the branch with no confirmation popup.

**Drop stash**

Popup when dropping a stash.

**Add a tracking reference for newly pushed branch**

When you push a local branch to a remote and it doesn't have a tracking reference, you are asked to confirm whether you want to add such a reference. If this setting is checked, a tracking reference will always be added if it does not exist.

**Push a new branch for the remote**

When pushing a new branch that does not exist on the remote repository, a confirmation popup will normally be displayed. If this setting is checked, then the new branch will be pushed with no confirmation popup.

**Update submodules on checkout**

When you check out a branch from a repository that has submodules, you will be asked to update the submodules. If this setting is checked, the submodules will be updated without asking.

**Resolve conflicts**

If enabled, then when conflicts are detected GitExtensions will start the Resolve conflicts dialog automatically without any prompt.

**Commit changes after conflicts have been resolved**

Enable this option to start the Commit dialog automatically after all conflicts have been resolved.

**Confirm for the second time to abort a merge**

When aborting a merge, rebase or other operation that caused conflicts to be resolved, an user is warned about the consequences of aborting and asked if he/she wants to continue. If the user chooses to continue the aborting operation, then he/she is asked for the second time if he/she is sure that he/she wants to abort. Enable this option to skip this second confirmation.

**Rebase on top of selected commit**

Rebase context menu command popup in revision graph.

**Undo last commit**

Browse Command popup.

**Fetch and prune all**

Browse fetch/prune popup.

**Switch Worktree**

Switch worktree popup.

### 3.1.9 Detailed

This page allows detailed settings to be modified.

**Push window****Get remote branches directly from the remote**

Git caches locally remote data. This data is updated each time a fetch operation is performed. For a better performance GitExtensions uses the locally cached remote data to fill out controls on the Push dialog. Enable this option if you want GitExtensions to use remote data received directly from the remote server.



## Merge window

### Add log messages

If enabled, then in addition to branch names, git will populate the log message with one-line descriptions from at most the given number actual commits that are being merged. See [https://git-scm.com/docs/git-merge#Documentation/git-merge.txt—log1n1ngt](https://git-scm.com/docs/git-merge#Documentation/git-merge.txt---log1n1ngt)

### 3.1.9.1 Browse repository window

#### Console emulator

##### Show the Console tab

Show the Console tab in the *Browse Repository* window.

##### Console style

Choose one of the predefined ConEmu schemes. See <http://conemu.github.io/en/SettingsColors.html>.

##### Shell to run

Choose one of the predefined terminals.

##### Font size

Console font size.

##### Show GPG information

Show tab for GPG information if available.

### 3.1.9.2 Commit dialog

This page contains settings for the Git Extensions *Commit* dialog. Note that the dialog itself has further options.

#### Behaviour

##### Provide auto-completion in commit dialog

Enables auto-completion in commit dialog message box. Auto-completion words are taken from the changed files shown by the commit dialog. For each file type there can be configured a regular expression that decides which words should be considered as candidates for auto-completion. The default regular expressions included with Git Extensions can be found here: <https://github.com/gitextensions/gitextensions/blob/master/GitExtensions/AutoCompleteRegexes.txt> You can override the default regular expressions by creating an `AutoCompleteRegexes.txt` file in the Git Extensions installation directory.

##### Show errors when staging files

If an error occurs when files are staged (in the Commit dialog), then the process dialog showing the results of the git command is shown if this setting is checked.

##### Ensure the second line of commit message is empty

Enforces the second line of a commit message to be blank.

##### Compose commit messages in Commit dialog

If this is unchecked, then commit messages cannot be entered in the commit dialog. When the Commit button is clicked, a new editor window is opened where the commit message can be entered.

##### Number of previous messages in commit dialog

The number of commit messages, from the top of the current branch, that will be made available from the Commit message combo box on the Commit dialog.

**Remember 'Amend commit' checkbox on commit form close**

Remembers the state of the 'Amend commit' checkbox when the 'Commit dialog' is being closed. The remembered state will be restored on the next 'Commit dialog' creation. The 'Amend commit' checkbox is being unchecked after each commit. So, when the 'Commit dialog' is being closed automatically after committing changes, the 'Amend commit' checkbox is going to be unchecked first and its state will be saved after that. Therefore the checked state is remembered only if the 'Commit dialog' is being closed by an user without committing changes.

**Show additional buttons in commit button area**

Tick the boxes in this sub-group for any of the additional buttons that you wish to have available below the commit button. These buttons are considered additional to basic functionality and have consequences if you should click them accidentally, including resetting unrecorded work.

### 3.1.9.3 Diff Viewer

**Remember the 'Ignore whitespaces' preference**

Remember in the GitExtensions settings the latest chosen value of the 'Ignore whitespaces' preference. Use the remembered value the next time GitExtensions is opened.

**Remember the 'Show nonprinting characters' preference**

Remember in the GitExtensions settings the latest chosen value of the 'Show nonprinting characters' preference. Use the remembered value the next time GitExtensions is opened.

**Remember the 'Show entire file' preference**

Remember in the GitExtensions settings the latest chosen value of the 'Show entire file' preference. Use the remembered value the next time GitExtensions is opened.

**Remember the 'Number of context lines' preference**

Remember in the GitExtensions settings the latest chosen value of the 'Number of context lines' preference. Use the remembered value the next time GitExtensions is opened.

**Omit uninteresting changes from combined diff**

Includes git `-cc` switch when generating a diff. See <https://git-scm.com/docs/git-diff-tree#Documentation/git-diff-tree.txt-cc>

**Open Submodule Diff in separate window**

If enabled then double clicking on a submodule in the Diff file list opens a new instance of GitExtensions with the submodule as the selected repository. If disabled, the File history window is opened for the double clicked submodule.

**Show file differences for all parents in browse dialog**

Enable this option to see diff against each of the revision parents, combined diff including.

**Vertical ruler position**

Position for ruler in TextEditor controls. Set to 0 to disable. (This should be moved to the TextEditor context menu.)

## 3.1.10 SSH

This page allows you to configure the SSH client you want Git to use. Git Extensions is optimized for PuTTY. Git Extensions will show command line dialogs if you do not use PuTTY and user input is required (unless you have configured SSH to use authentication with key instead of password). Git Extensions can load SSH keys for PuTTY when needed.

**Specify which ssh client to use**

**PuTTY**

Use PuTTY as SSH client.

**OpenSSH**

Use OpenSSH as SSH client.

**Other ssh client**

Use another SSH client. Enter the path to the SSH client you wish to use.

**Configure PuTTY****Path to plink.exe**

Enter the path to the plink.exe executable.

**Path to puttygen**

Enter the path to the puttygen.exe executable.

**Path to pageant**

Enter the path to the pageant.exe executable.

**Automatically start authentication**

If an SSH key has been configured, then when accessing a remote repository the key will automatically be used by the SSH client if this is checked.

## 3.2 Git

The settings that are used by Git are stored in the configuration files of Git. The global settings are stored in the file called `.gitconfig` in the user directory. The local settings are stored in the `.git\config` file of the repository.

### 3.2.1 Paths

This page contains the settings needed to access git repositories. The repositories will be accessed using external tools. For Windows usually “Git for Windows” is used. Git Extensions will try to configure these settings automatically.

**Git****Command used to run git (git.cmd or git.exe)**

Needed for Git Extensions to run Git commands. Set the full command used to run git (“Git for Windows”). Use the `Browse` button to find the executable on your file system. (Cygwin Git may work but is not officially supported.)

**Path to Linux tools (sh).**

A few Linux tools are used by Git Extensions. When Git for Windows is installed, these tools are located in the bin directory of Git for Windows. Use the `Browse` button to find the directory on your file system. Leave empty when it is in the path.

**Environment****Change HOME**

This button opens a dialog where the HOME directory can be changed.

The global configuration file used by git will be put in the HOME directory. On some systems the home directory is not set or is pointed to a network drive. Git Extensions will try to detect the optimal setting for your environment. When there is already a global git configuration file, this location will be used. If

you need to relocate the home directory for git, click the `Change HOME` button to change this setting. Otherwise leave this setting as the default.

## 3.2.2 Config

This page contains some of the settings of Git that are used by and therefore can be changed from within Git Extensions.

If you change a Git setting from the Git command line using `git config` then the same change in setting can be seen inside Git Extensions. If you change a Git setting from inside Git Extensions then that change can be seen using `git config --get`.

Git configuration can be global or local configuration. Global configuration applies to all repositories. Local configuration overrides the global configuration for the current repository.

### User name

User name shown in commits and patches.

### User email

User email shown in commits and patches.

### Editor

Editor that `git.exe` opens (e.g. for editing commit message). This is not used by Git Extensions, only when you call `git.exe` from the command line. By default Git will use the built in editor.

### Mergetool

Merge tool used to solve merge conflicts. Git Extensions will search for common merge tools on your system.

### Path to mergetool

Path to merge tool. Git Extensions will search for common merge tools on your system.

### Mergetool command

Command that Git uses to start the merge tool. Git Extensions will try to set this automatically when a merge tool is chosen. This setting can be left empty when Git supports the mergetool (e.g. `kdiff3`).

### Keep backup (.orig) after merge

Check to save the state of the original file before modifying to solve merge conflicts. Refer to Git configuration setting `mergetool.keepBackup``.

### Difftool

Diff tool that is used to show differences between source files. Git Extensions will search for common diff tools on your system.

### Path to difftool

The path to the diff tool. Git Extensions will search for common diff tools on your system.

### DiffTool command

Command that Git uses to start the diff tool. This setting should only be filled in when Git doesn't support the diff tool.

### Path to commit template

A path to a file whose contents are used to pre-populate the commit message in the commit dialog.

### Line endings

#### Checkout/commit radio buttons

Choose how git should handle line endings when checking out and checking in files. Refer to <https://help.github.com/articles/dealing-with-line-endings/#platform-all>

#### **Files content encoding**

The default encoding for files content.

### **3.2.3 Advanced**

Various settings for Git.

## **3.3 Plugins**

Plugins provide extra functionality for Git Extensions. Please refer to *Plugins*.

---

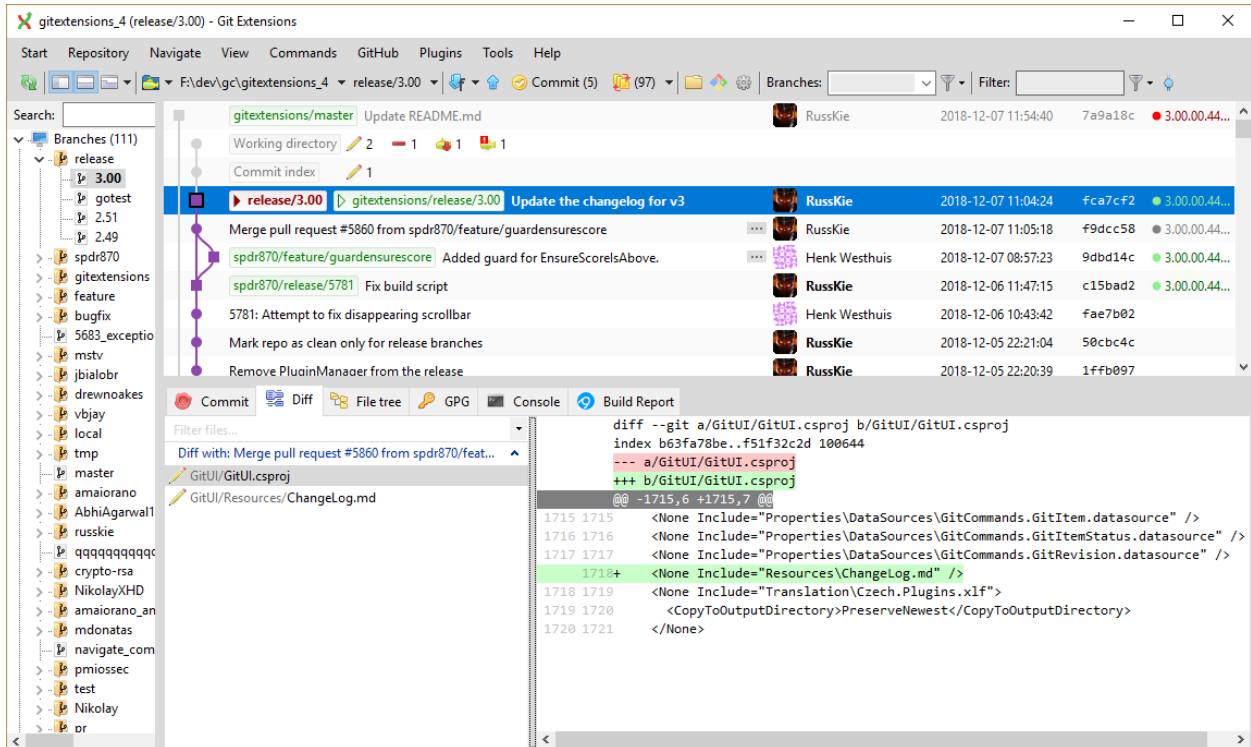
## Browse Repository

---

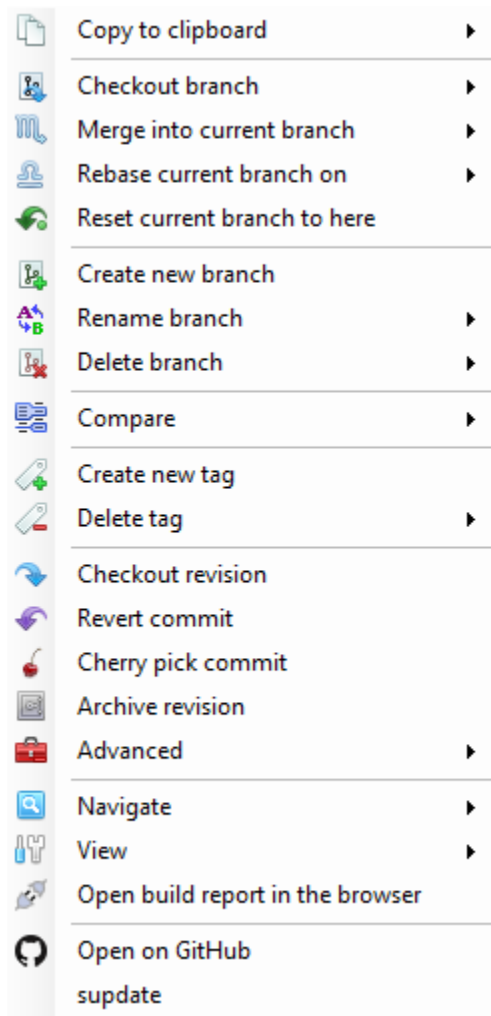
You can browse a repository by starting Git Extensions and select the repository to open. The main window contains the revision graph (commit log). You could also open the 'Browse' window from the shell extensions and from Visual Studio.

### 4.1 View revision graph

The full commit history can be browsed. There is a graph that shows branches and merges. You can show the difference between any two revisions by selecting them using ctrl-click.



The context menu for a commit can both execute Git commands and change the appearance for the form.



## 4.2 Search or filter the commit history

You can find text in the commit messages or jump to a specific commit in the current commit history shown in Git Extensions. You can also filter the commit history so that fewer commits are shown.

### 4.2.1 Quick search in history

You can find a commit in the commit history that is shown in Git Extensions by searching for text in the commit message, branch label or tag. This is a quick search function. Simply click into the commit history to give that pane focus and start typing. Git Extensions will show your search term in the top left corner and will immediately jump to the next commit with matching text. You can search for the next or previous commit with matching text using `Alt-Down Arrow` or `Alt-Up Arrow`.

In Settings, `Git Extensions` you can change the timeout for typing the text for the quick search.

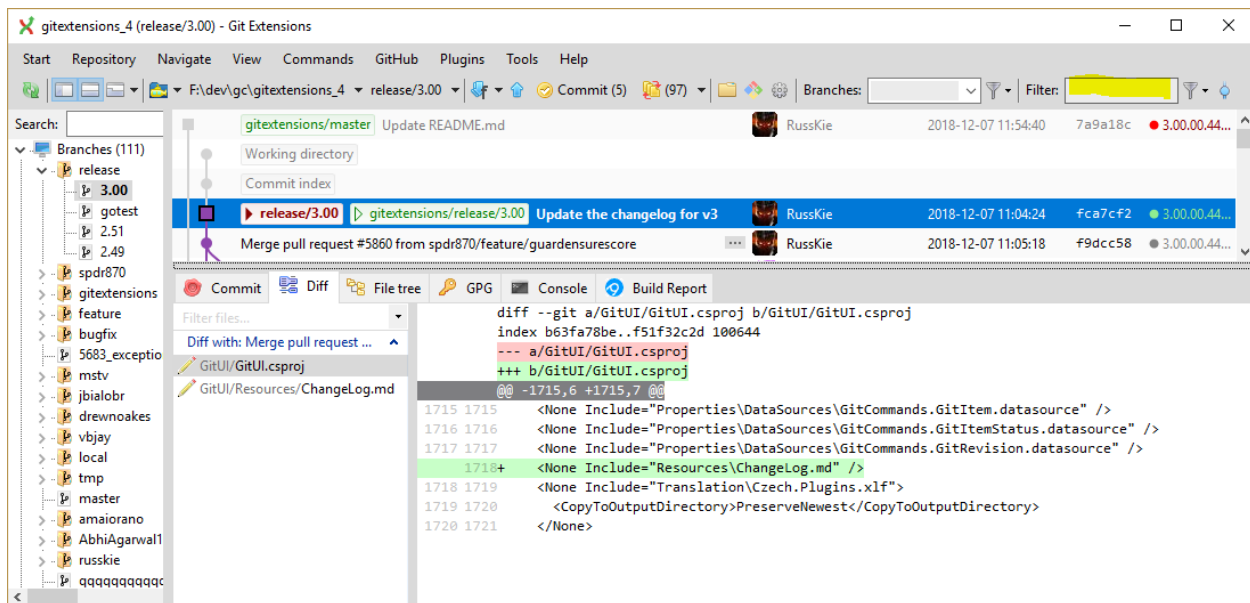


## 4.2.2 Go to a specific commit

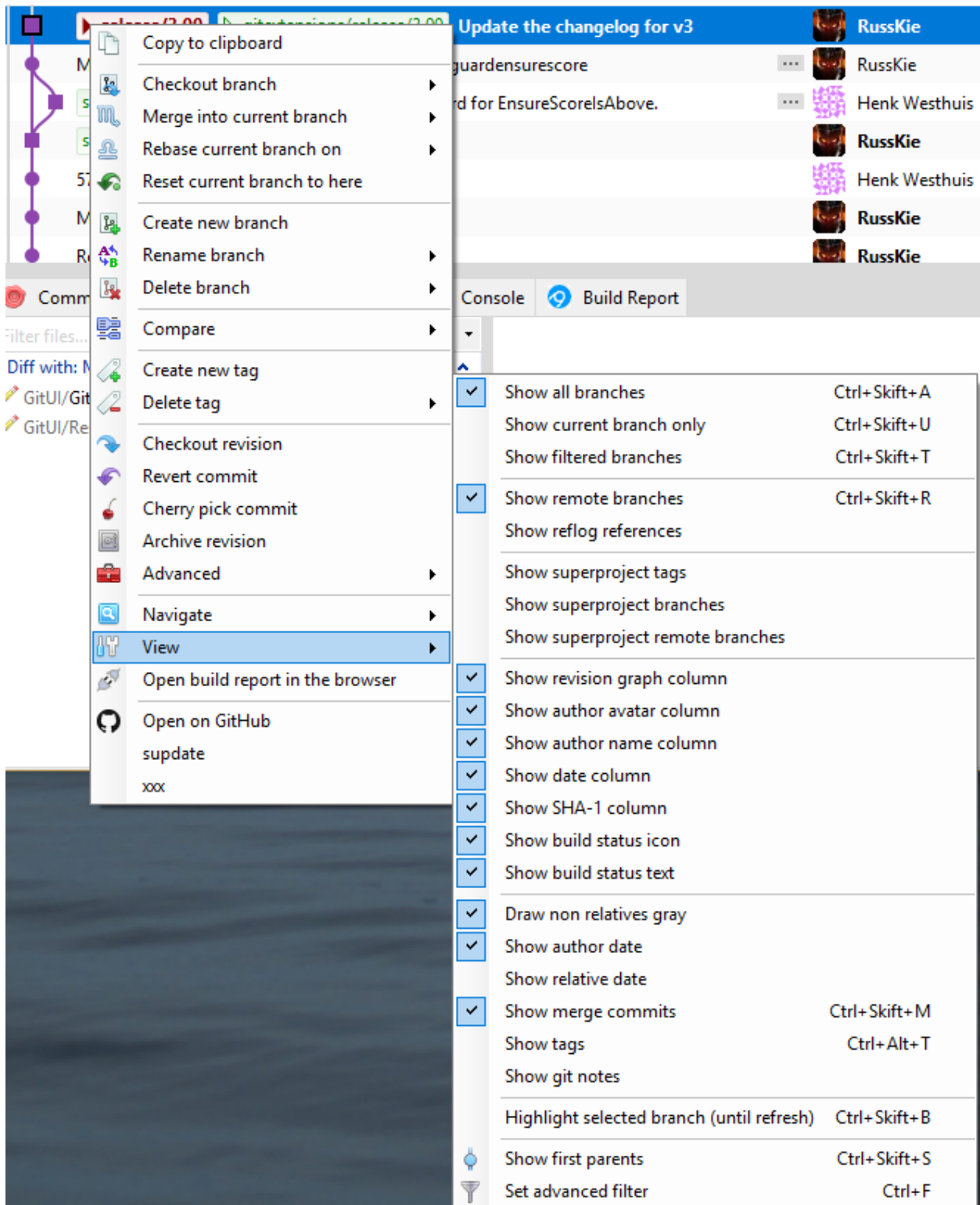
You can jump to a particular commit in the commit history if you know the SHA, tag or branch. In fact you can use any expression valid for git-rev-parse. Select **Navigate, Go to commit** or press **Ctrl-Shift-G** to open the **Go to commit** window. Enter an SHA or other term to be passed to git-rev-parse into the box at the top and click **Go**, or select a branch or tag from one of the two combo boxes below.

## 4.2.3 Filter history

The history can be filtered using regular expressions and basic filter terms. Filtering will reduce the number of commits that are shown in the Git Extensions commit history. The quick filter in the toolbar filters by the commit message, the author and/or the committer.



In the context menu of the commit log you can open the advanced filter dialog. The advanced filter dialog allows you to filter for more specific commits. To remove the filter either remove the filter in the toolbar and press enter or remove the filter in the advanced filter dialog.



**Filter** [X]

Since  den 9 december 2018 [calendar icon]

Until  den 9 december 2018 [calendar icon]

Author  [text box]

Committer  [text box]

Message  [text box]

Ignore case

Limit  100000 [spinners]

File filter  [text box]

Branches  [text box]

Show current branch only

Simplify by decoration

OK

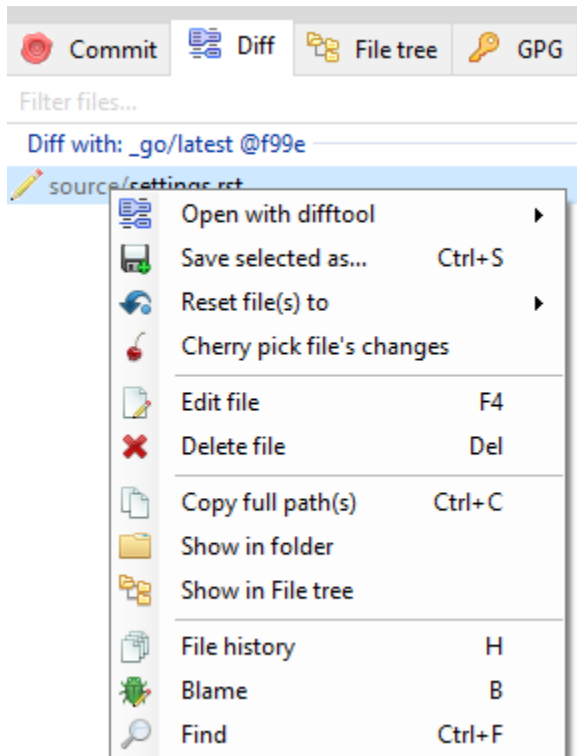
## CHAPTER 5

---

### File history

---

To display the single file history, right click on a file name in the *Browse Repository* File tree or in the Diff tab and select File history or Blame. The single file history viewer shows all revisions of a single file. (This is available for submodules too, but the information is mostly not interesting.)



## 5.1 Commit

The `Commit` tab contains the information about the commit, including the other files in the commit.

The screenshot shows the 'Commit' tab in Git Extensions. At the top, there's a 'File History' section with a tree view showing the commit history. The current commit is highlighted in blue. Below this, there's a 'Commit' tab with a profile picture of the author, commit details (Author, Date, Committer, Commit hash, Parents), and the commit message: 'Merge pull request #5641 from NikolayXHD/fix\_commit\_info\_scroll'. The commit message also includes 'Fix commitInfo scroll on mouse wheel' and 'Notes:'. Below the commit message, there are 'Related links' and 'Contained in branches'. The bottom part of the screenshot shows a 'Diff' view with a list of files on the left and a code diff on the right. The diff shows changes to 'a/GitUI/CommandsDialogs/FormBrowse.Designer.cs' and 'a/GitUI/CommandsDialogs/FormBrowse.cs'. The code diff shows a change in the 'RevisionInfoHeader.cs' file, where a line is added (marked with '+') and a line is removed (marked with '-').

## 5.2 Diff

You can view the difference report from the commit in the `Diff` tab.

**Note:** Added lines are marked with a +, removed lines are marked with a -.

File History - GitUI/CommandsDialogs/FormBrowse.cs - F:\dev\gc\gitextensi

Branches:  Filter:

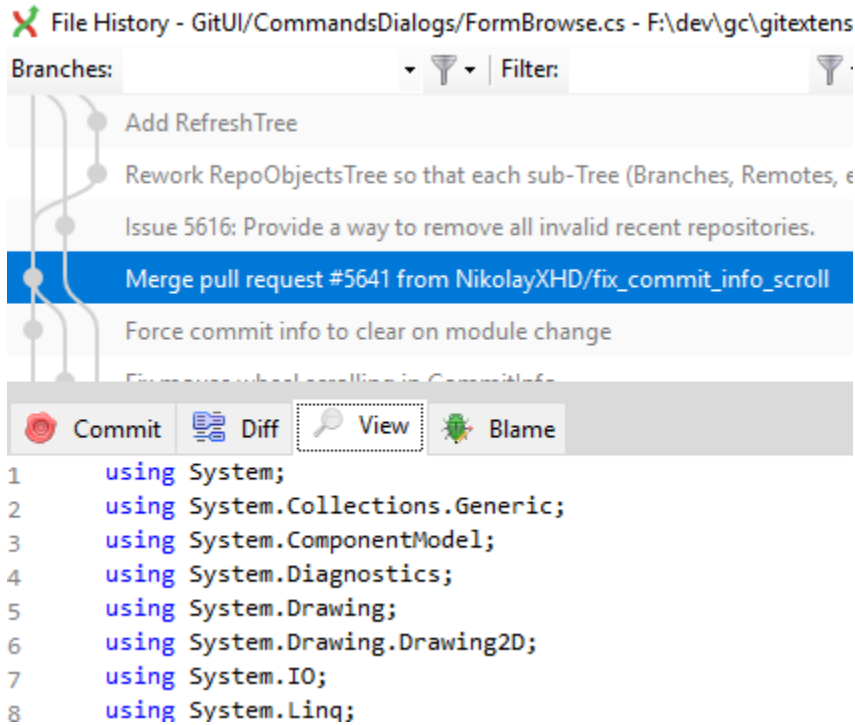
- Add RefreshTree
- Rework RepoObjectsTree so that each sub-Tree (Branches, Remotes, et
- Issue 5616: Provide a way to remove all invalid recent repositories.
- Merge pull request #5641 from NikolayXHD/fix\_commit\_info\_scroll
- Force commit info to clear on module change
- Fix mouse wheel scrolling in CommitInfo

Commit Diff View Blame

```
diff --git a/GitUI/CommandsDialogs/FormBrowse.cs b/GitUI/CommandsDialogs/FormBrowse.cs
index 7306fece0..759d88737 100644
--- a/GitUI/CommandsDialogs/FormBrowse.cs
+++ b/GitUI/CommandsDialogs/FormBrowse.cs
@@ -1214,11 +1214,6 @@ private void FillCommitInfo(
1214 1214
1215 1215         var children = RevisionGrid.GetRevisionInfo();
1216 1216         RevisionInfo.SetRevisionWithChildren(revisionInfo, children);
1217 -         if (RevisionInfo.Parent is Panel parent)
1218 -         {
1219 -             parent.AutoScroll = true;
1220 -             parent.AutoScrollMinSize = RevisionInfo.Parent.AutoScrollMinSize;
1221 -         }
1222 1217     }
1223 1218
1224 1219     private async Task FillGpgInfoAsync()
```

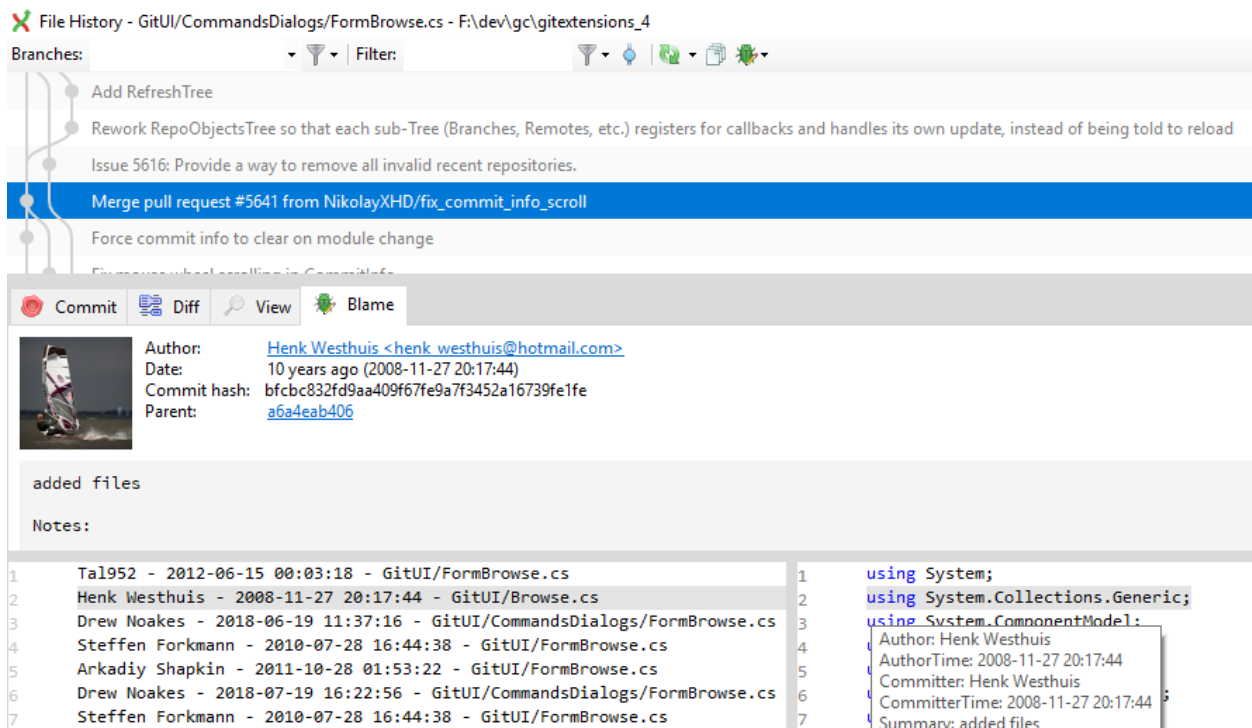
## 5.3 View

You can view the content of the file in after each commit in the `View` tab.



## 5.4 Blame

There is a blame function in the file history browser. The commit for the selected line is displayed.



Double clicking on a code line shows the full commit introducing the change.

A commit is a set of changes with some extra information. Every commit contains the following information:

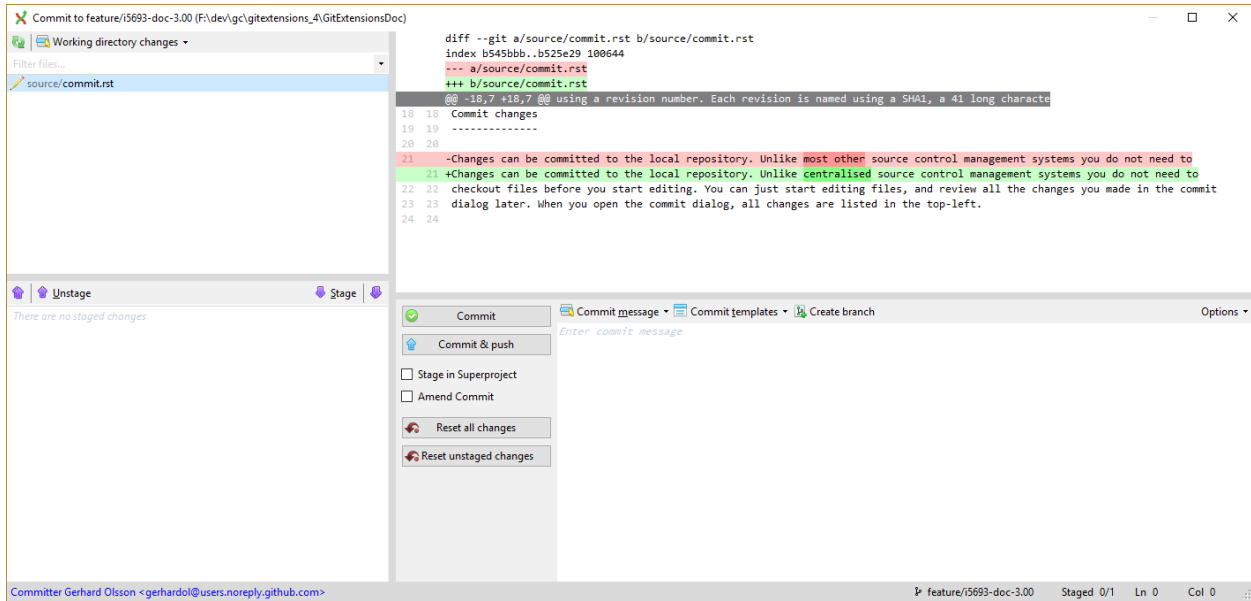
- Changes
- Committer name and email
- Commit date
- Commit message
- Cryptographically strong SHA1 hash

Each commit creates a new revision of the source. Revisions are not tracked per file; each change creates a new revision of the complete source. Unlike most traditional source control management systems, revisions are not named using a revision number. Each revision is named using a SHA1, a 41 long characters cryptographically strong hash.

## 6.1 Commit changes

Changes can be committed to the local repository. Unlike centralised source control management systems you do not need to checkout files before you start editing. You can just start editing files, and review all the changes you made in the commit dialog later. When you open the commit dialog, all changes are listed in the top-left.



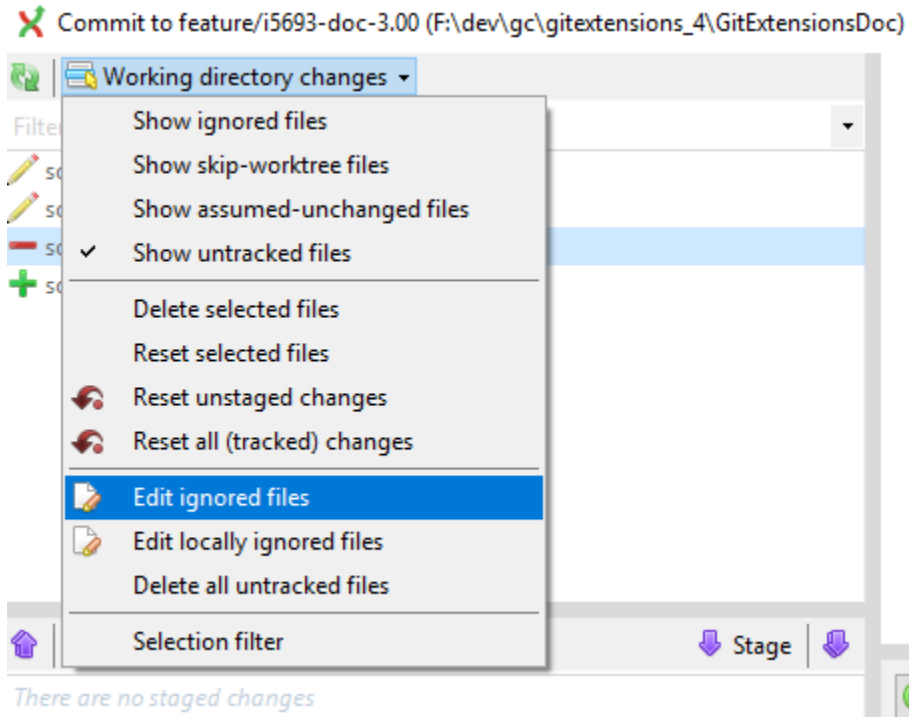


There are three kinds of changes:

Un-tracked	This file is not yet tracked by Git. This is probably a new file, or a file that has not been committed to Git before.
Modified	This file is modified since the last commit.
Deleted	This file has been deleted.

When you rename or move a file Git will notice that this file has been moved and notice in index pane (not in working directory).

During your initial commit there are probably lots of files you do not want to be tracked. You can ignore these files by not staging them, but they will show every time. You can instead add them to the `.gitignore` file of your repository. Files that are in the `.gitignore` file will not show up in the commit dialog again. You can open the `.gitignore` editor from the menu Working dir changes by selecting Edit ignored files.



Making a commit is a two step procedure:

- Adding to index (staging) the changes to be committed, which saves a snapshot of the changes into the Git “index”.
- Committing those staged changes, which records the staged changes and other information into the repository.

You do not have to commit immediately after staging changes. You can close the commit dialog, make further changes to the files in the working dir, then re-open the commit dialog to stage further changes and commit. Changes that you have staged previously will still be staged when you re-open the dialog.

### 6.1.1 Staging changes

The changes that you have made to your working directory are not automatically included in a commit. You must choose which of the changed files, or individual changes from within those files, will be included in the commit by “staging” the changes in Git Extensions. Staging changes in Git Extensions is the same as using `git add` on the Git command line.

You can stage the changes you want to commit by selecting the files in the top-left or “Unstaged changes” pane and pressing the `Stage` button or pressing the `[S]` key. The file entries will move to the lower left or “Staged changes” pane. You need to stage deleted files because you stage the change and not the file. If you have staged changes from a file and you wish to exclude those changes from the commit, select the entry in the staged changes pane and press the `Unstage` button or press the `[U]` key.

If the file that is selected in either the unstaged or staged changes pane is text format, Git Extensions will show a Git “diff” view in the right side pane of the window.

### 6.1.2 Staging selected lines

You do not have to commit all of the changes in a text format file in one commit. You can select and stage individual lines from within a file such that only the chosen lines will be included in your next commit; the remaining changes in

the file will appear as unstaged changes for the next commit.

In the diff view on the right, select the line or lines that you want to stage then right-click and choose `Stage selected line(s)` or press the `[S]` key. The file will now appear in both the staged changes and unstaged changes panes on the left since now there are both staged and unstaged changes in the same file. The change that was selected will disappear from the diff view on the right because the diff view is showing only the unstaged changes.

To see the line changes that have been staged select the entry for the file in the staged changes pane. To unstage selected changed lines from a file, select that file in the staged changes pane, then select the line or lines in the diff view, right -click, and choose `Unstage selected line(s)` or press the `[U]` key.

---

**Note:** If you select an entire line including the end-of-line character then staging or unstaging that line will include both the selected line and the next line. To select a single line to stage or unstage you may simply click onto the line without selecting any particular characters.

---

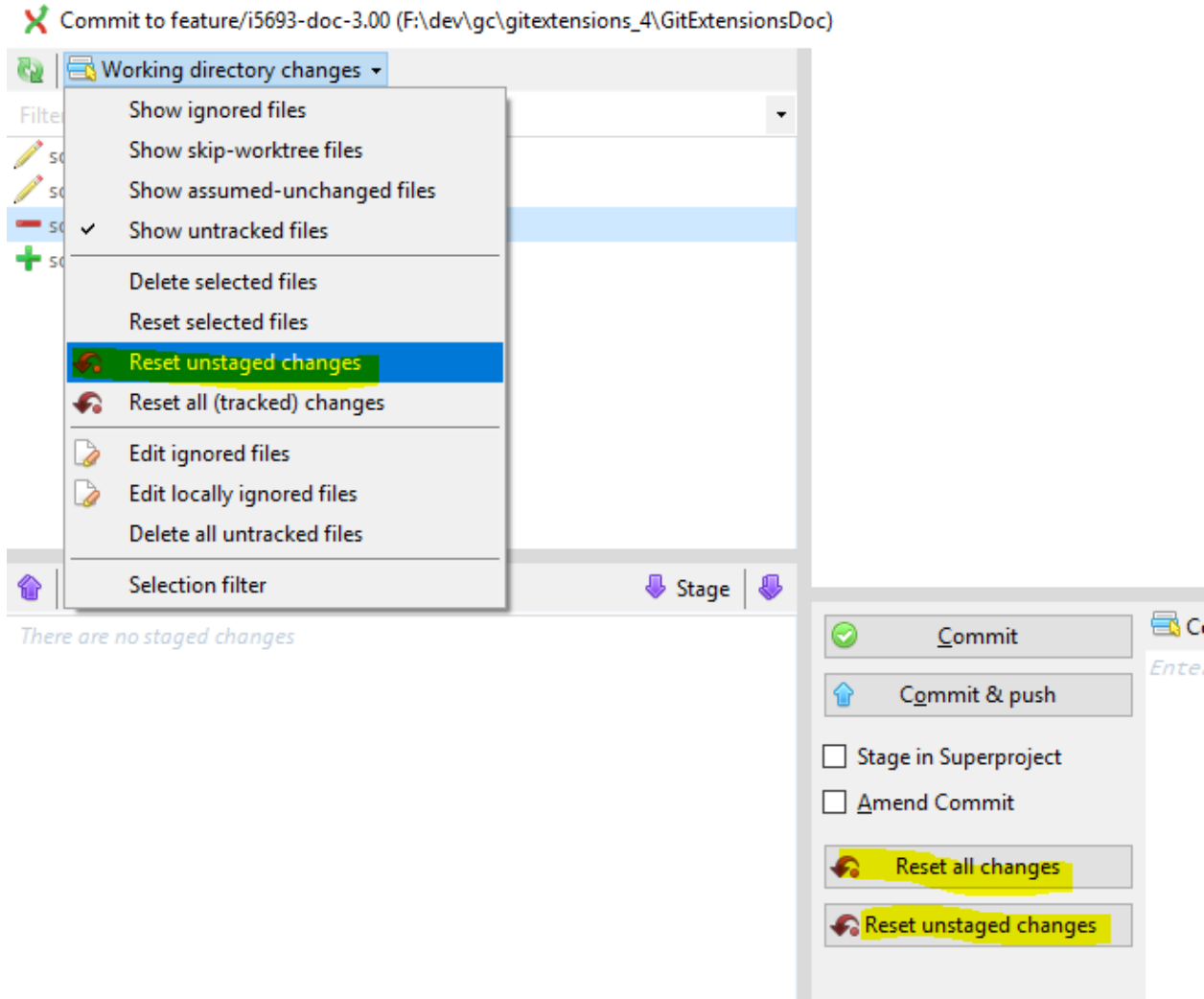
---

**Note:** Staging and unstaging individual lines from a file does not change the file itself. It is simply choosing which changes from within that file will be included in the next commit.

---

### 6.1.3 Undoing or resetting changes

You can undo or reset changes to files from the commit dialog. You can only do this from the top-left or “Unstaged changes” pane. If you have already staged the changes then you must first unstage them as described above. To reset the changes in a file, select the file in the unstaged changes pane, right-click and choose `Reset file or directory changes` or press the `[R]` key.

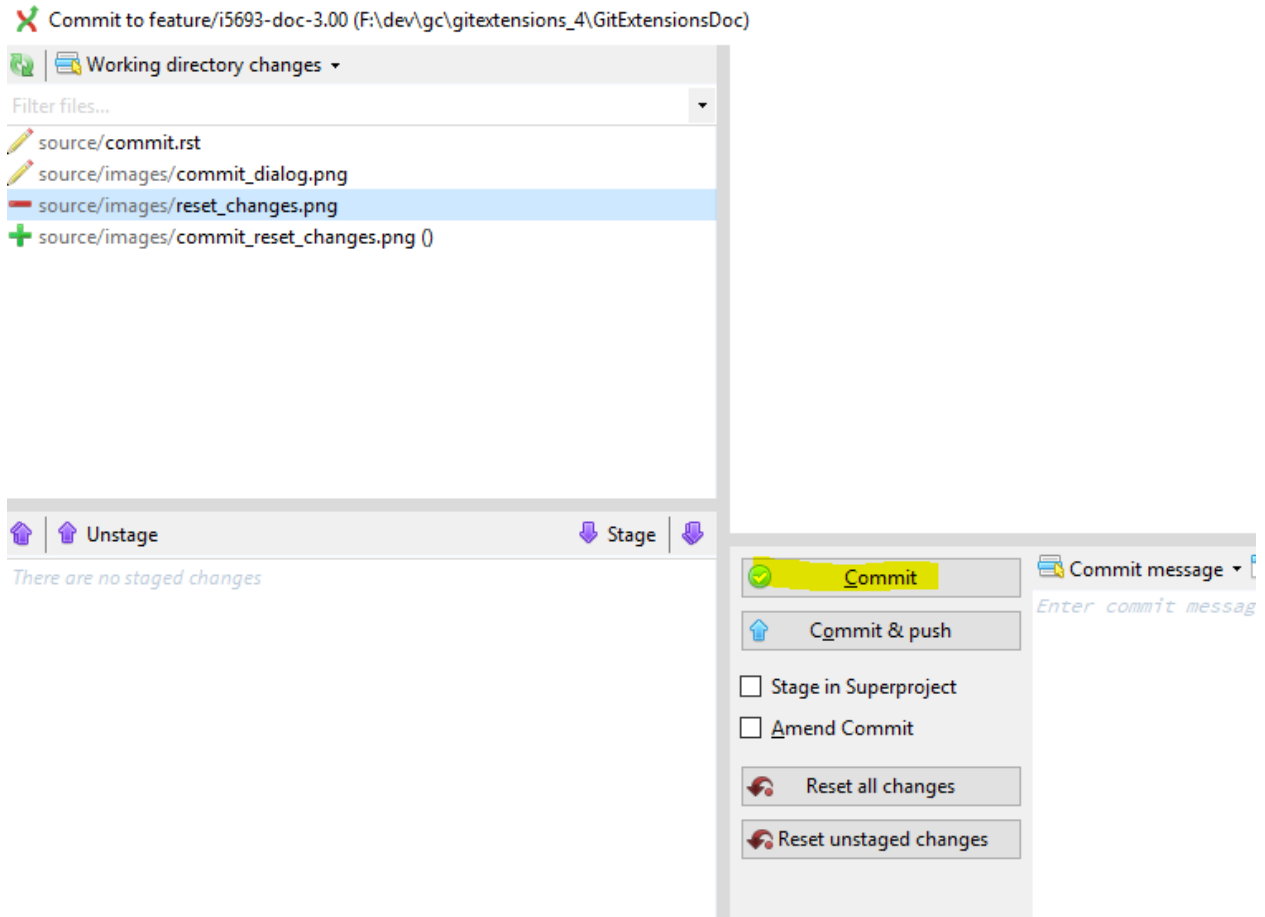


You can reset individual changed lines in a similar way to staging and unstaging individual lines, which are described above. To reset an individual line, select the line or lines in the diff view on the right then right-click and choose `Reset selected lines` or press the `[R]` key.

**Warning:** Resetting changes modifies the file, discarding either all of the changes or the changes on the selected lines.

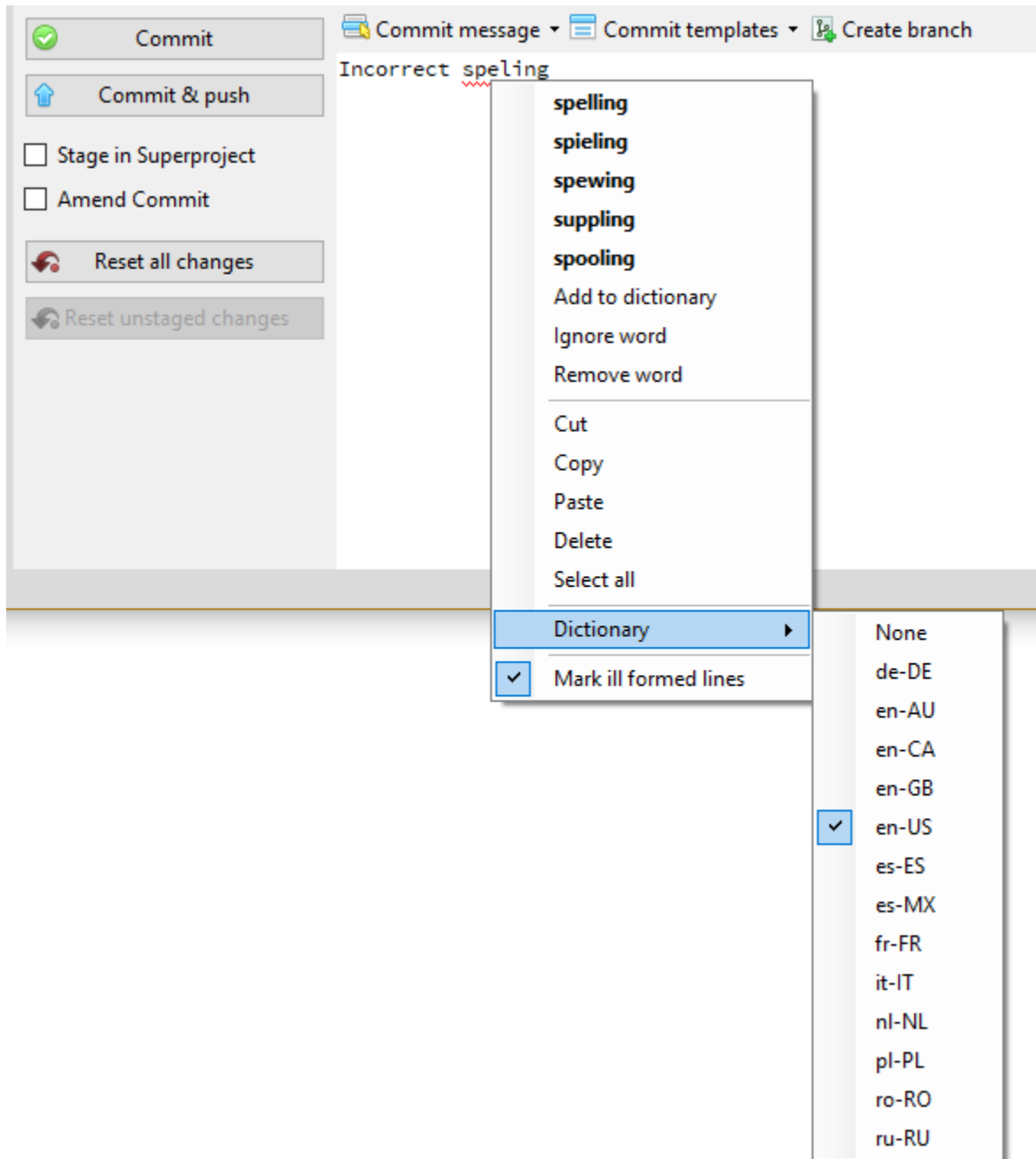
### 6.1.4 Making the commit

When all the changes you want to commit are staged, enter a commit message into the lower-right pane and press the commit button.



There is a built-in spelling checker that checks the commit message. Incorrectly spelled words are underlined with a wavy red line. Right-click on the misspelled word to choose the correct spelling or choose one of the other options.

Git Extensions installs a number of dictionaries by default. You can choose another language in the context menu of the spelling checker or in the settings dialog. To add a new spelling dictionary add the dictionary file to the `Dictionaries` folder inside the Git Extensions installation folder.

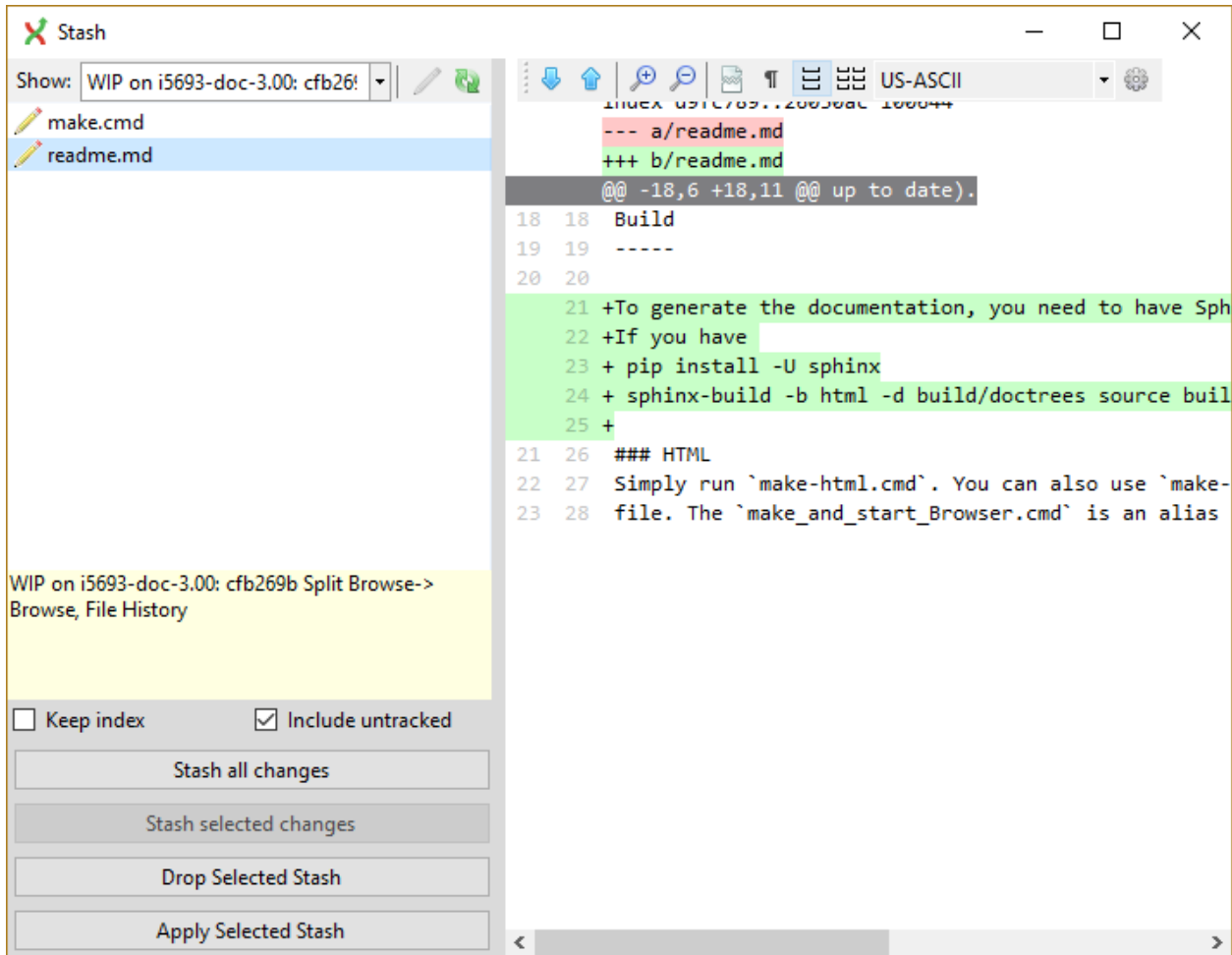


## 6.2 Amend commit

It is also possible to add changes to your last commit by checking the `Amend Commit` checkbox. This can be very useful when you forgot some changes. This function rewrites history; it deletes the last commit and commits it again including the added changes.

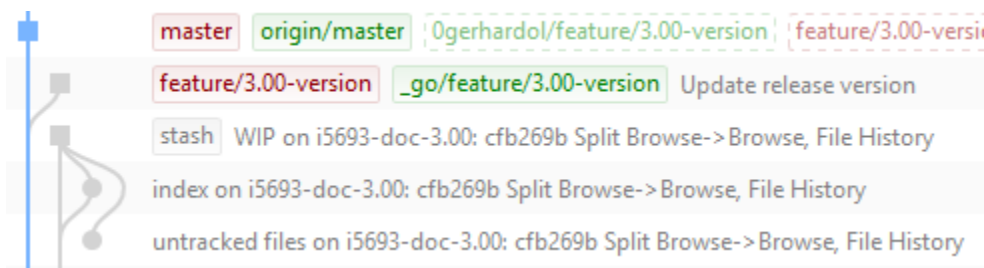
See also [Modify Git history](#), especially if you have published the changes to a remote repository already.

If there are local changes that you do not want to commit yet and not want to throw away either, you can temporarily stash them. This is useful when working on a feature and you need to start working on something else for a few hours. You can stash changes away and then reapply them to your working dir again later. Stashes are typically used for very short periods.



## 7.1 Revision graph

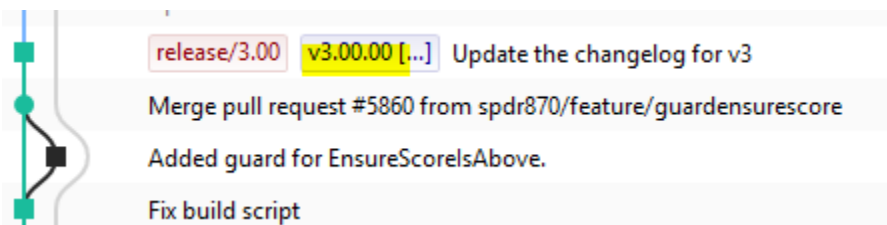
You can create multiple stashes if needed. The latest stash is shown in the commit log with the text `[stash]`, all stashes if `reflog` is visible (see *Maintenance*).



The stash is especially useful when pulling remote changes into a dirty working directory. If you want a more permanent stash, you should create a branch.

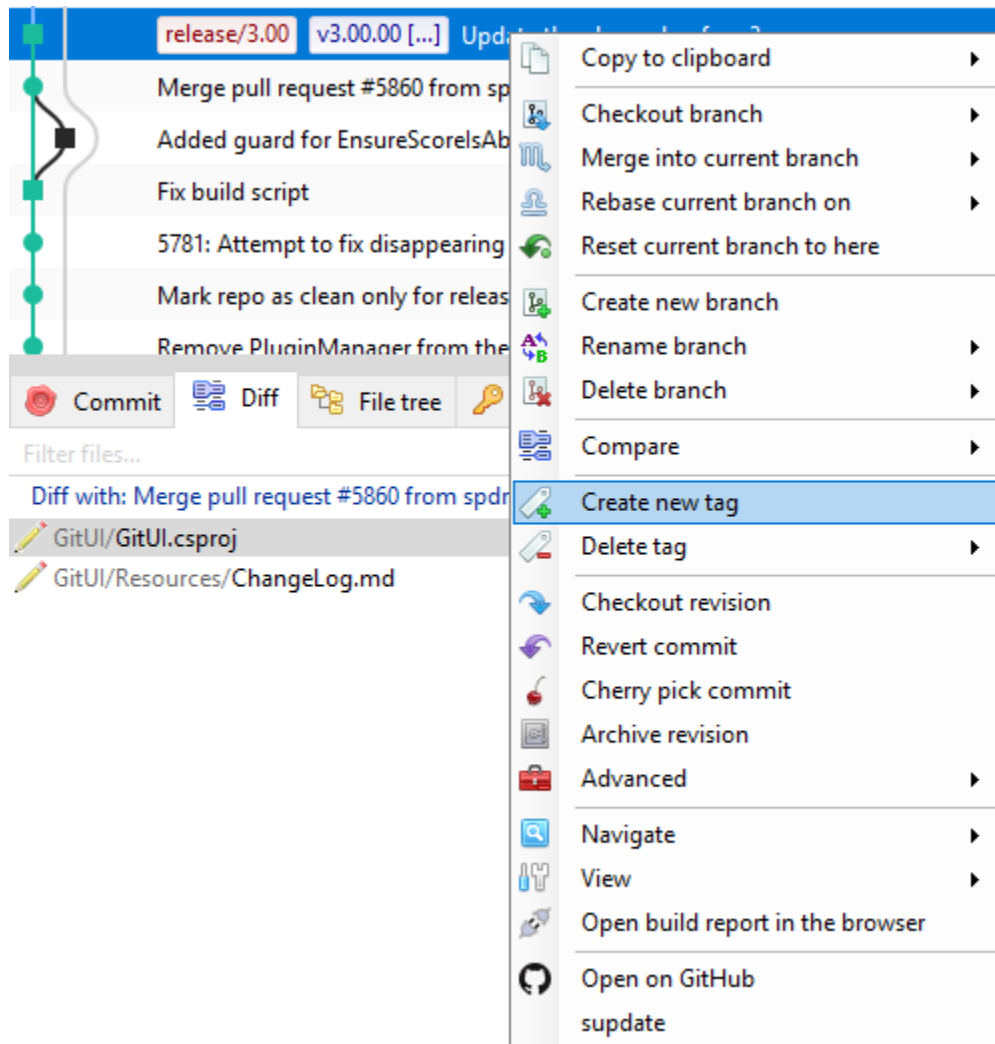


Tags are used to mark a specific version. Usually a tag will not be moved anymore. The image below shows the commit log of Git Extensions with a tag indicating version [3.00.00].



## 8.1 Create tag

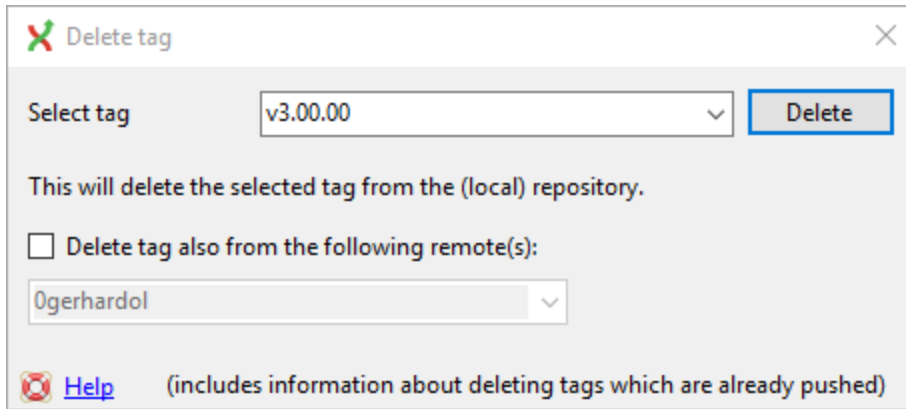
In Git Extensions you can tag a revision by choosing `Create new tag` in the commit log context menu. A dialog will prompt for the name of the tag. You can also choose `Create tag` from the `Commands` menu, which will show a dialog to choose the revision and enter the tag name.



Once a tag is created, it cannot be moved again. You need to delete the tag and create it again to move it.

## 8.2 Delete tag

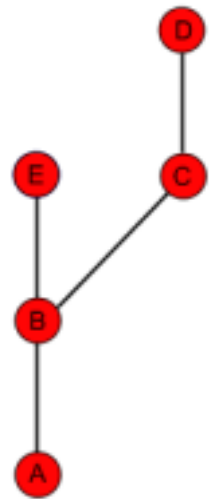
Tags can be deleted, read about “What should you do when you tag a wrong commit and you would want to re-tag?” here: [https://www.kernel.org/pub/software/scm/git/docs/git-tag.html#\\_on\\_re\\_tagging](https://www.kernel.org/pub/software/scm/git/docs/git-tag.html#_on_re_tagging)



---

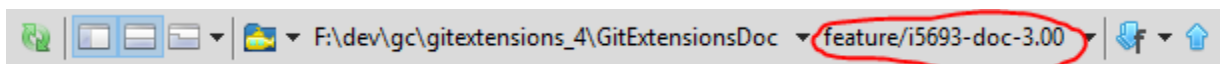
## Branches

---



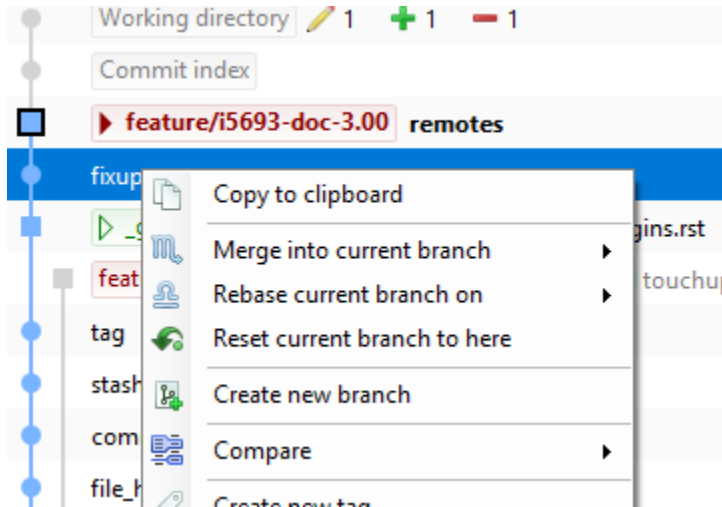
Branches are used to commit changes separate from other commits. It is very common to create a new branch when you start working on a feature to keep the work done on that feature separate from other work. When the feature is complete the branch can be merged or rebased as you choose such that the commits for the feature either remain as a parallel branch or appear as a continuous single line of development as if the branch had never existed in the first place. The image on the right illustrates a branch created on top of commit B.

You can see the name of your current branch in a combo box in the toolbar. You can switch to another branch by choosing from the combo box list. In the commit log the current branch has an arrow head to the left of its name. If you are not currently on a branch because you have checked out a specific commit but not any particular branch then Git Extensions will show `(no branch)` in place of a branch name in the toolbar. This is called “Detached HEAD mode”. In Git you can refer to your current branch or commit by the special reference `HEAD` in place of the branch name or commit reference.

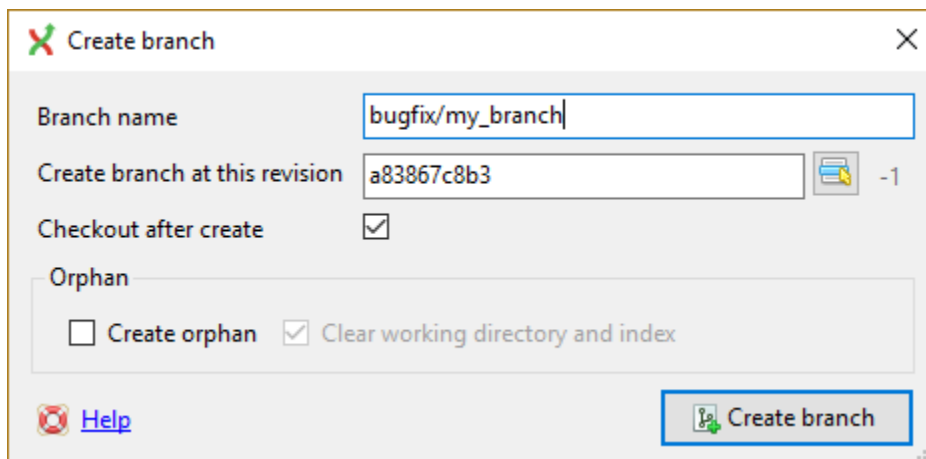


## 9.1 Create branch

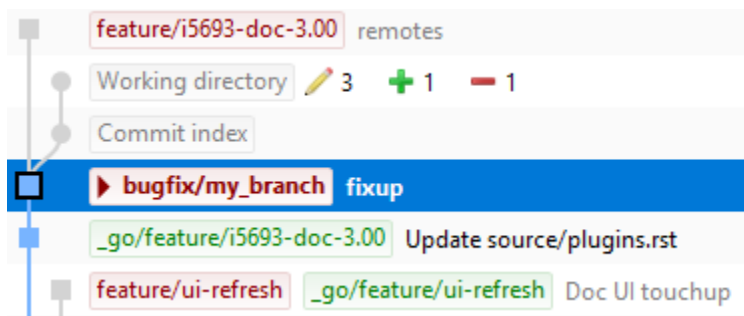
In Git Extensions there are multiple ways to create a new branch. In the image below I create a new branch from the context menu in the commit log. This will create a new branch on the revision that is selected.



I will create a new branch called `feature/refactor`. In this branch I can do whatever I want without affecting others. The default in Git Extensions is to check out a new branch after it is created. If you want to create a new branch but remain on your current branch, uncheck the `Checkout after create` checkbox in the `Create branch` dialog.



When the branch is created you will see the new branch `feature/refactor` in the commit log. If you chose to checkout this branch the next commit will be committed to the new branch.



Creating branches in Git requires only 41 bytes of space in the repository. Creating a new branch is very easy and fast. The complete work flow of Git is optimized for branching and merging.

### 9.1.1 Orphan branches

In special cases it is helpful to have orphan branches (see for example <https://www.google.com/search?q=why+use+orphan+branches+in+git>). Check the “Create orphan” checkbox to create an orphan branch (`--orphan` option in `git`).

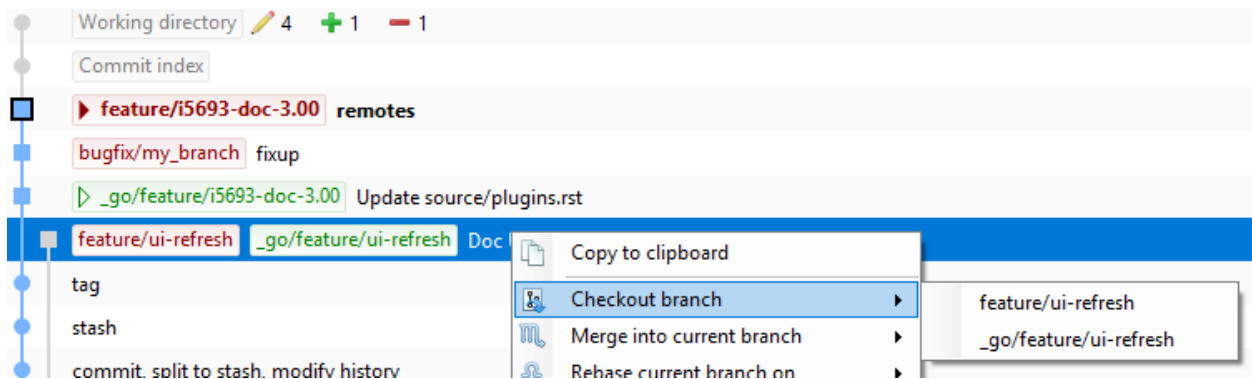
The newly created branch will have no parent commits.

The option “Clear working dir and index” (`git rm -rf`) is active by default. So the working dir and index will be cleared. If you uncheck the last option then the working dir and index will not be touched.

## 9.2 Checkout branch

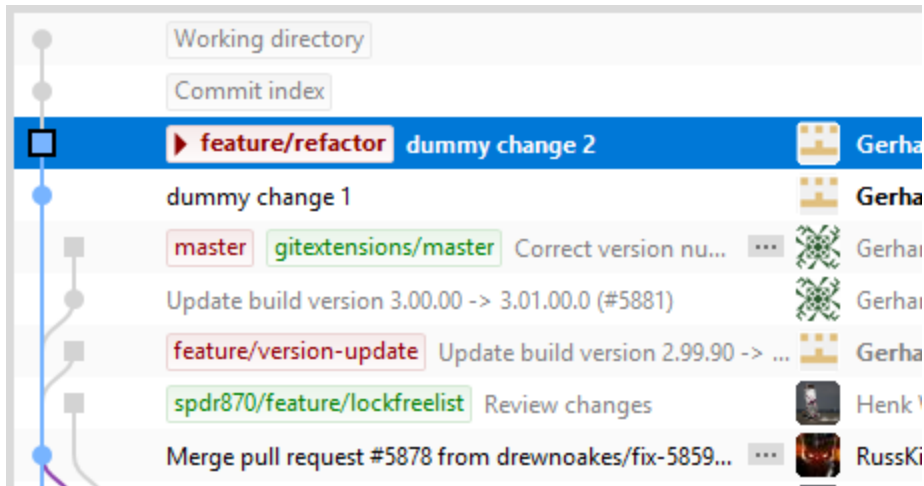
You can switch from the current branch to another branch using the checkout command. Checking out a branch sets the current branch and updates all of the source files in the working directory. Uncommitted changes in the working directory can be overwritten so it is best practice to make sure your working directory is clean by either committing or stashing any current changes before checking out a branch. If you do not clean your working directory then, in the `Checkout branch` dialog, you can choose between four options for your local uncommitted changes:

Don't change	Local changes will be retained if there are not conflicting changes from the branch you are checking out.
Merge	Performs a three-way merge between your current branch, your local changes and the branch you are checking out.
Stash	Your local changes are stashed and the new branch is checked out. You can retrieve your changes on the new branch with <code>stash-pop</code> .
Reset	Your local changes are discarded and the new branch is checked out. Use caution with this option as Git has no record of uncommitted changes so they cannot be retrieved.

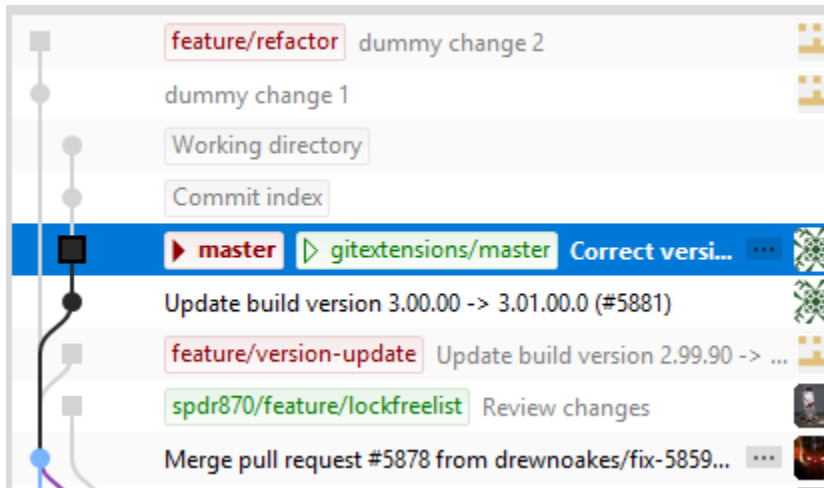


## 9.3 Merge branches

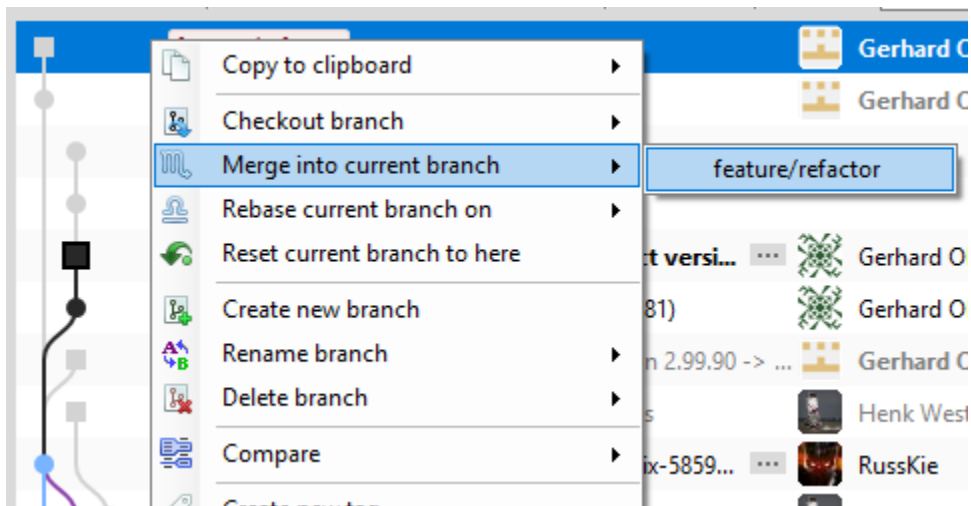
In the image below there are two branches, `[feature/refactor]` and `[master]`. We can merge the commits from the master branch into the feature/refactor branch. If we do this, the feature/refactor branch will be up to date with the master branch, but not the other way around. As long as we are working on the feature/refactor branch we cannot touch the master branch itself. We can merge the sources of master into our branch, but cannot make any change to the master branch.



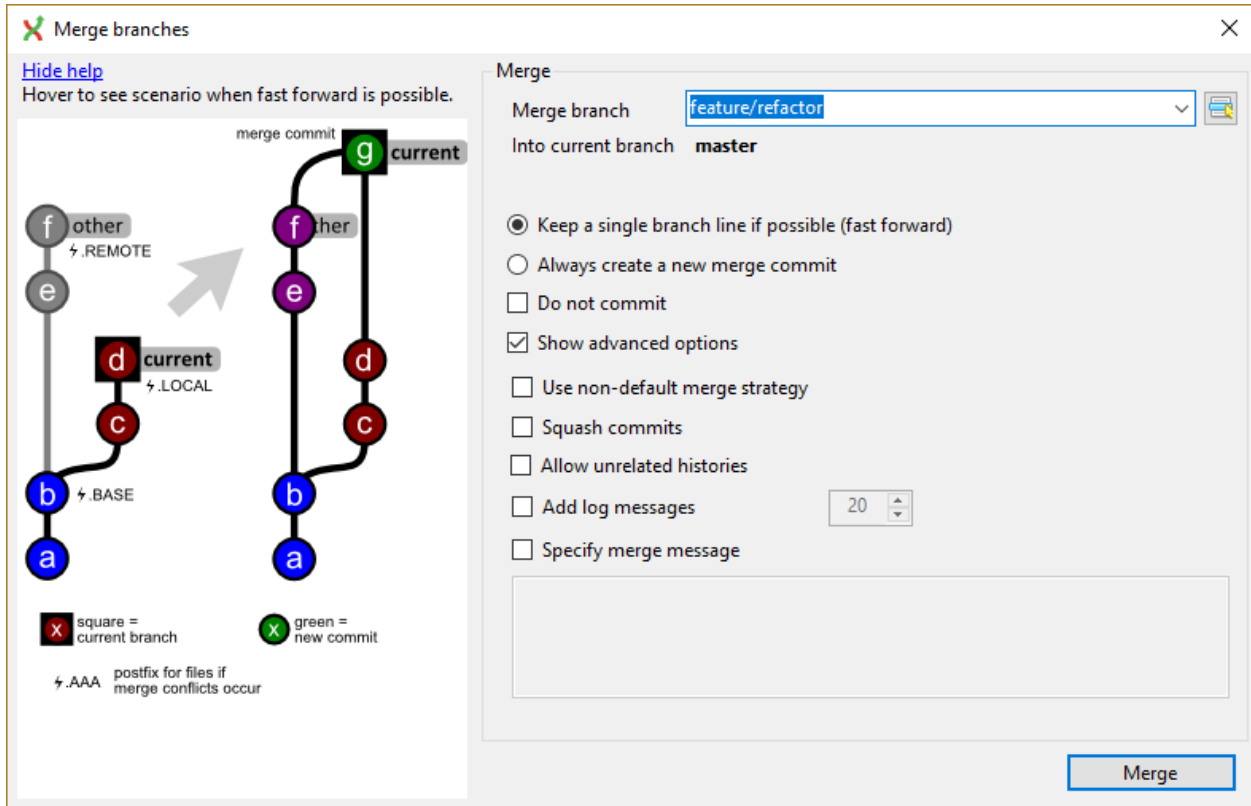
To merge the feature/refactor branch into the master branch, we first need to switch to the master branch.



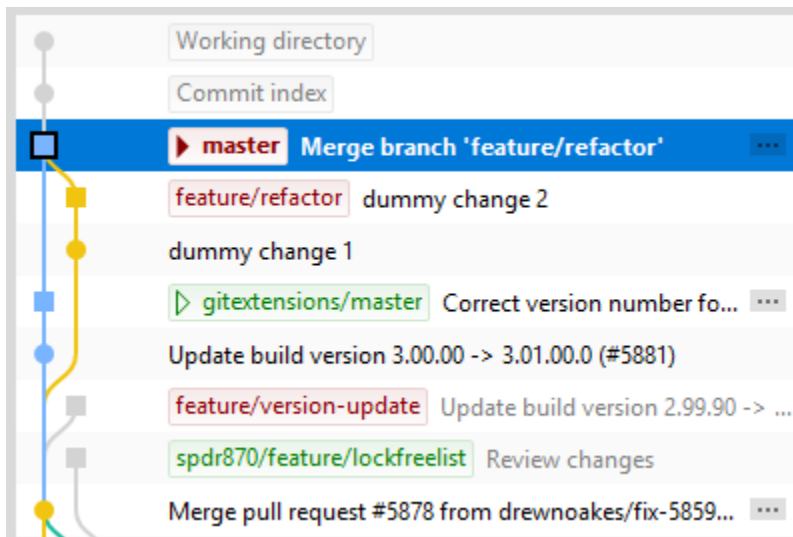
Once we are on the master branch, select the feature/refactor branch and select merge. Alternatively choose Merge branches from the Commands menu and select the feature/refactor branch.



In the merge dialog you can verify which branch you are working on. Select the branch to merge with then click the Merge button.



After the merge the commit log will show the new commit containing the merge. Notice that the feature/refactor branch is not changed by this merge. If you want to continue working on the feature/refactor branch you can merge the feature/refactor branch with master. You can instead delete the feature/refactor branch if it is not used anymore.



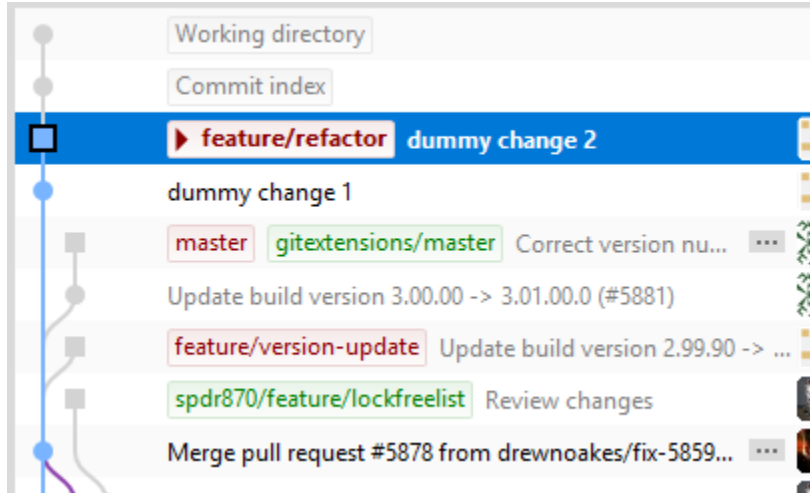
**Note:** When you need to merge with an unnamed branch you can use a tag to give it a temporary name.

**Note:** During a merge conflicts can occur. See [Merge Conflicts](#) for more information.

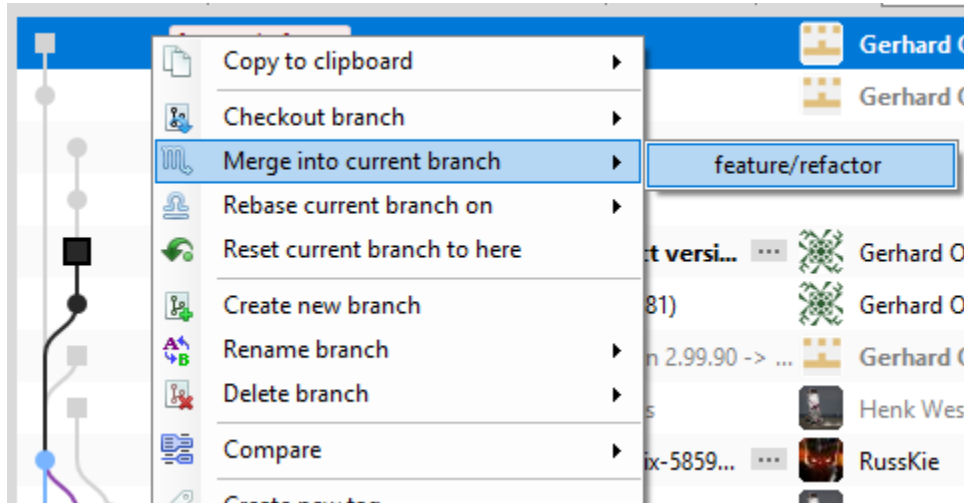


## 9.4 Rebase branch

The rebase command is the most complex command in Git. The rebase command is very similar to the merge command. Both rebase and merge are used to get a branch up-to-date. The main difference is that rebase can be used to keep the history linear contrary to merges.



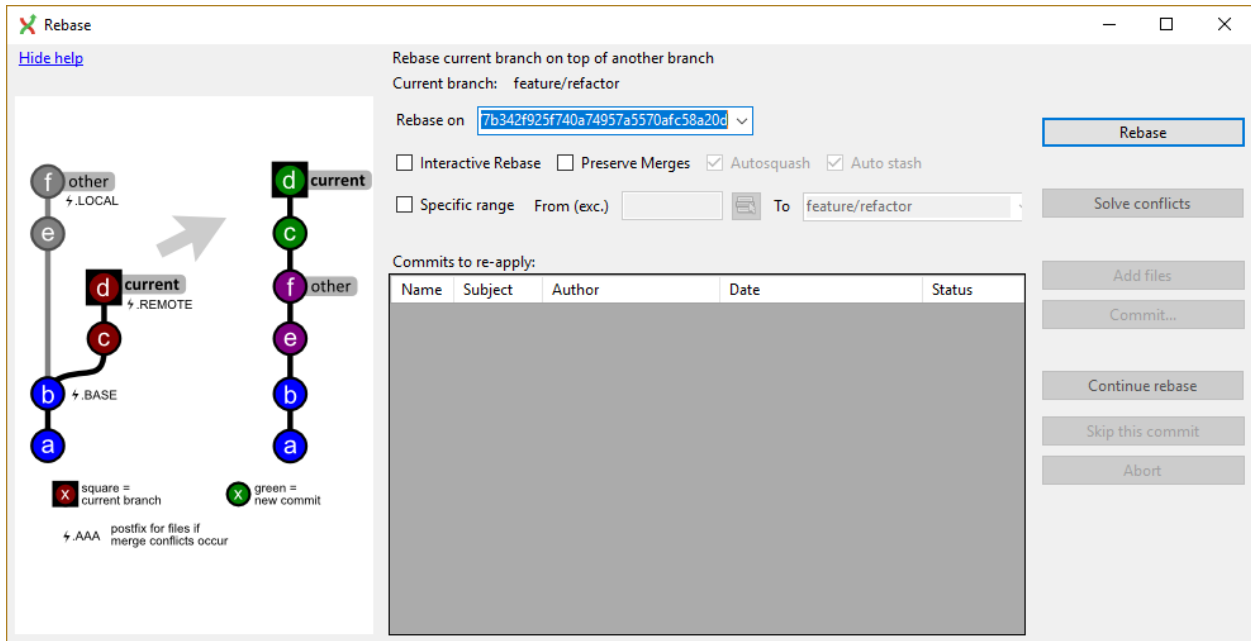
Select the commit where you want to rebase the current branch.



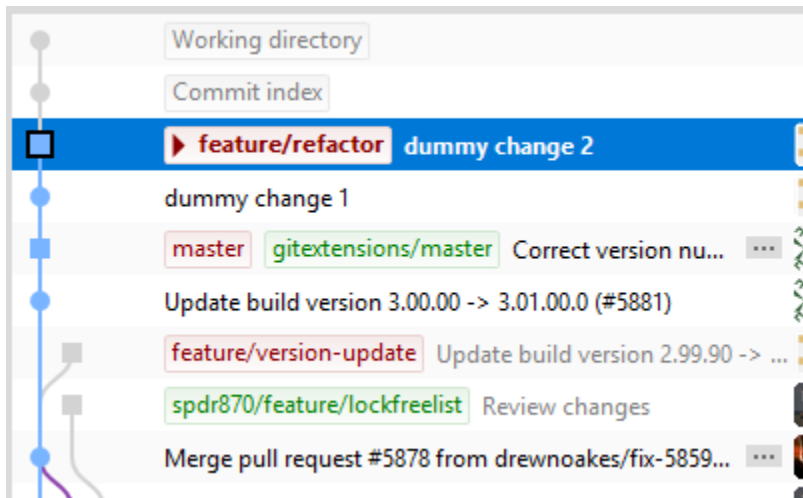
A rebase of feature/refactor on top of master will perform the following actions:

- All commits specific to the feature/refactor branch will be stashed in a temporary location
- The branch feature/refactor will be removed
- The branch feature/refactor will be recreated on the master branch
- All commits will be recommitted in the new feature/refactor branch

**Note:** During a rebase merge conflicts can occur. You need to solve the merge conflicts for each commit that is rebased. The rebase function in Git Extensions will guide you through all steps needed for a successful rebase. See [Merge Conflicts](#) for more information.



The image below shows the commit log after the rebase. Notice that the history is changed and it seems like the commits on the feature/refactor branch are created after the commits on the master branch.



**Warning:** Because this function rewrites history you should only use this on branches that are not published to other repositories yet. When you rebase a branch that is already pushed it will be harder to pull or push to that remote. If you want to get a branch up-to-date that is already published you should merge.

## 9.5 Interactive rebase

It is possible to modify the order, merge commits etc when committing.

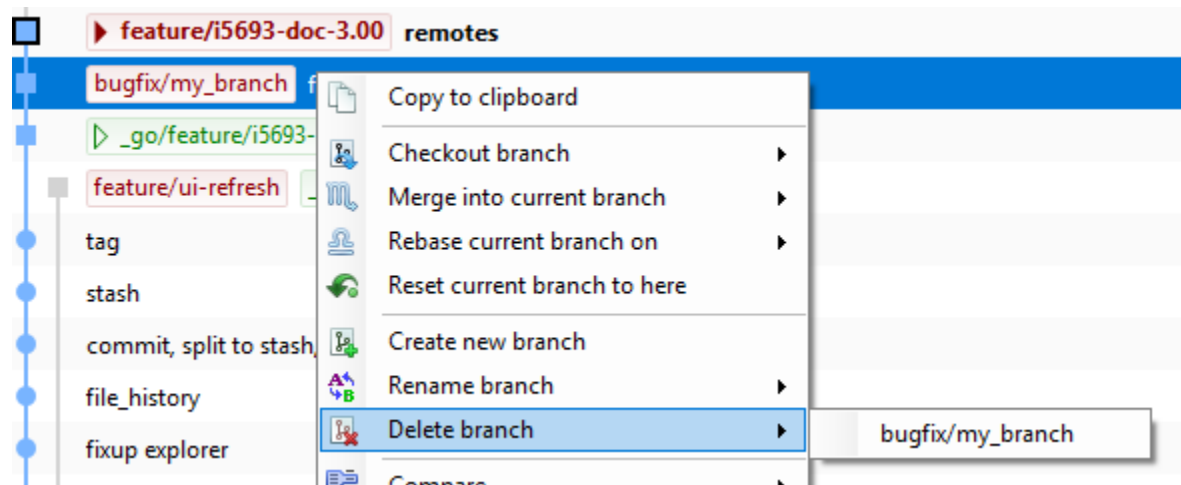
See *Modify Git history* for more information.

## 9.6 Delete branch

Since it is common to create many branches, it is often necessary to delete branches. Most commonly you will need to delete branches on which work has finished and their contents are merged into master or your main branch. You can also delete unmerged branches when they are not needed anymore and you do not want to keep the work done in that branch.

When you delete a branch that is not yet merged, all of the commits that are in only the deleted branch will be lost. When you delete a branch that is already merged with another branch, the merged commits will not be lost because they are also part of another branch.

You can delete a branch using `Delete branch` from the `Commands` menu. If you want to delete a branch that is not merged into your current branch (HEAD in Git), you need to check the `Force delete` checkbox.



# CHAPTER 10

---

## Patches

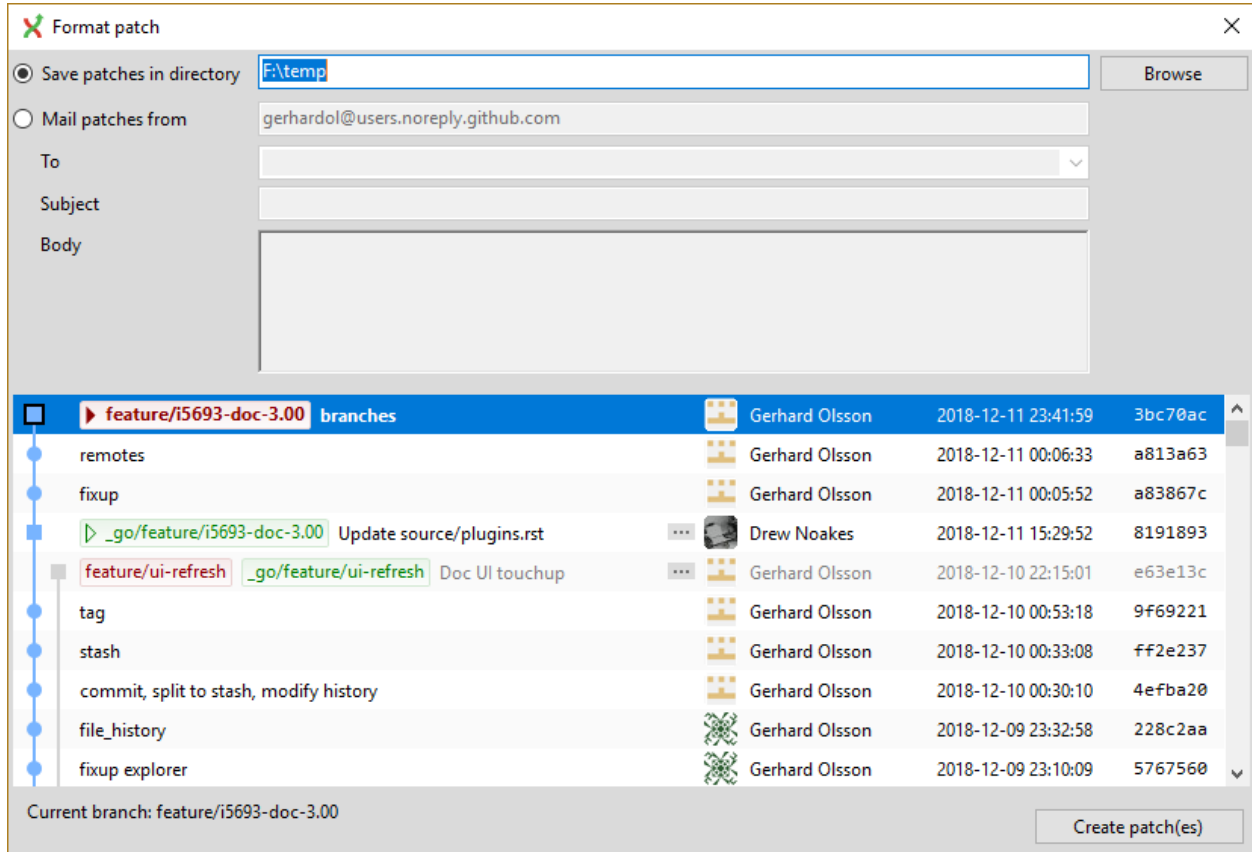
---

Every commit contains a change-set, a commit date, the committer name, the commit message and a cryptograph SHA1 hash. Local commits can be published by pushing it to a remote repository. To be able to push you need to have sufficient rights and you need to have access to the remote repository. When you cannot push directly you can create patches. Patches can be e-mailed to someone with access to the repository. Each patch contains an entire commit including the commit message and the SHA1.

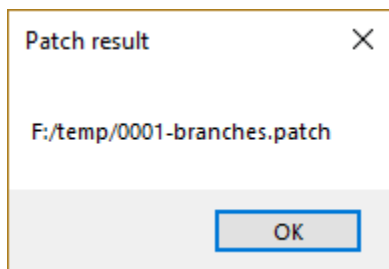
```
1 |From 58c02ec4701c94c671a41e1e5d50c582e859851f Mon Sep 17 00:00:00 2001
2 |From: Russell King <rmk@dyn-67.arm.linux.org.uk>
3 |Date: Sun, 17 Apr 2005 15:40:46 +0100
4 |Subject: [PATCH 000213/123824] [PATCH] ARM: h3600_irda_set_speed arguments
5
6 |h3600_irda_set_speed() had the wrong type for the "speed" argument.
7 |Fix this.
8
9 |Signed-off-by: Russell King <rmk@arm.linux.org.uk>
10 |---
11 | arch/arm/mach-sa1100/h3600.c |    2 +-
12 | 1 files changed, 1 insertions(+), 1 deletions(-)
13
14 |diff --git a/arch/arm/mach-sa1100/h3600.c b/arch/arm/mach-sa1100/h3600.c
15 |index 9788d3a..84c8654 100644
16 |--- a/arch/arm/mach-sa1100/h3600.c
17 |+++ b/arch/arm/mach-sa1100/h3600.c
18 |@@ -130,7 +130,7 @@ static int h3600_irda_set_power(struct device *dev, unsigned int state)
19 |     return 0;
20 | }
21
22 |-static void h3600_irda_set_speed(struct device *dev, int speed)
23 |+static void h3600_irda_set_speed(struct device *dev, unsigned int speed)
24 | {
25 |     if (speed < 4000000) {
26 |         clr_h3600_egpio(IPAQ_EGPIO_IR_FSEL);
27 |     }
28 | 1.6.1.9.g97c34
```

## 10.1 Create patch

Format a single patch or patch series using the format patch dialog. You need to select the newest commit first and then select the oldest commit using ctrl-click. You can also select an interrupted patch series, but this is not recommended because the files will not be numbered.



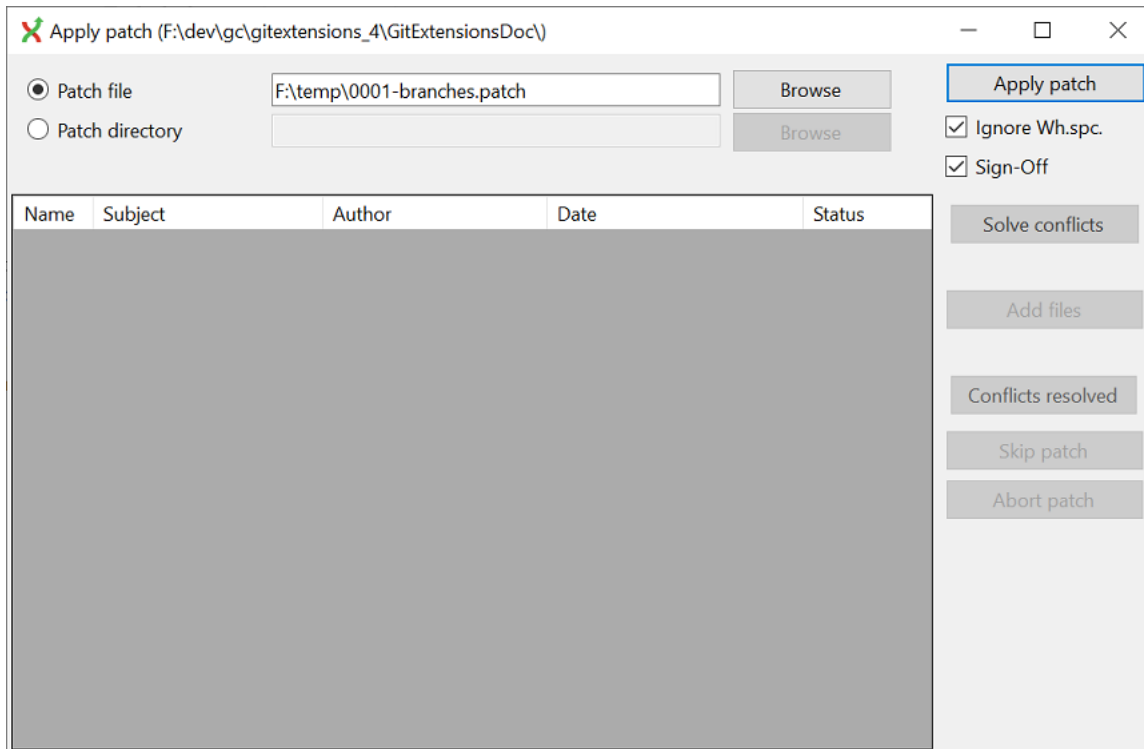
When the patches are created successfully the following dialog will appear.



## 10.2 Apply patches

It is possible to apply a single patch file or all patches in a directory. When there are merge conflicts applying the patch you need to resolve them before you can continue. Git Extensions will help you applying all patches by marking the next recommended step.

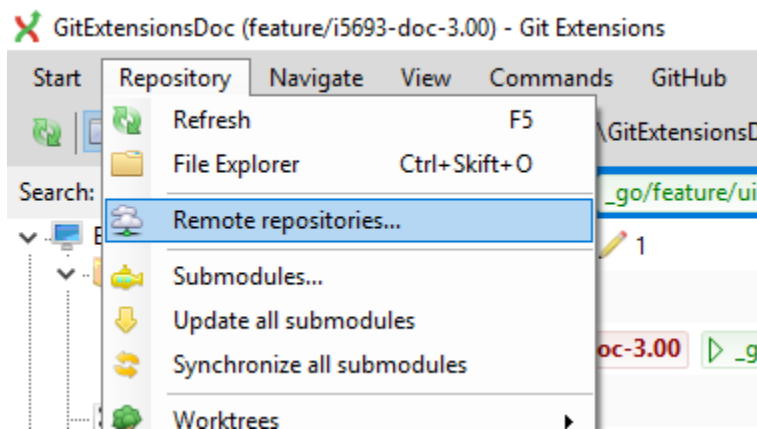
Use 'Sign-Off' checkbox to sign off commits of applying patch. Git Extensions will remember your choice.



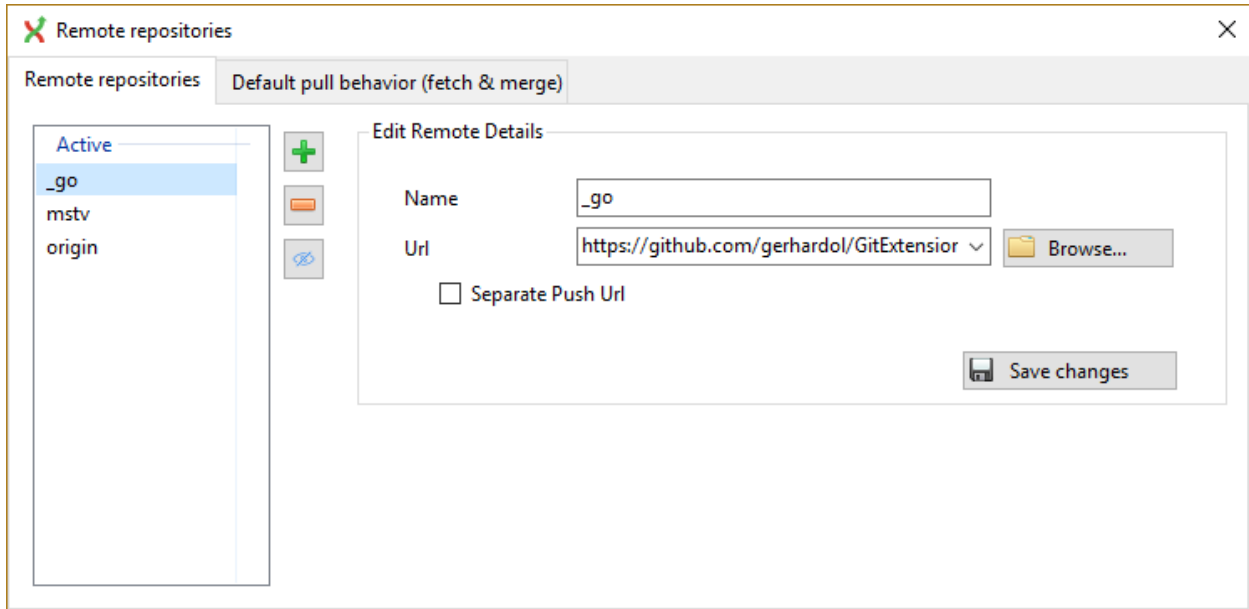
Git is a distributed source control management system. This means that all changes you make are local. When you commit changes, you only commit them to your local repository. To publish your local changes you need to push. In order to get changes committed by others, you need to fetch/pull.

### 11.1 Manage remote repositories

You can manage the remote repositories in the `Remotes` menu.

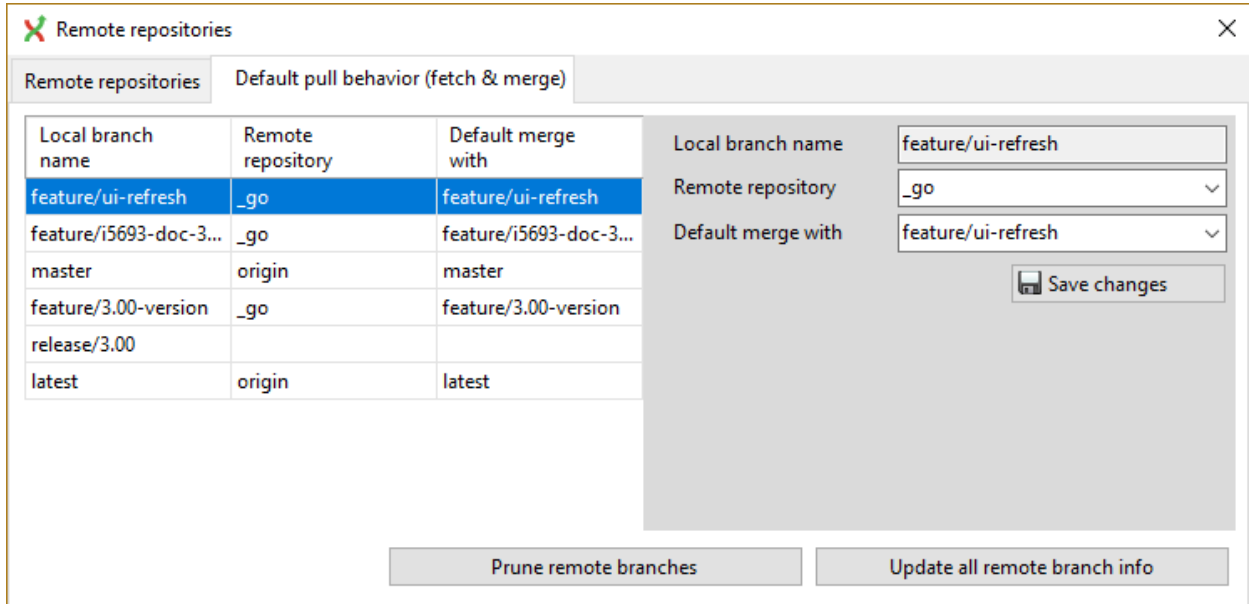


When you cloned your repository from a public repository, this remote is already configured. You can rename each remote for easy recognition. The default name after cloning a remote is `origin`. If you use PuTTY as SSH client you can also enter the private key file for each remote. Git Extensions will load the key when needed. How to create a private key file is described in the next paragraph.



In the `Default pull behaviour` tab you can configure the branches that need to be pulled and merged by default. If you configure this correctly you will not need to choose a branch when you pull or push. There are two buttons on this dialog:

Prune remote branches	Throw away remote branches that do not exist on the remote anymore.
Update all remote branch info	Fetch all remote branch information.



After cloning a repository you do not need to configure all remote branches manually. Instead you can checkout the remote branch and choose to create a local tracking branch.



## 11.2 Git Credential Manager

The Git Credential Manager can be used to authenticate http links. For more information and instructions, see <https://github.com/Microsoft/Git-Credential-Manager-for-Windows>

## 11.3 Create SSH key

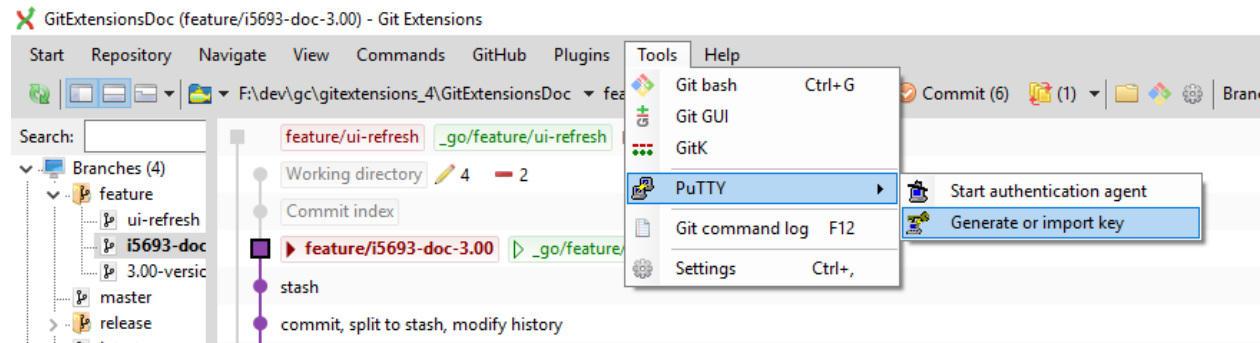
Git uses SSH for accessing private repositories. SSH uses a public/private key pair for authentication. This means you need to generate a private key and a public key. The private key is stored on your computer locally and the public key can be given to anyone. SSH will encrypt whatever you send using your secret private key. The receiver will then use the public key you send to decrypt the data.

This encryption will not protect the data itself but it protects the authenticity. Because the private key is only available to the sender, the receiver can be sure about the origin of the data. In practise the key pair is only used for the authentication process. The data itself will be encrypted using a key that is exchanged during this initial phase.

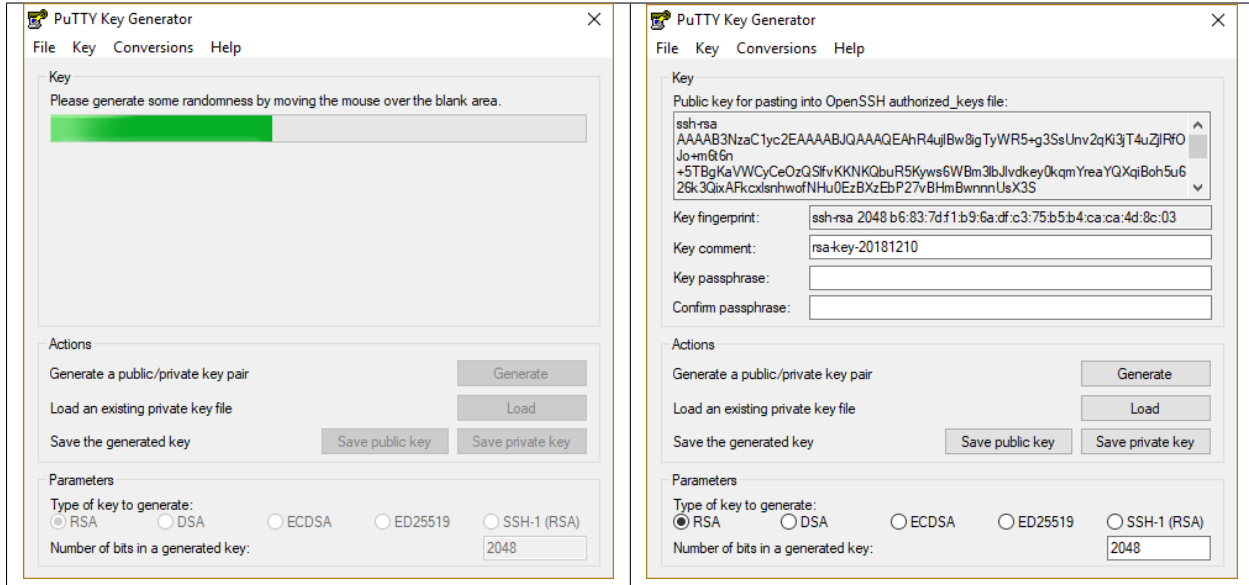
### 11.3.1 PuTTY and github

PuTTY is SSH client that for Windows that is a bit more user friendly then OpenSSH. Unfortunately PuTTY does not work with all servers. In this paragraph I will show how to generate a key for github using putty.

First make sure GitExtensions is configured to use PuTTY and all paths are correct, see [SSH](#)

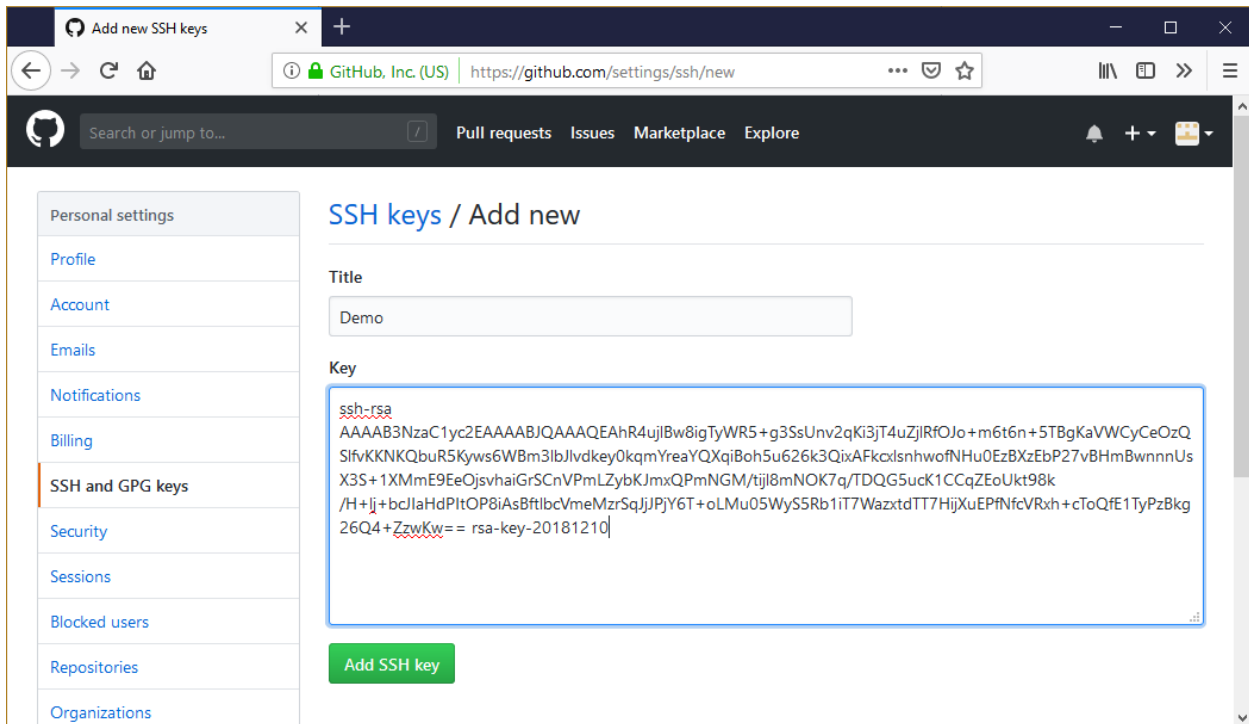


can choose `Generate` or `import key` to start the key generator.



PuTTY will ask you to move the mouse around to generate a more random key. When the key is generated you can save the public and the private key in a file. You can choose to protect the private key with a password but this is not necessary.

Now you have a key pair you need to give github the public key. This can be done in Account Settings in the tab SSH Public Keys. You can add multiple keys here, but you only need one key for all repositories.



After telling github what public key to use to decrypt, you need to tell GitExtensions what private key to use to encrypt. Load the private key into the PuTTY authentication agent in Clone dialog or by starting the PuTTY authentication agent and choose add key in the context menu in the system tray.

GitExtensions can load the private keys automatically for you when communicating with a remote. You need to

configure the private key for the remote.

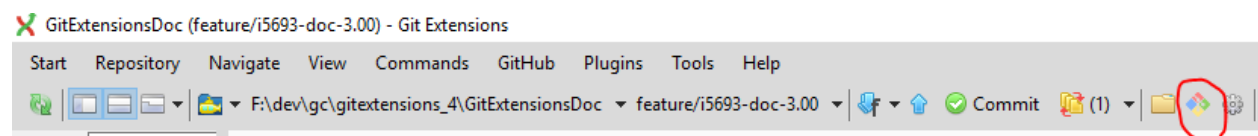
This is done in the `Manage remote repositories` dialog.

### 11.3.2 OpenSSH and github

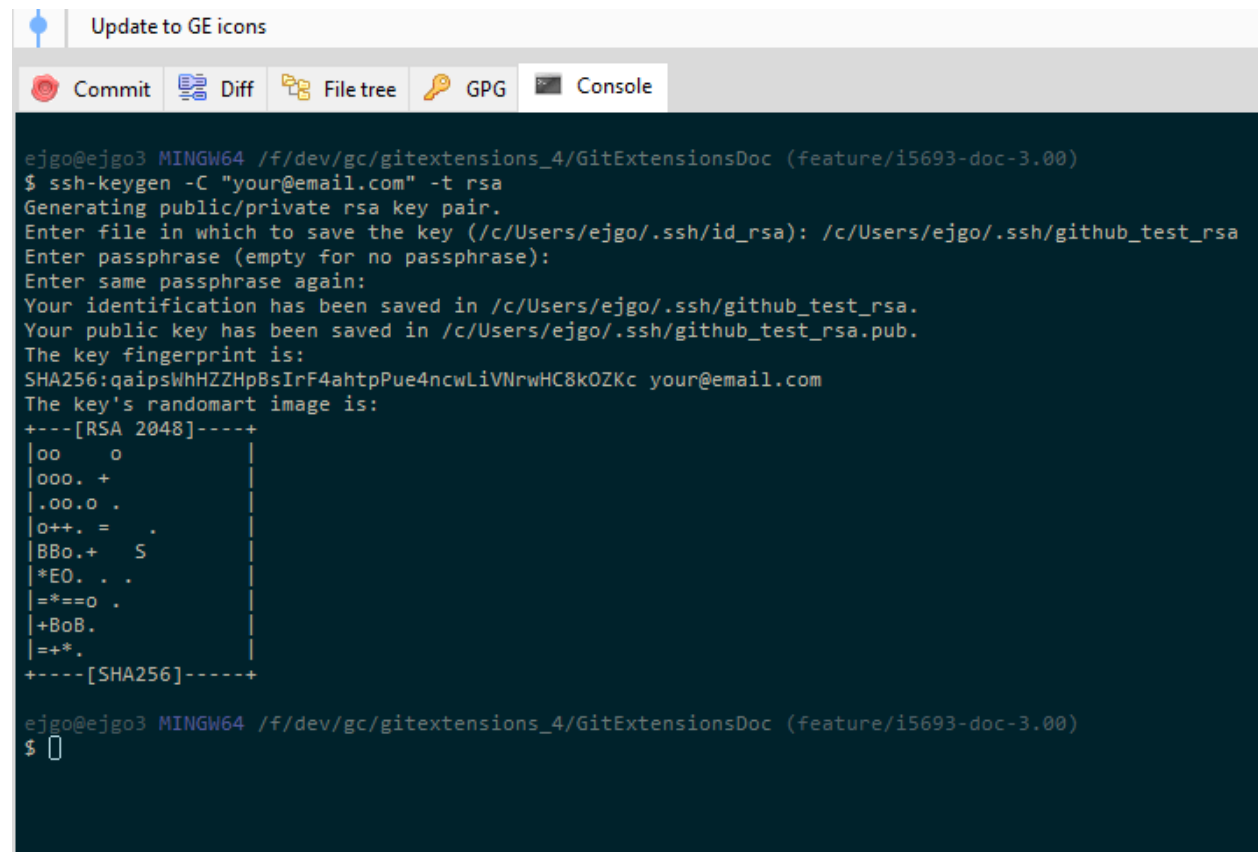
To configure GitExtensions to use OpenSSH, see [SSH](#).

OpenSSH is the best SSH client there is but it lacks Windows support. Therefore it is slightly more complex to use. Another drawback is that GitExtensions cannot control OpenSSH and needs to show the command line dialogs when OpenSSH might be used. GitExtensions will show the command line window for every command that might require a SSH connection. For this reason PuTTY is the preferred SSH client in GitExtensions.

To generate a key pair in OpenSSH you need to go to the command line. I recommend to use the `git bash` because the path to OpenSSH is already set. Open the separate Git bash or the console tab.



Type the following command: `ssh-keygen -C "your@email.com" -t rsa` Use the same email address as the email address used in git. You will be asked where if you want to protect the private key with a password. This is not necessary. By default the public and private keys are stored in `c:\Documents and Settings\[User]\.ssh\` or `c:\Users\[user]\.ssh\`.

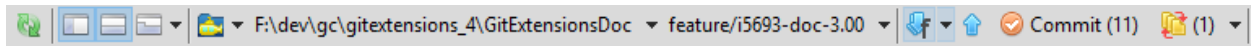


You do not need to tell GitExtensions about the private key because OpenSSH will load it for you. Now open the public key using notepad and copy the key to github. This can be done in `Account Settings` in the tab `SSH`

Public Keys on GitHub.

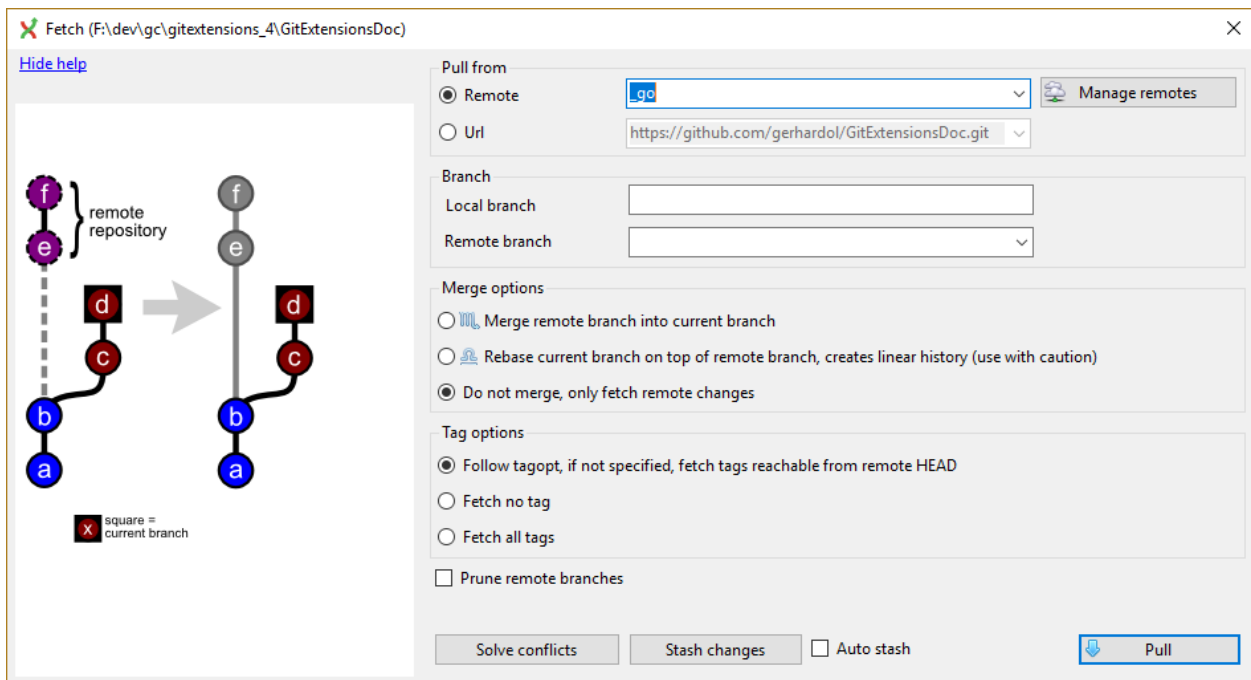
## 11.4 Pull changes

You can get remote changes using the pull function. Before you can pull remote changes you need to make sure there are no uncommitted changes in your local repository. If you have uncommitted changes you should commit them or stash them during the pull. You can read about how to use the stash in the Stash chapter.

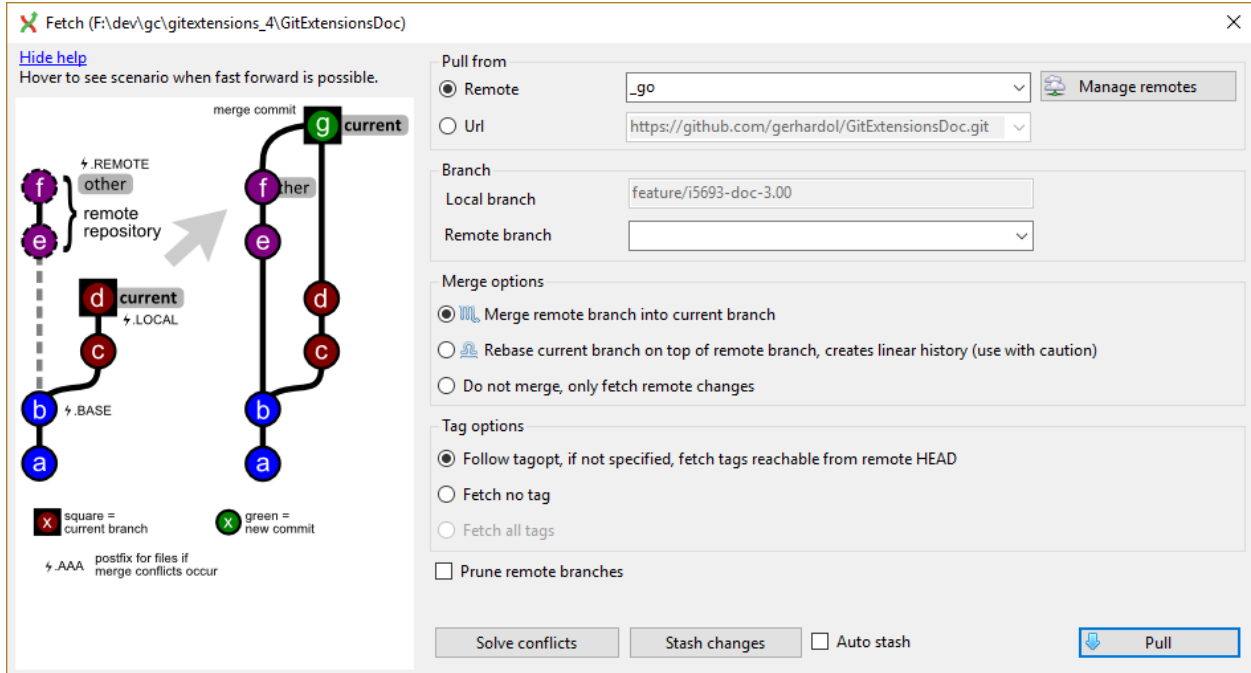


In order to get your personal repository up-to-date, you need to fetch changes from a remote repository. You can do this using the Pull dialog. When the dialog starts the default remote for the current branch is set. You can choose another remote or enter a custom url if you like. When the remote branches configured correctly, you do not need to choose a remote branch.

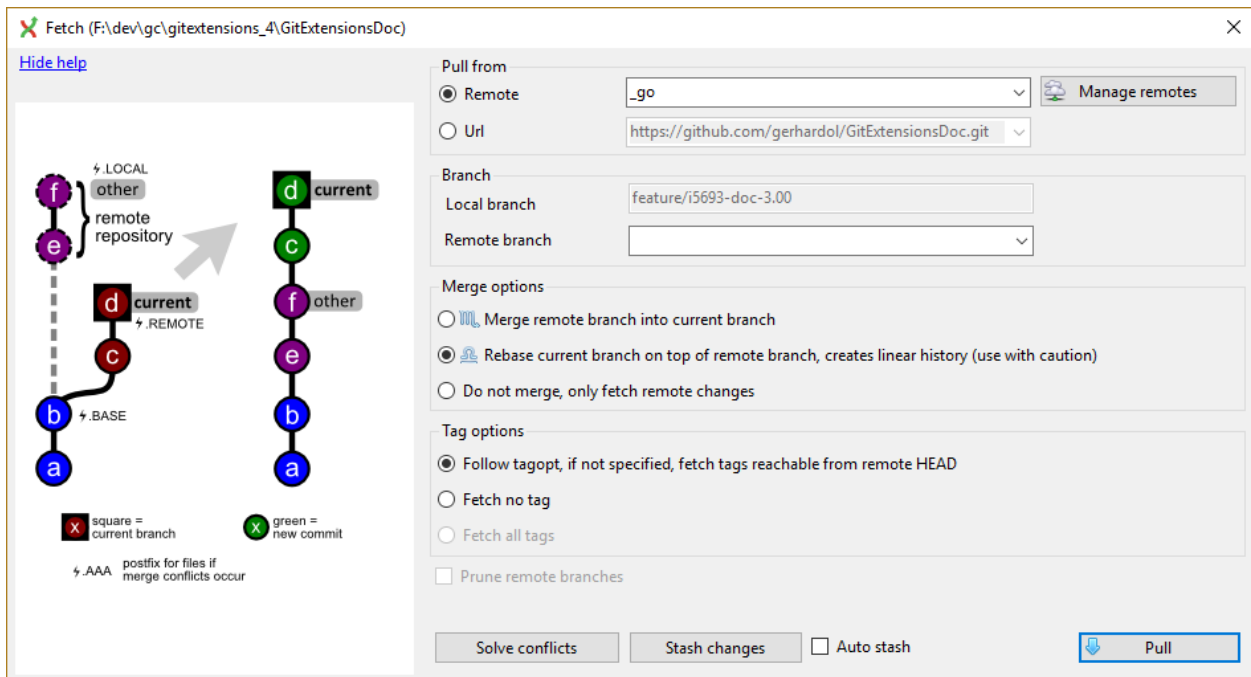
If you just fetch the commits from the remote repository and you already committed some changes to your local repository, the commits will be in a different branch. In the pull dialog this is illustrated in the image on the left. This can be useful when you want to review the changes before you want to merge them with your own changes.



When you choose to merge the remote branch after fetching the changes a branch will be created, and will be merged into your commit. Doing this creates a lot of branches and merges, making the history harder to read.



Instead of merging the fetched commits with your local commits, you can also choose to rebase your commits on top of the fetched commits. This is illustrated on the left in the image below. A rebase will first undo your local commits (c and d), then fetch the remote commits (e) and finally recommit your local commits. When there is a merge conflict during the rebase, the rebase dialog will show.

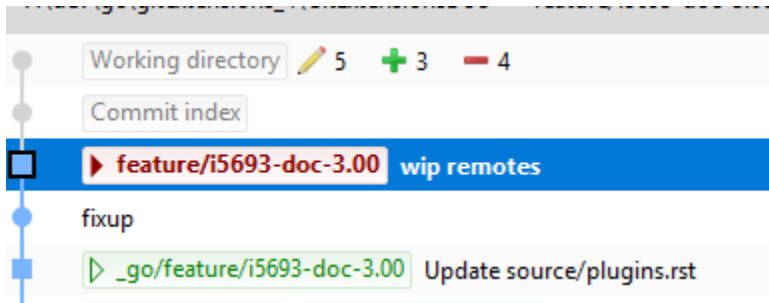


Next to the pull button there are some buttons that can be useful:

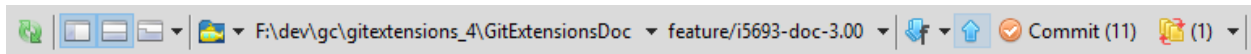
Solve conflicts	When there are merge conflicts, you can solve them by pressing this button.
Stash changes	When the working dir contains uncommitted changes, you need to stash them before pulling.
Auto stash	Check this checkbox if you want to stash before pulling. The stash will be reapplied after pulling.
Load SSH key	This button is only available when you use PuTTY as SSH client. You can press this button to load the key configured for the remote. If no key is set, a dialog will prompt for the key.

## 11.5 Push changes

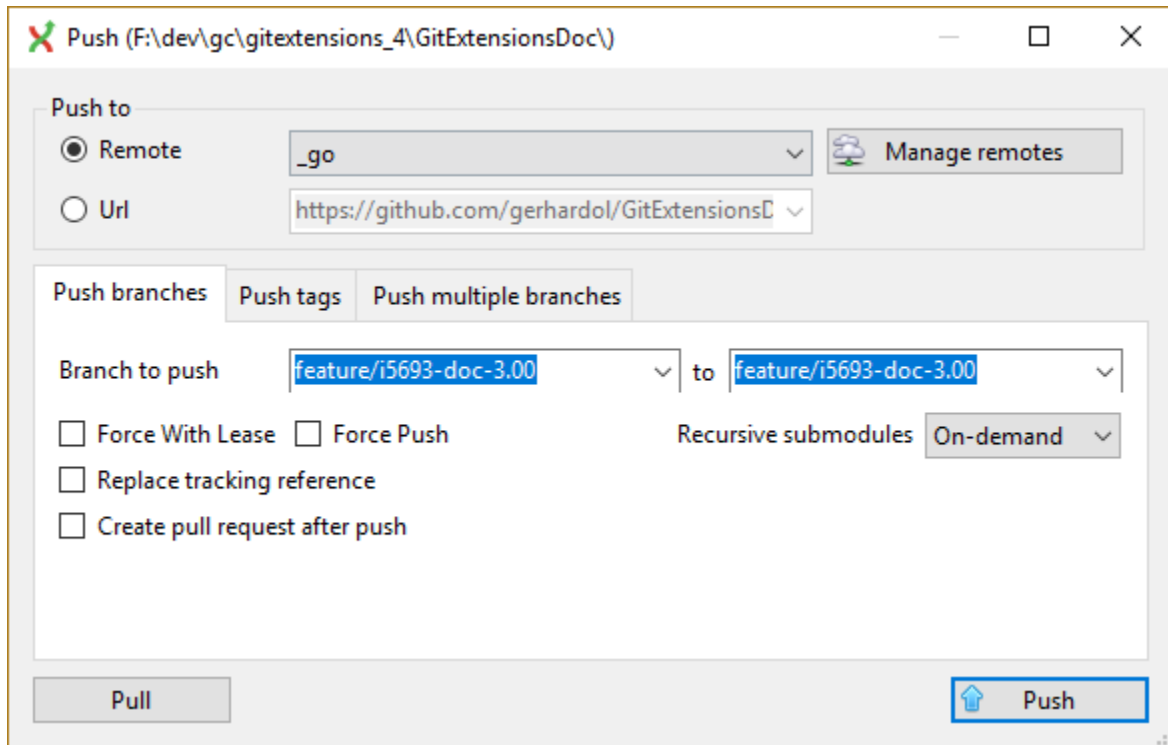
In the browse window you can check if there are local commits that are not pushed to a remote repository yet. In the image below the green labels mark the position of the master branch on the remote repository. The red label marks the position of the master branch on the local repository. The local repository is ahead three commits.



To push the changes press `Push` in the toolbar.



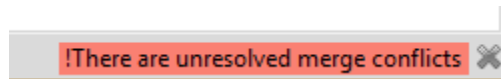
The push dialog allows you to choose the remote repository to push to. The remote repository is set to the remote of the current branch. You can choose another remote or choose a url to push to. You can also specify a branch to push.



Tags are not pushed to the remote repository. If you want to push a tag you need to open the `Tags` tab in the dialog. You can choose to push a single tag or all tags. No commits will be pushed when the `Tags` tab is selected, only tags.

You cannot merge your changes in the remote repository. Merging must be done locally. This means that you cannot push your changes before the commits are merged locally. In practice you need to pull before you can push most of the times.

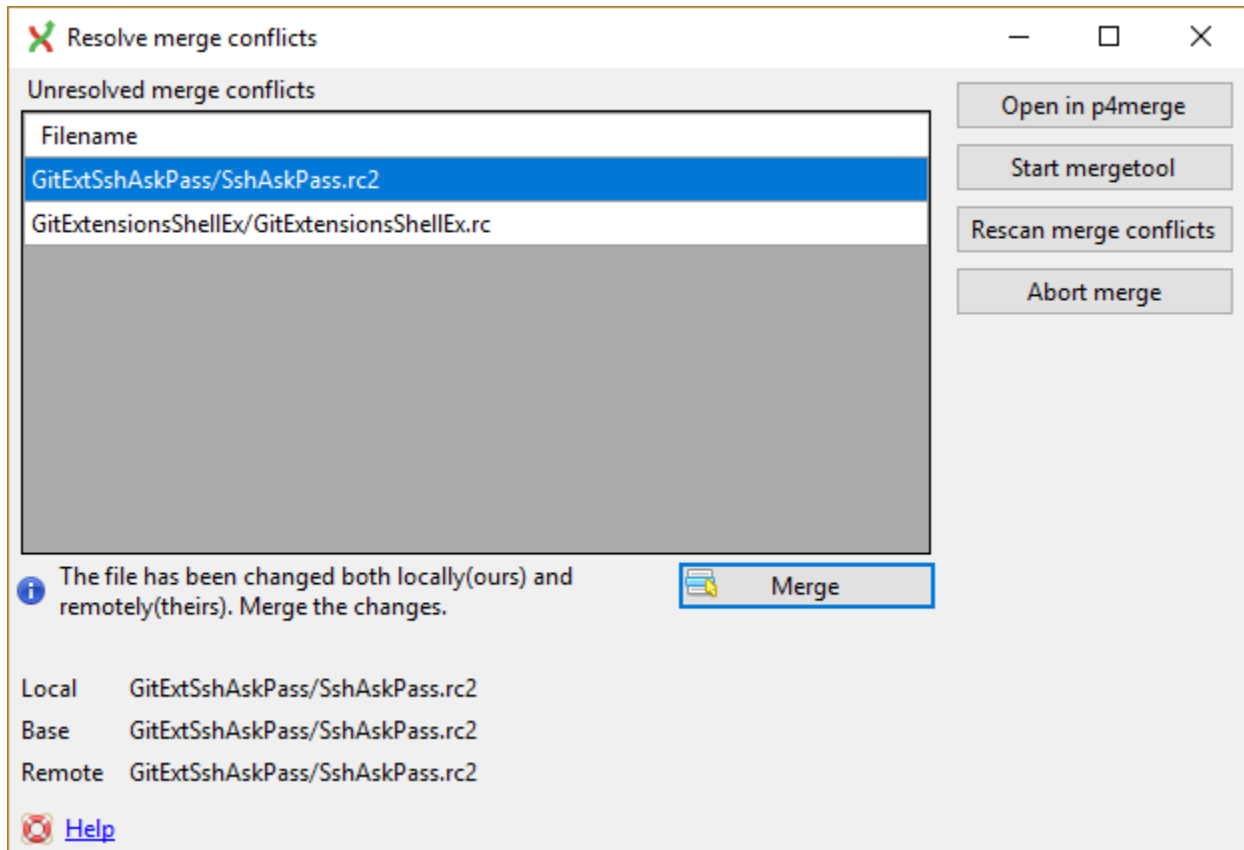
When merging or rebasing branches or commits you can get conflicts. Git will try to resolve these, but some conflicts need to be resolved manually. Git Extensions will show warnings when there is a merge conflict in the status bar in the bottom right corner.



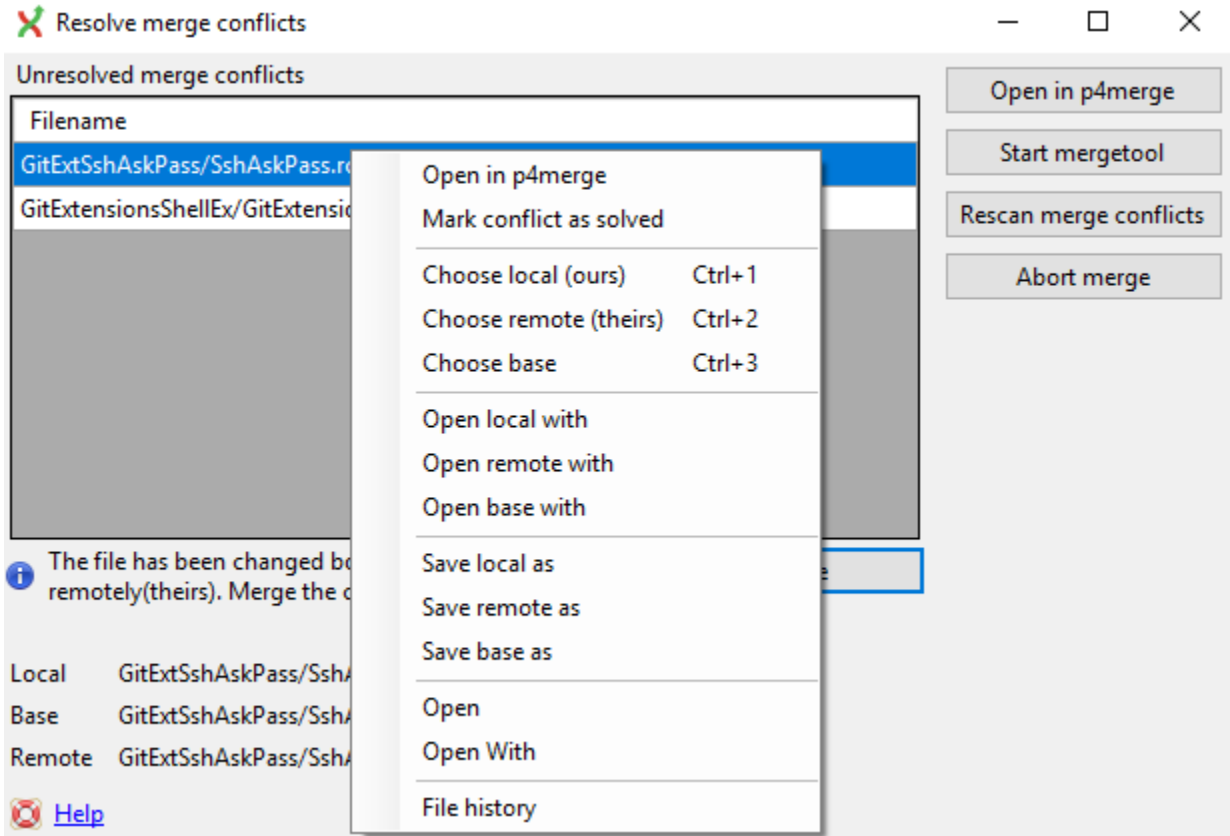
### 12.1 Handle merge conflicts

To solve merge conflicts just click on a warning or open the `Solve merge conflicts...` dialog from the Commands menu. A dialog will prompt showing all conflicts.





The context menu shows the actions to resolve the conflicts. Double-click on a filename will start the mergetool.



There are three kinds of conflicts:

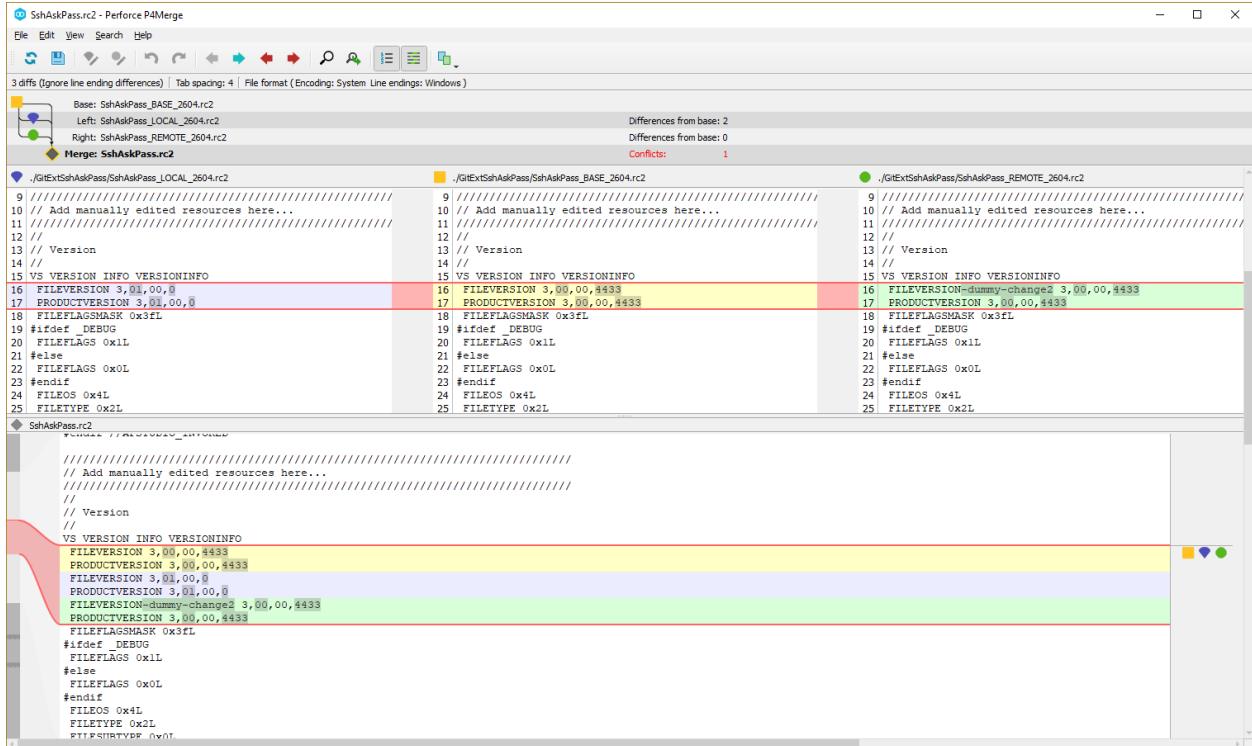
File deleted and changed	Use modified or deleted file?
File deleted and created	Use created or deleted file?
File changed both locally and remotely	Start merge tool.

If the file is deleted in one commit and changed in another commit, a dialog will ask to keep the modified file or delete the file. When there is a conflicting change the merge tool will be started. You can configure the tool you want to use for merge conflicts. The image below shows Perforce P4Merge, a merge tool free to use for small teams.

In the merge tool you will see four versions of the same file:

Base	The latest version of the file that exist in both repositories
Local	The latest local version of the file
Remote	The latest remote version of the file
Merged	The result of the merge

**Caution:** When you are in the middle of a merge the file named local represents your file. When you are in the middle of a rebase the file named remote represents your file. This can be confusing, so double check if you are in doubt.



---

## Modify Git history

---

A Git commit cannot be changed, the sha for the commit will be replaced at all changes. However, the contents of a commit can be modified and committed again as a new commit with a new sha and the branch/tag can be moved to the modified (new) commit.

- A commit can be reverted, the changes of a certain commit can be reverted and added as a new commit. Similar, a commit can be applied again (possibly to a new branch), known as cherry picking.
- The commit can be added again (and all commits that are children to the commit) as new commits and git branches can be made to point to the new commit instead.

There are 2 different cases, and consequently 2 ways to do it with git when we want to modify the history:

- Modify the last commit of the current branch with doing an `amend`
- Modify an older commit with doing an `interactive rebase`

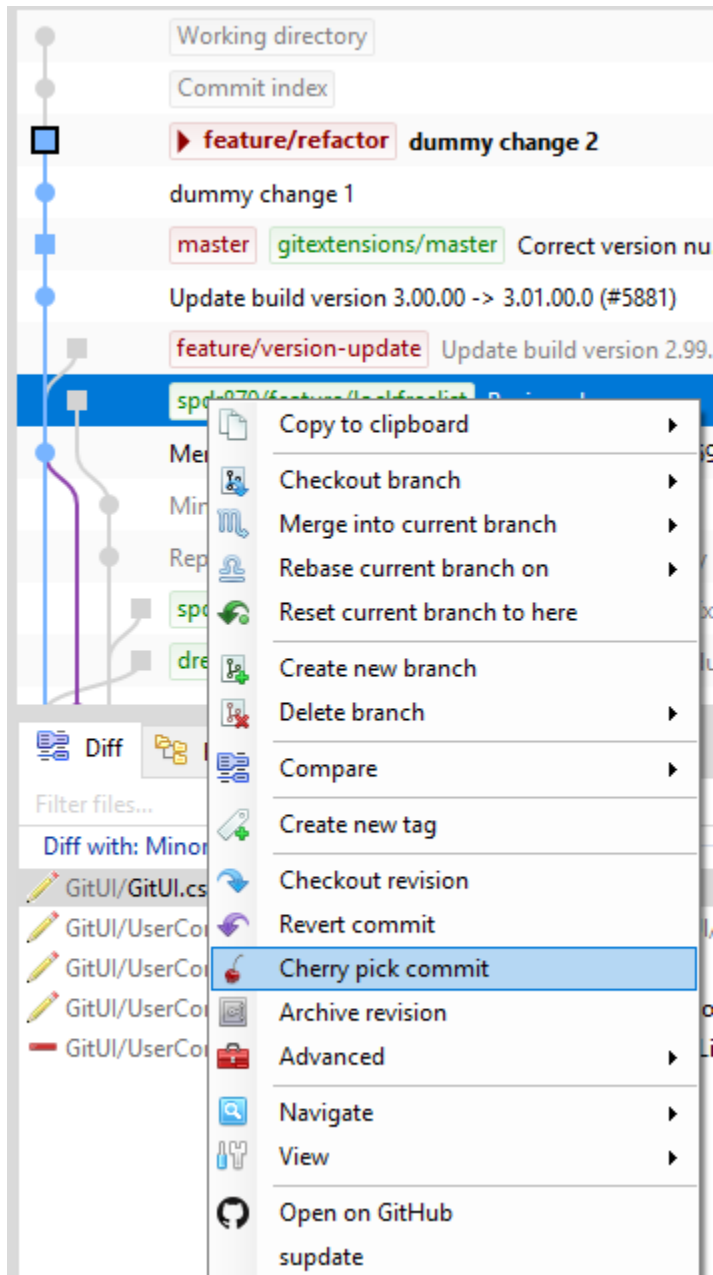
Note: There are 2 things to understand when working with the history with git:

- As git only creates immutable commits (sealed by the sha1), “modifying” a commit is in fact creating a new more or less similar commit.
- Consequently, the entire history of children following the changed commit will be different.

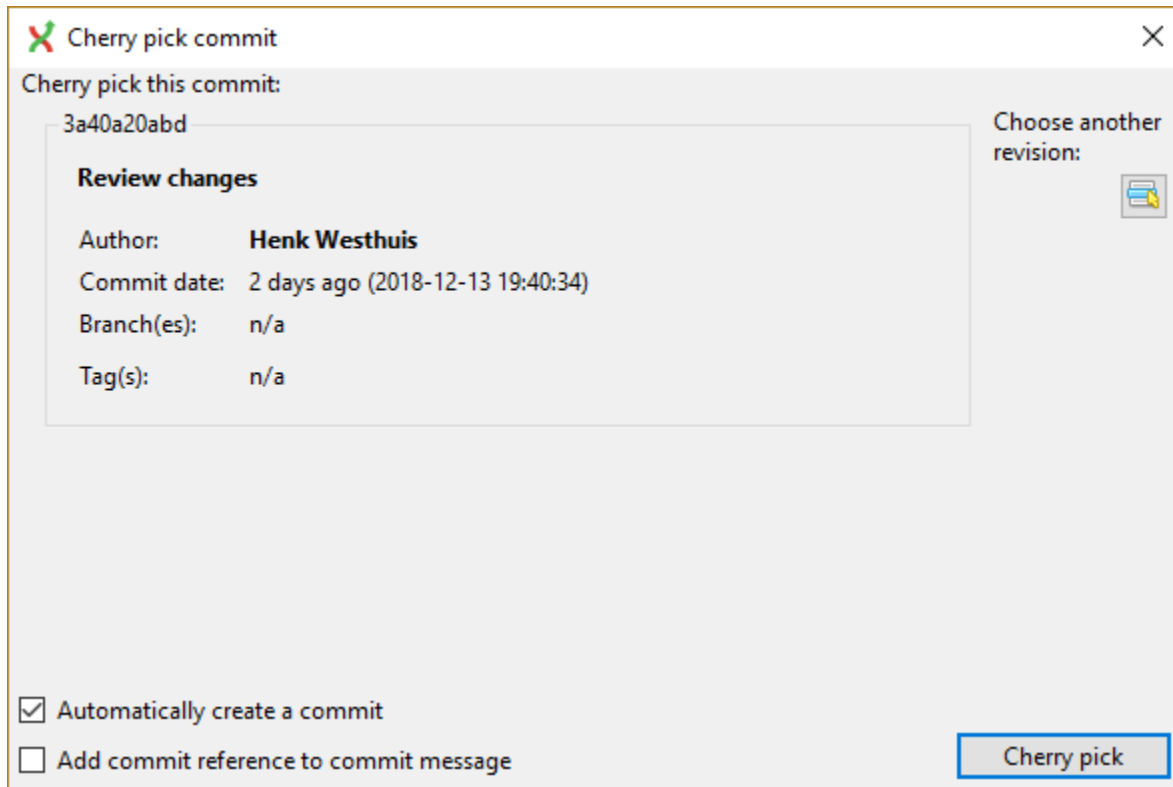
So, except if the history has not been already pushed, or if you have good reasons, it is a bad practice to change the history because you will mess the history of other developers.

### 13.1 Cherry pick commit

A commit can be recommitted by using the cherry pick function. This can be very useful when you want to make the same change on multiple branches. Select the commit (or range of commits) you want to cherry pick:

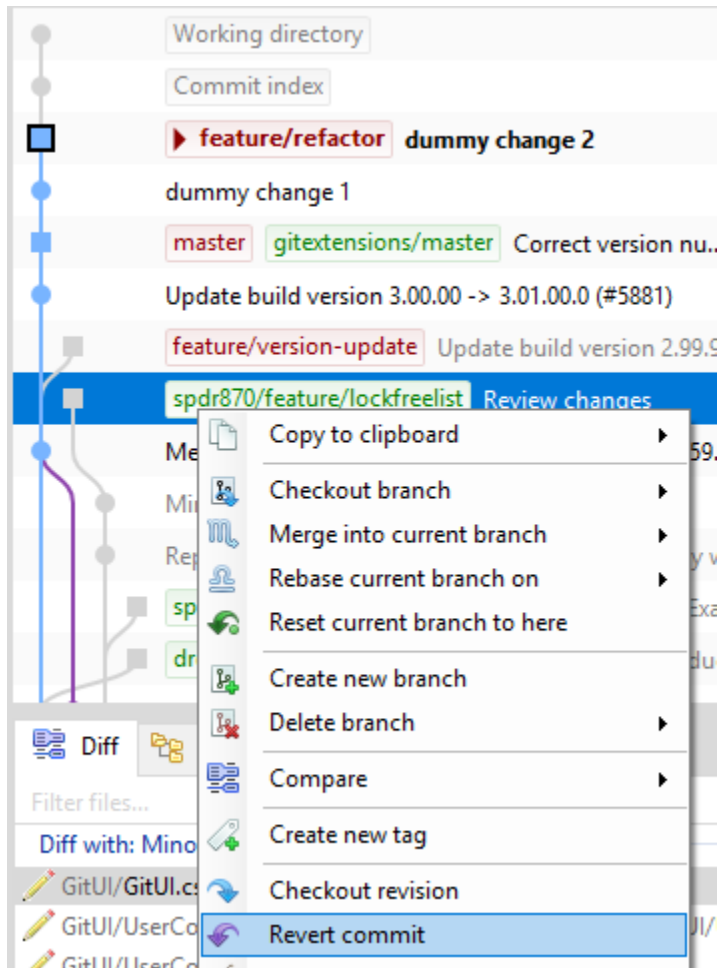


The confirm dialog opens:

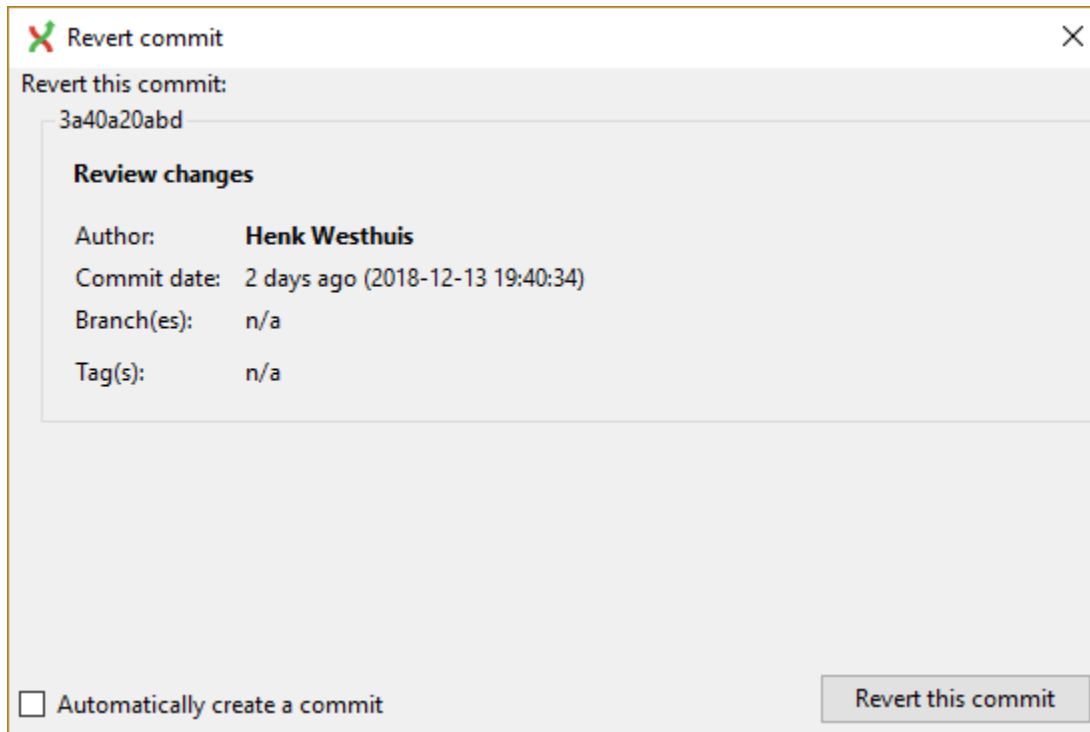


## 13.2 Revert commit

A commit cannot be deleted once it is published. If you need to undo the changes made in a commit, you need to create a new commit that undoes the changes. This is called a revert commit. A revert commit is similar to a cherry pick, but the cherry pick tries to apply the same changes as the original commit, a revert will try to reverse the changes.

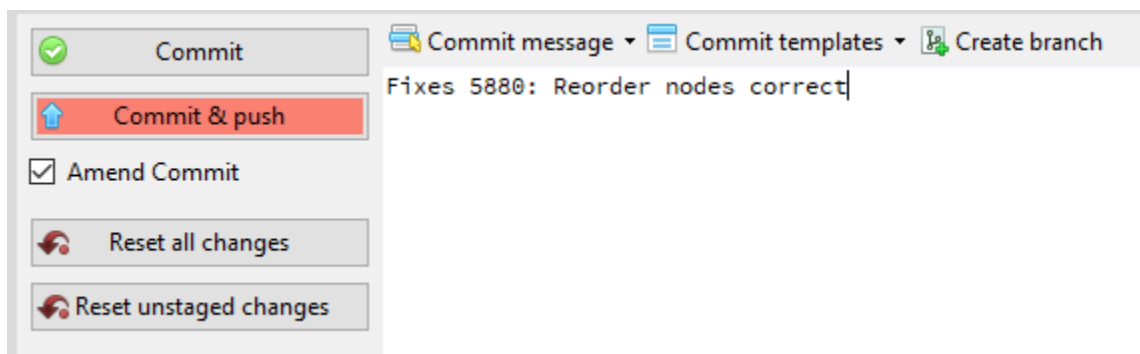


The confirm dialog opens:



### 13.3 Modify the last commit

The easiest way to modify the last commit is to do an `amend` commit. To do that, open the commit windows and check the option “Amend commit”. If the commit message text area was empty, it is now filled with the message of the last commit. You could now just update the commit message and commit or also add some more changes in the staging area to add them to the commit.



### 13.4 Modify an older commit

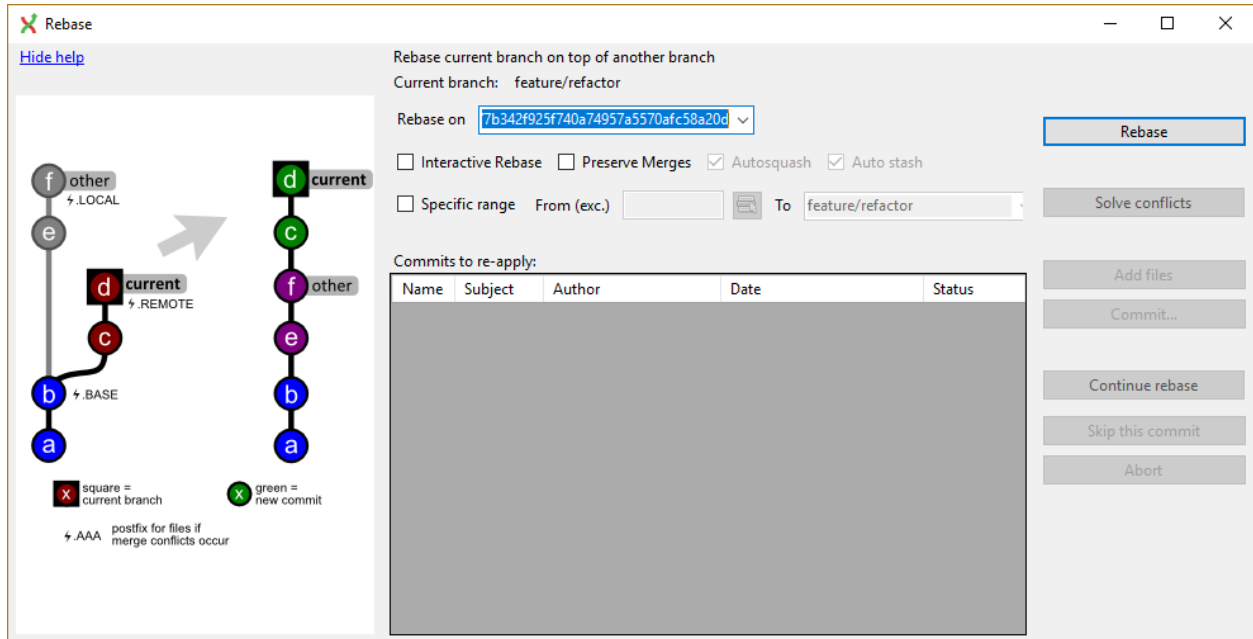
It normally makes sense just to change the history for the current branch. To change the parents of the current branch you will have to make a `rebase`. Git Extensions has functionality that wraps the Git rebase commands and simplifies usage in some situations.



### 13.4.1 Interactive rebase

First, you should create a commit containing the changes you want to add to a previous commit (or know an existing commit that contains this changes).

Then use the *rebase* feature in interactive mode on a base commit older than the one that you want to modify. See [Branches](#) for how to start a rebase, start an interactive rebase from the context menu or by selecting the checkbox in the rebase dialog.



You will be prompted by a text editor displaying all the commits that will be rebased

```

F:/dev/gc/gitextensions/.git/rebase-merge/git-rebase-todo
1 |pick 9d7a081f5 dummy change 1
2 |pick 4feed7716 dummy change 2
3
4 # Rebase 3c2ac977b..4feed7716 onto 3c2ac977b (2 commands)
5 #
6 # Commands:
7 # p, pick <commit> = use commit
8 # r, reword <commit> = use commit, but edit the commit message
9 # e, edit <commit> = use commit, but stop for amending
10 # s, squash <commit> = use commit, but meld into previous commit
11 # f, fixup <commit> = like "squash", but discard this commit's log message
12 # x, exec <command> = run command (the rest of the line) using shell
13 # b, break = stop here (continue rebase later with 'git rebase --continue')
14 # d, drop <commit> = remove commit
15 # l, label <label> = label current HEAD with a name
16 # t, reset <label> = reset HEAD to a label
17 # m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
18 # .      create a merge commit using the original merge commit's
19 # .      message (or the oneline, if no original merge commit was
20 # .      specified). Use -c <commit> to reword the commit message.
21 #
22 # These lines can be re-ordered; they are executed from top to bottom.
23 #
24 # If you remove a line here THAT COMMIT WILL BE LOST.
25 #
26 # However, if you remove everything, the rebase will be aborted.
27 #
28 # Note that empty commits are commented out
29

```

You could have a look to this [\\_documentation](https://git-scm.com/book/en/v2/Git-Tools-Rewriting-History): <https://git-scm.com/book/en/v2/Git-Tools-Rewriting-History> to better understand all the possibilities offered.

The options offered are :

- reorder the lines to reorder the commits,
- delete a line to throw away a commit and the changes introduced by the commit,
- write *r* or *reword* in front of a commit to rewrite the commit message,
- write *f* or *fixup* in front of a commit to meld the commit with the previous commit and with keeping the commit message of the first commit,
- write *s* or *squash* in front of a commit to meld the commit with the previous commit and with rewriting the commit message.

Often, we will use interactive rebase to move the line and squash or fixup commits to modify the history.

Once we did the changes, save and close the editor to let git do the rebase.

### 13.4.2 Using autosquash rebase feature

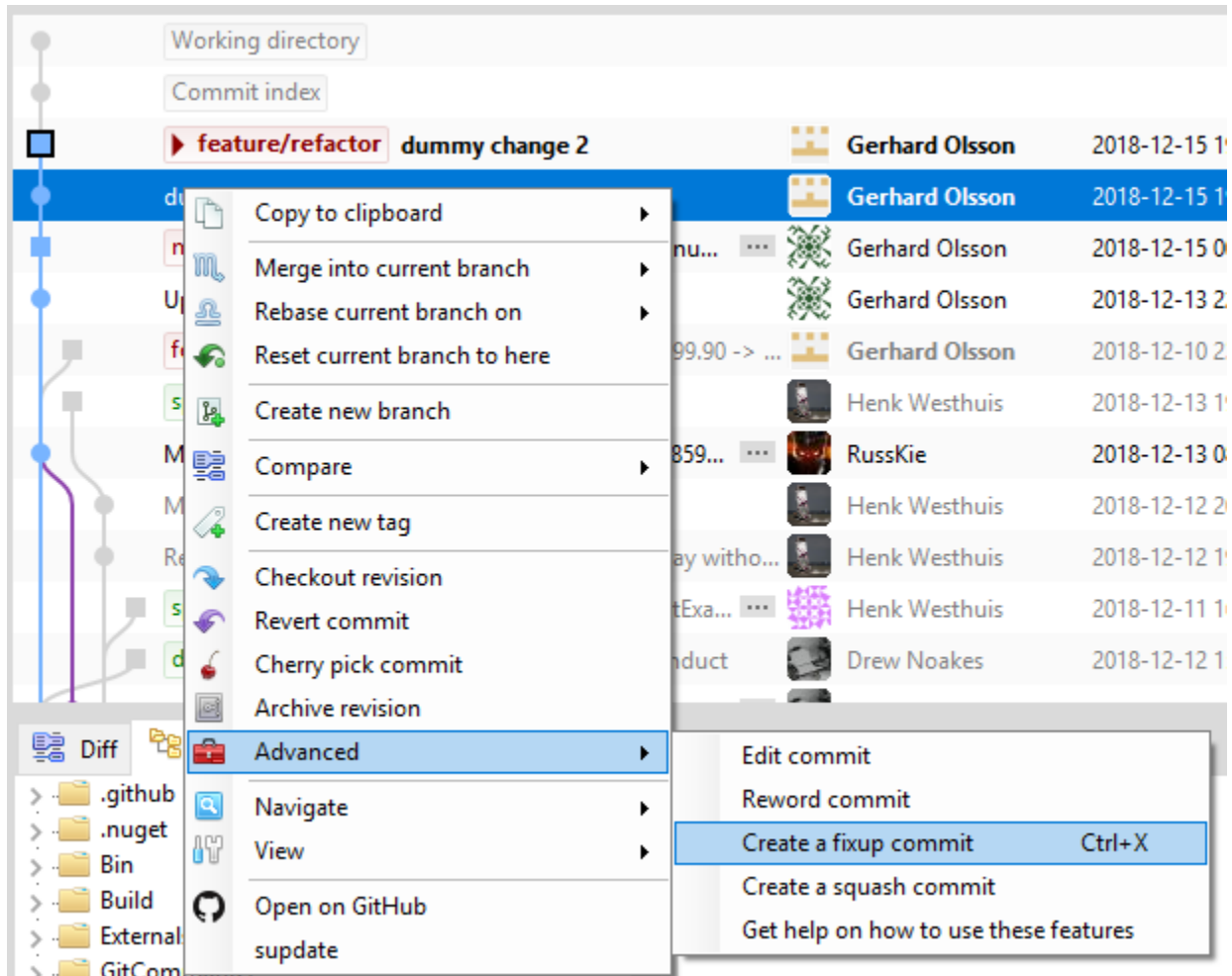
There is an option to facilitate the use of the `interactive rebase` when you know, at the moment of doing a commit that the changes introduced by this commit should have been made in an older commit (the case of a *fixup* or *squash*).

In this case, you should create a commit containing the changes you want to add to a previous commit and use the *Advanced* menu to:

- create a *fixup* commit
- create a *squash* commit

Right click on the commit in the history, you know that you want to “modify”.

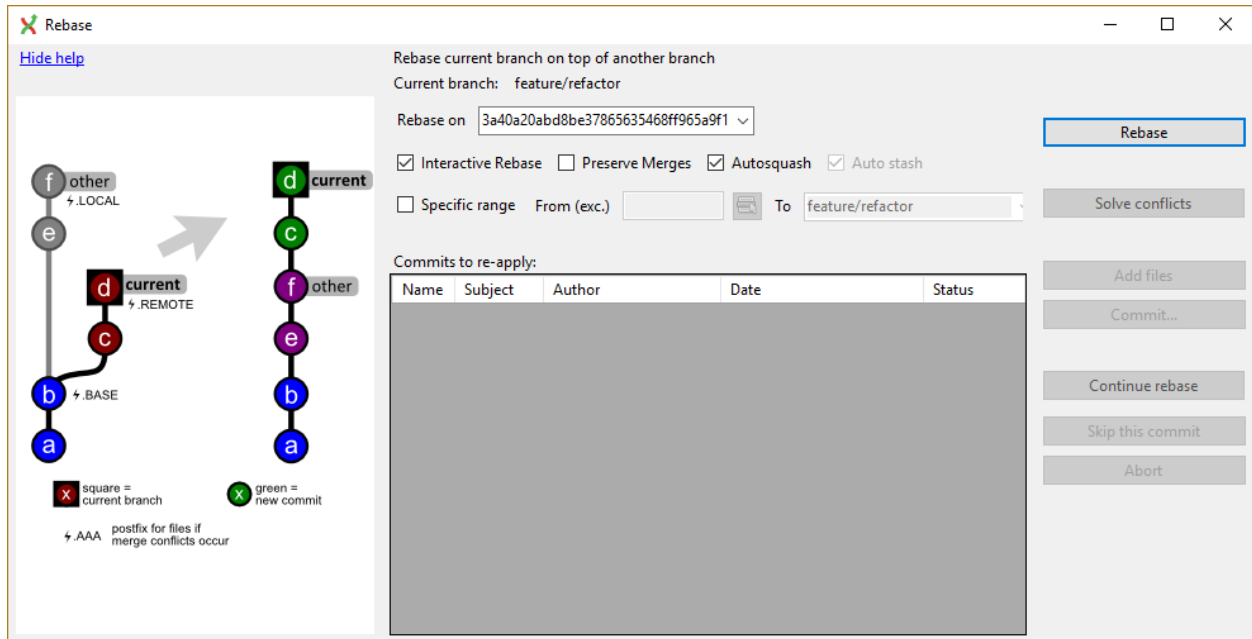
And choose the suitable option...



If you have not the changes prior to open the dialog, do them now.

GitExtensions will open the commit window with an already filled commit message containing the needed information to find the commit to “modify”. Do not change the commit message and commit all the changes needed.

Then process to the interactive rebase, like describe in the previous paragraph but with enabling the option *Autosquash*.



Launch the rebase by clicking on *Rebase*.

The interactive rebase will process the same way but with a major difference! When enabling the *Autosquash* option, git will automatically reorder the commits lines and write the good actions in front of the commits when it will open the text editor. You normally have just to close the editor (except if you want to do additional changes). And let git do the rebase.

### 13.4.3 Edit/reword commit

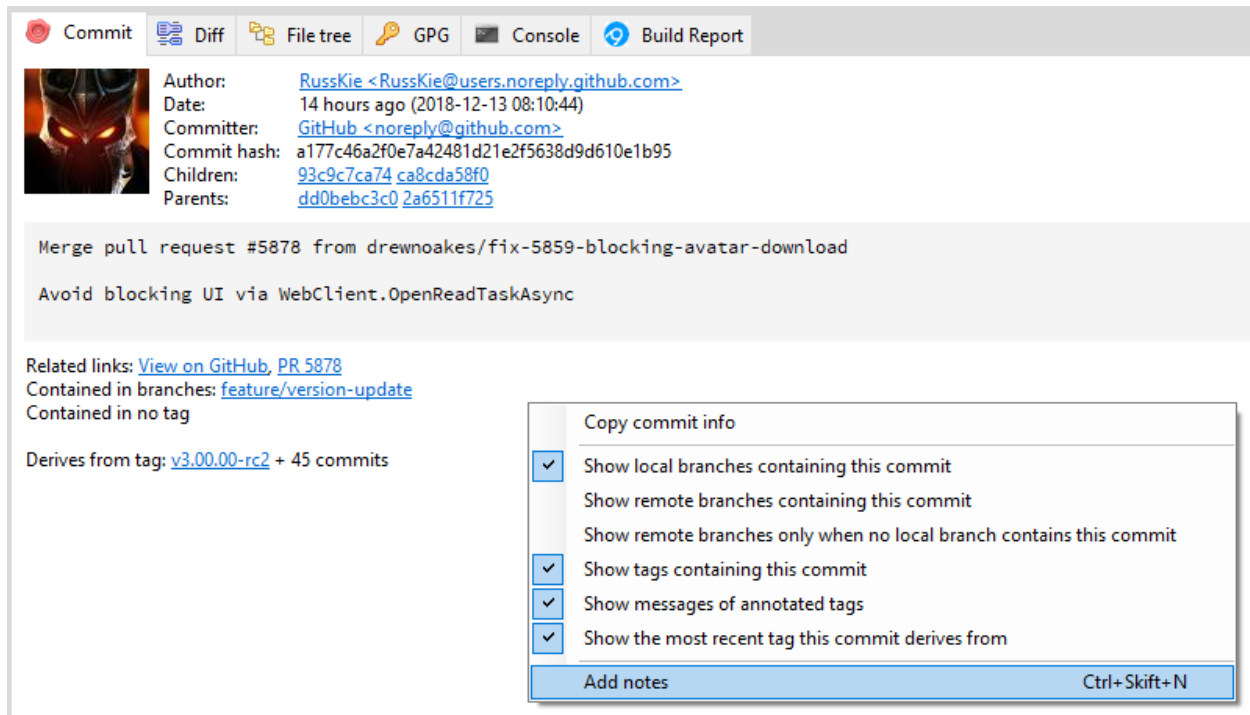
These options are the same as starting an interactive rebase on the parent to the selected commit and doing an `edit` (allow to amend to the commit) or `reword` (editing the commit message) and then run an interactive rebase in the background.

Note especially that this functionality will fail if you try to edit/reword a commit that is not a parent to the current checkout.

# CHAPTER 14

## Notes

Notes can be added to a commit. Notes will be stored separately and will not be pushed. To add a new note choose `add notes` in the context menu of the commit information box.



The screenshot shows a commit information box with a toolbar at the top containing icons for Commit, Diff, File tree, GPG, Console, and Build Report. The commit details include a profile picture of a character with glowing eyes, and the following information:

- Author: [RussKie <RussKie@users.noreply.github.com>](mailto:RussKie@users.noreply.github.com)
- Date: 14 hours ago (2018-12-13 08:10:44)
- Committer: [GitHub <noreply@github.com>](mailto:noreply@github.com)
- Commit hash: `a177c46a2f0e7a42481d21e2f5638d9d610e1b95`
- Children: [93c9c7ca74](#) [ca8cda58f0](#)
- Parents: [dd0bebc3c0](#) [2a6511f725](#)

Below the details, it shows a merge pull request #5878 from `drewnoakes/fix-5859-blocking-avatar-download` with the commit message: `Avoid blocking UI via WebClient.OpenReadTaskAsync`.

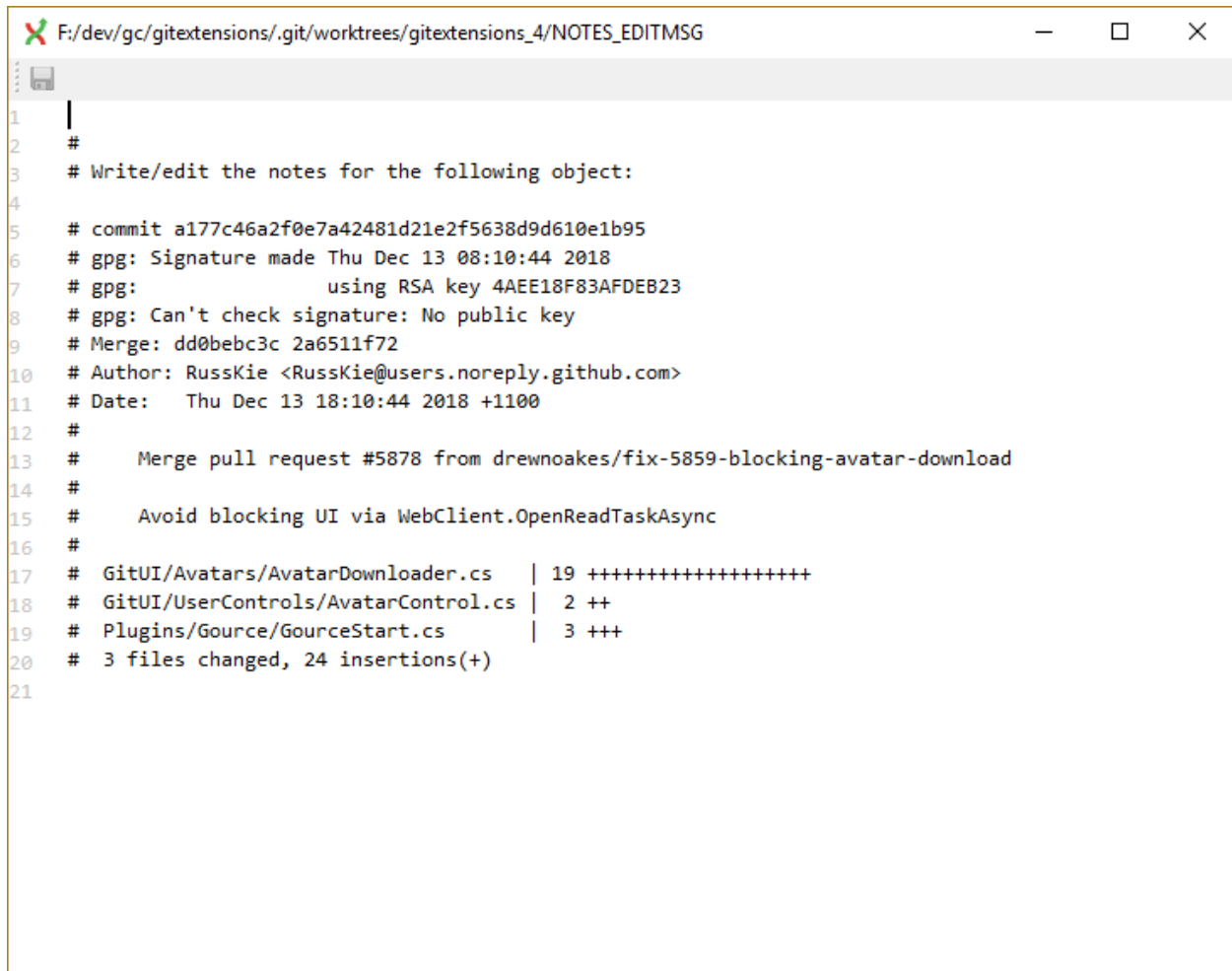
Related links: [View on GitHub, PR 5878](#)  
Contained in branches: [feature/version-update](#)  
Contained in no tag

Derives from tag: [v3.00.00-rc2](#) + 45 commits

The context menu is open, showing the following options:

- Copy commit info
- Show local branches containing this commit
- Show remote branches containing this commit
- Show remote branches only when no local branch contains this commit
- Show tags containing this commit
- Show messages of annotated tags
- Show the most recent tag this commit derives from
- Add notes Ctrl+Shift+N

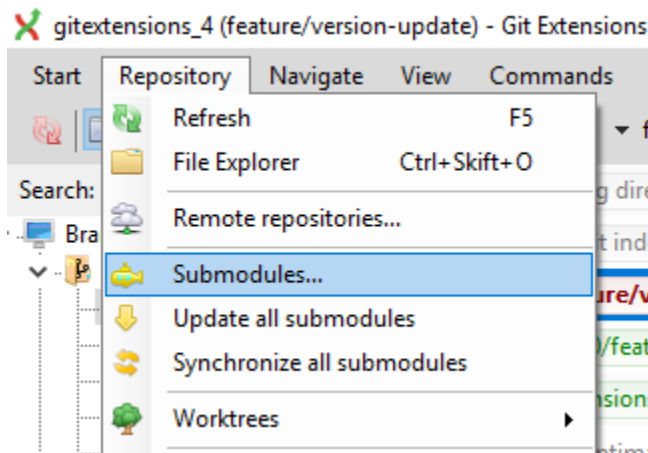
The editor that has been configured in the settings dialog will be used to enter or edit the notes. The Git Extensions editor is advised.



The image shows a Notepad window titled "F:/dev/gc/gitextensions/.git/worktrees/gitextensions\_4/NOTES\_EDITMSG". The window contains a git commit message template with the following text:

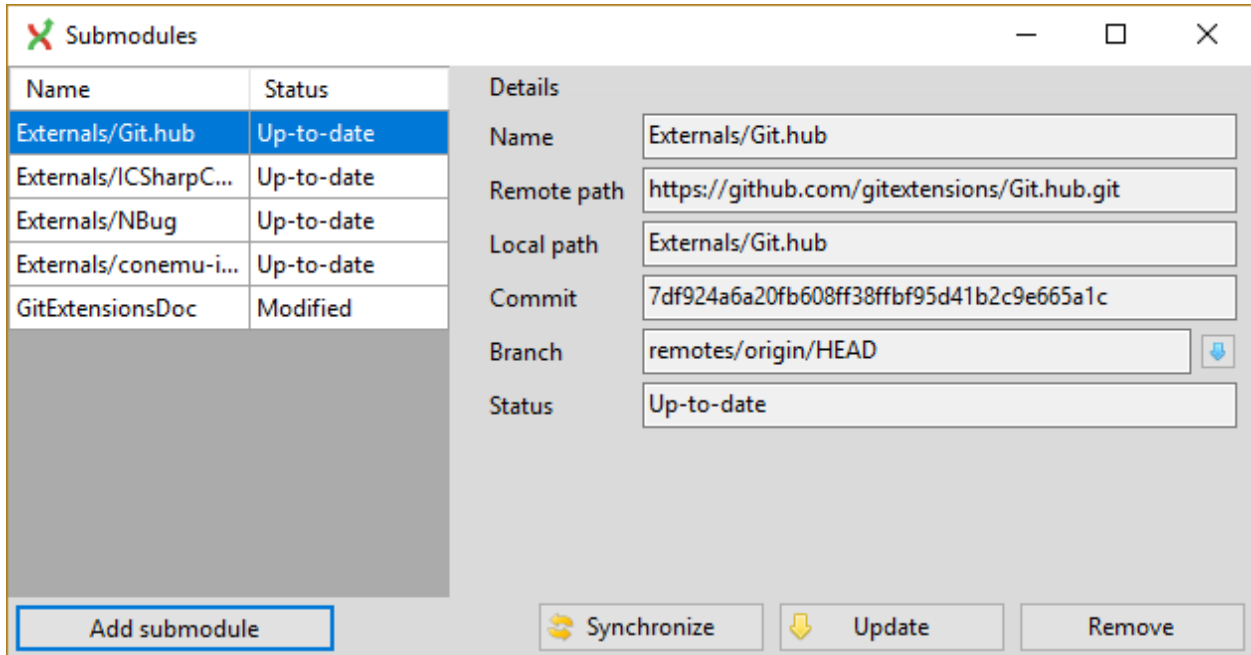
```
1 |
2 #
3 # Write/edit the notes for the following object:
4
5 # commit a177c46a2f0e7a42481d21e2f5638d9d610e1b95
6 # gpg: Signature made Thu Dec 13 08:10:44 2018
7 # gpg:                using RSA key 4AEE18F83AFDEB23
8 # gpg: Can't check signature: No public key
9 # Merge: dd0bebc3c 2a6511f72
10 # Author: RussKie <RussKie@users.noreply.github.com>
11 # Date:   Thu Dec 13 18:10:44 2018 +1100
12 #
13 #   Merge pull request #5878 from drewnoakes/fix-5859-blocking-avatar-download
14 #
15 #   Avoid blocking UI via WebClient.OpenReadTaskAsync
16 #
17 #   GitUI/Avatars/AvatarDownloader.cs | 19 ++++++
18 #   GitUI/UserControls/AvatarControl.cs | 2 ++
19 #   Plugins/Gource/GourceStart.cs     | 3 +++
20 # 3 files changed, 24 insertions(+)
21
```

Large projects can be split into smaller parts using submodules. A submodule contains the name, url and revision of another repository. To create a submodule in an existing git repository you need to add a link to another repository containing the files of the submodule.



### 15.1 Manage submodules

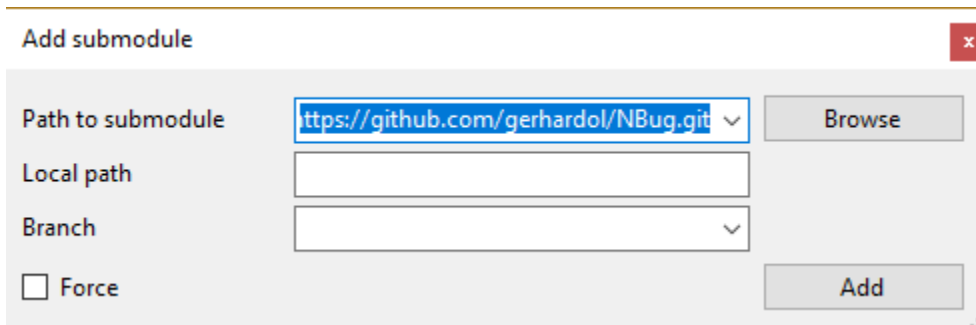
The current state of the submodules can be viewed with the `Manage submodules` function. All submodules are shown in the list on the left.



Add submodule	Add a new submodule to the repository
Synchronize	Synchronizes the remote URL configuration setting to the value specified in <code>.gitmodules</code> for the selected submodule.
Initialize	Initialize the selected submodules, i.e. register each submodule name and url found in <code>.gitmodules</code> into <code>.git/config</code> . The submodule will also be updated.
Update	Update the registered submodules, i.e. clone missing submodules and checkout the commit specified in the index of the containing repository.
Remove	Remove the submodule from the repository

## 15.2 Add submodule

To add a new submodule choose `Add submodule` in the `Manage submodules` dialog.



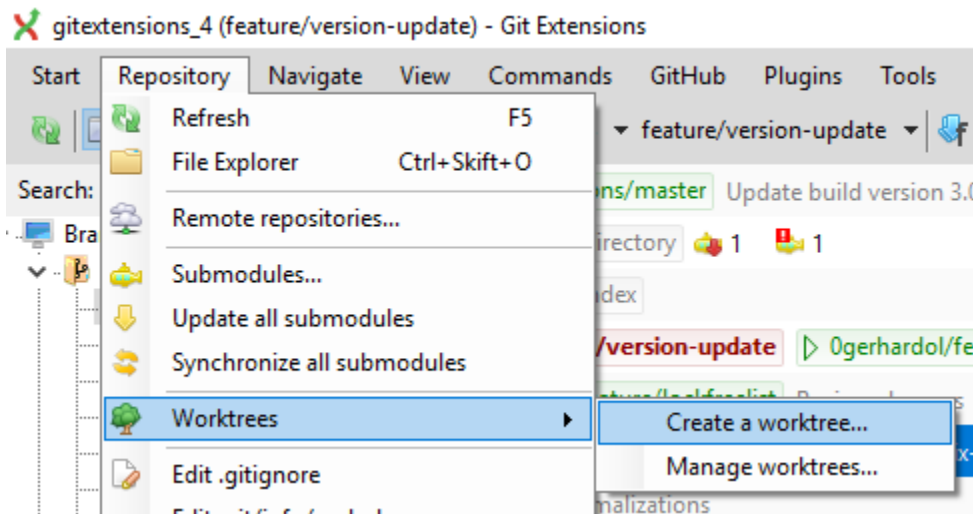
Path to submodule	Path to the remote repository to use as submodule.
Local path	Local path to this submodule, relative to the root of the current repository.
Branch	Branch to track.



# CHAPTER 16

## Worktrees

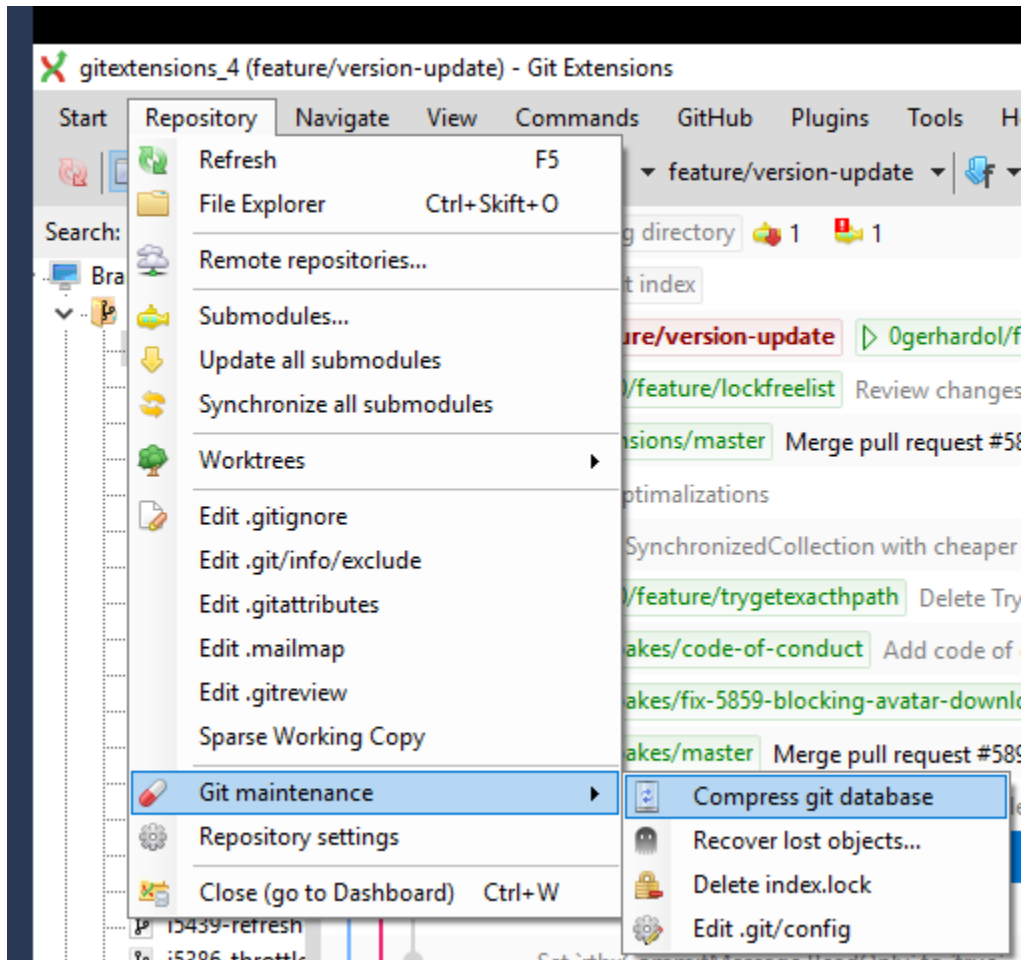
Git Extensions support Git worktrees: Multiple checked out working directories can share local branches. For more information see the Git documentation: <https://git-scm.com/docs/git-worktree>



In this chapter some of the functions to maintain a repository are discussed.

### 17.1 Compress Git database

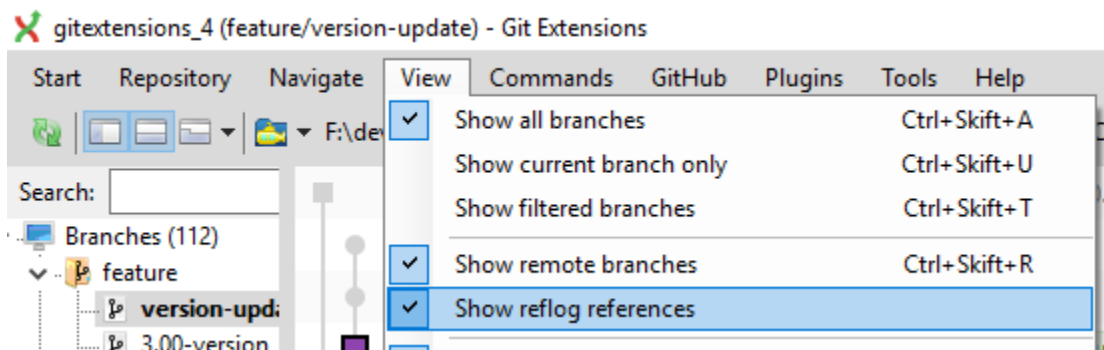
Git will create a lot of files. You can run the `Compress git database` to pack all small files building up a repository into one big file. Git will also garbage collect all unused objects that are older than 15 days. When a database is fragmented into a many small files compressing the database can increase performance.



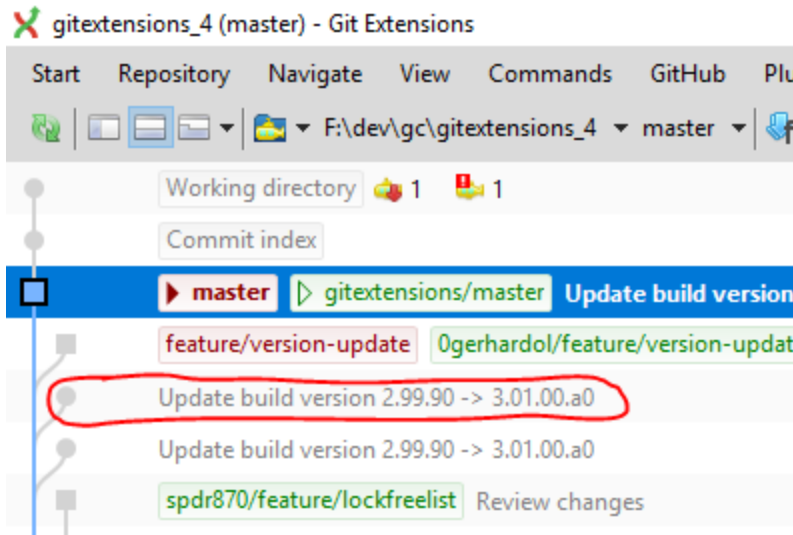
## 17.2 Recover lost objects

Normally Git will not delete files right away when you remove something from your repository. The reason for this is that you can restore deleted items if you need to. Git will delete removed items when they are older than 15 days and you run `Compress git database`.

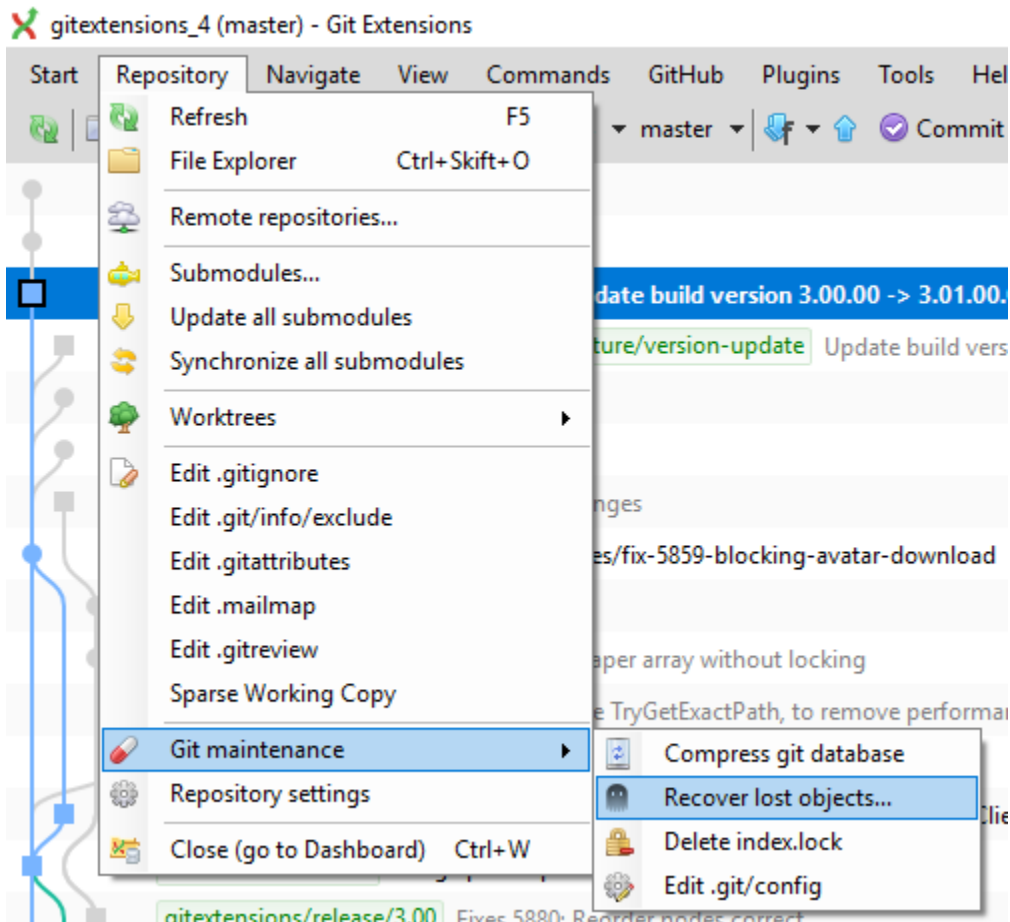
Commits without branches or tags can be shown with Git relog <https://git-scm.com/docs/git-reflog> The easiest way to view the commits is to show Git relog in the revision graph:

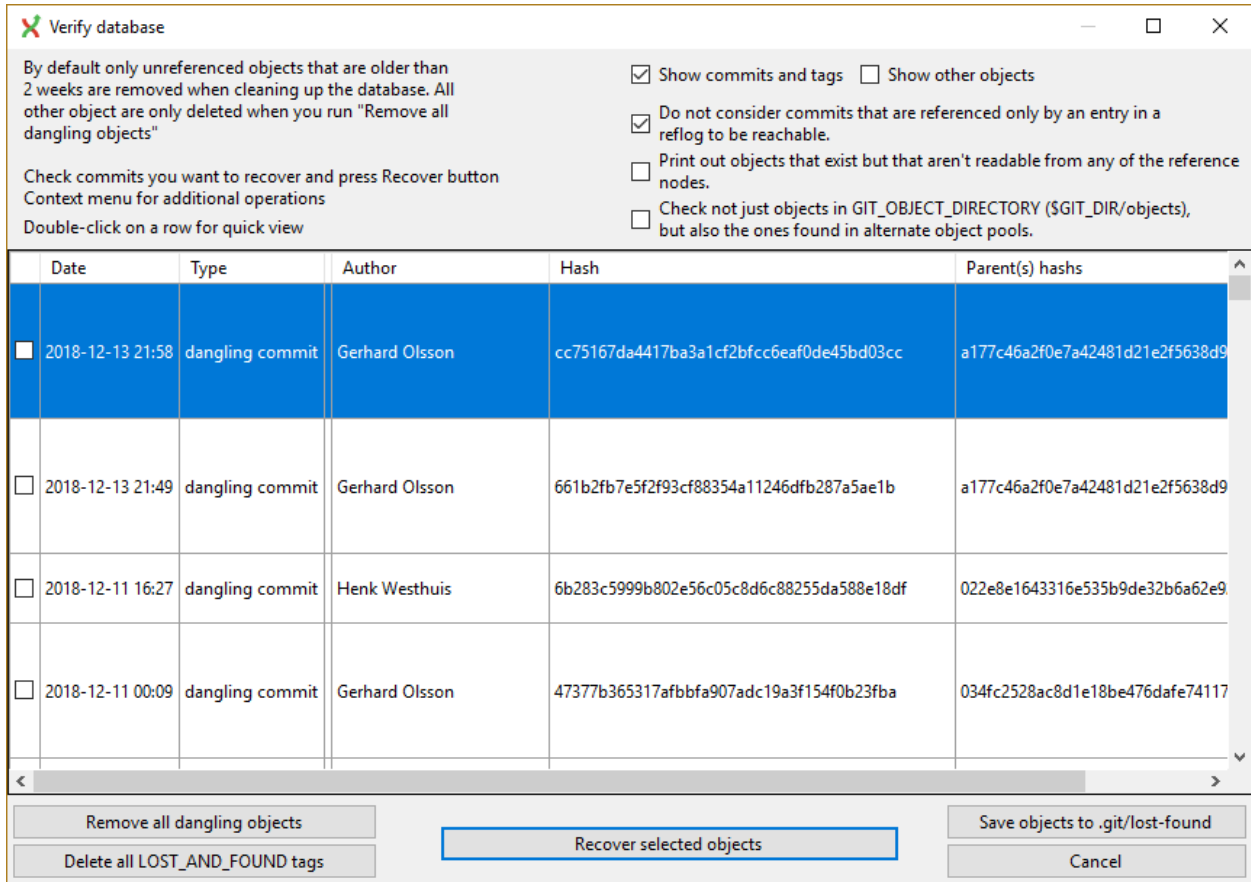


The relog commits are listed as gray:



GE also supports the previous way to show you all dangling objects and will allow you to review and recover them. If you accidentally deleted a commit you can try to recover it using the `Recover lost objects` function.





Git Extensions also is able to tag all lost objects. Doing this will make all lost objects visible again making it very easy to locate the commit(s) you would like to recover.

### 17.3 Fix user names

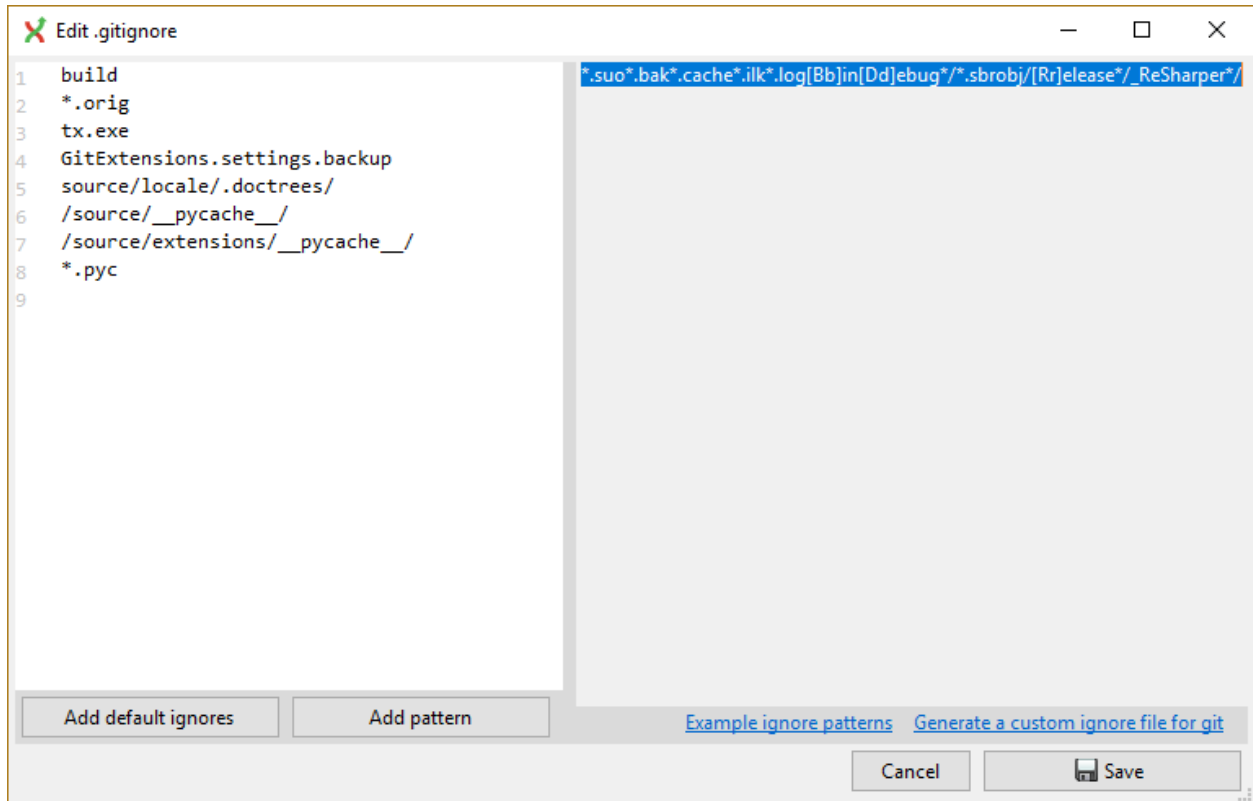
When someone accidentally committed using a wrong username this can be fixed using the Edit `.mailmap` function. Git will use the username for an email address when it is set in the `.mailmap` file.



For more information, see <https://git-scm.com/docs/git-check-mailmap>.

## 17.4 Ignore files

Git will track all files that are in the working directory. Normally you do not want to exclude all files that are created by the compiler. You can add files that should be ignored to the `.gitignore` file. You can use wildcards and regular expressions. All entries are case sensitive. The button `Add default ignores` will add files that should be ignored when using Visual Studio.



A short overview of the syntax:

#	Lines started with # are handled as comments
!	Lines started with ! are exclude patterns
[Dd]	Characters inside [ . . ] means that 1 of the characters must match
*	Wildcard
/	A leading slash matches the beginning of the pathname; for example, <code>/*.c</code> matches <code>cat-file.c</code> but not <code>mozilla-sha1/sha1.c</code>
/	If the pattern ends with a slash, it is removed for the purpose of the following description, but it would only find a match with a directory. In other words, <code>foo/</code> will match a directory <code>foo</code> and paths underneath it, but will not match a regular file or a symbolic link <code>foo</code> (this is consistent with the way how <code>pathspecc</code> works in general in git).

For more [detailed information](#).

## 18.1 Change language

In the settings dialog the language can be chosen.



## 18.2 Translate Git Extensions

More information in the Git Extensions wiki: <https://github.com/gitextensions/gitextensions/wiki/Translations>

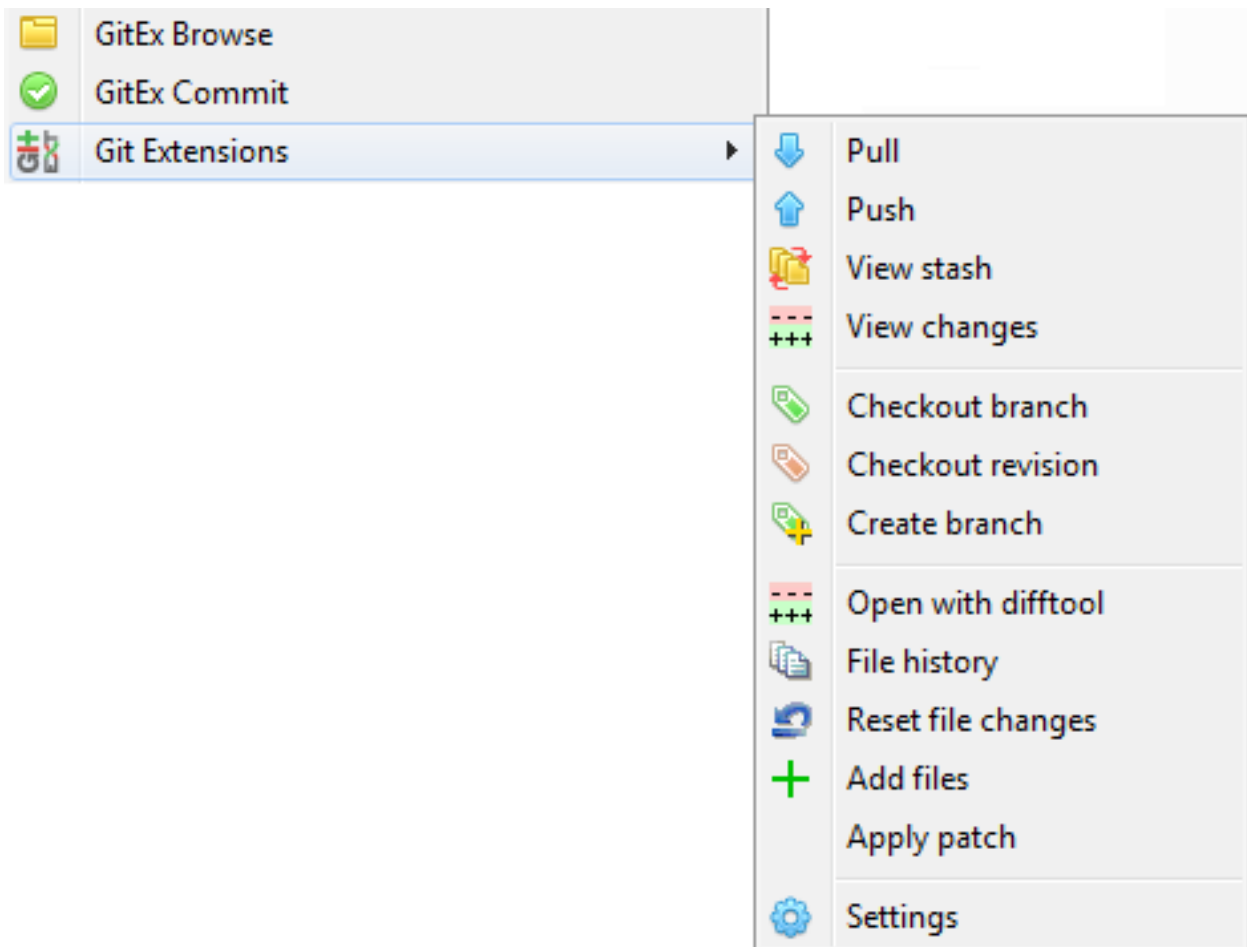
Translations are done on Transifex: <https://www.transifex.com/git-extensions/git-extensions/>



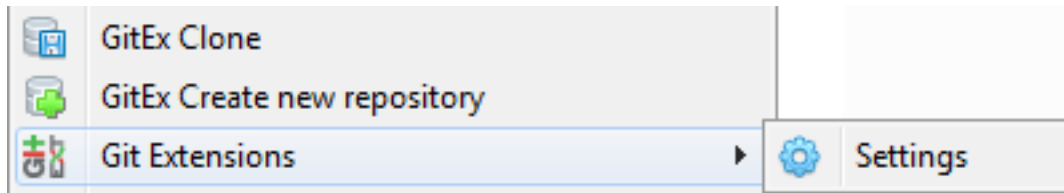
## CHAPTER 19

### Windows Explorer

The common commands can be started from Windows Explorer using the shell extensions. This option is only available when Shell Extensions are installed.

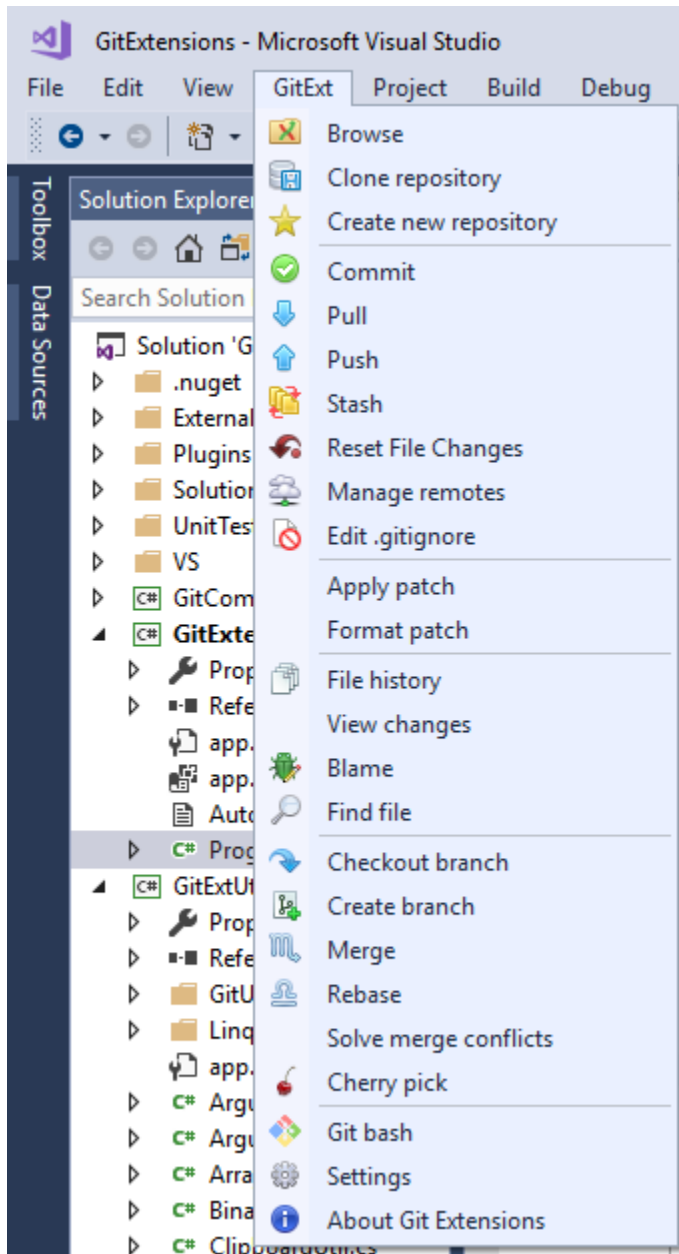


If the folder do not have a Git repository, you can clone.



### 20.1 Menu

Almost all function can be started from the `GitExt` menu in Visual Studio.

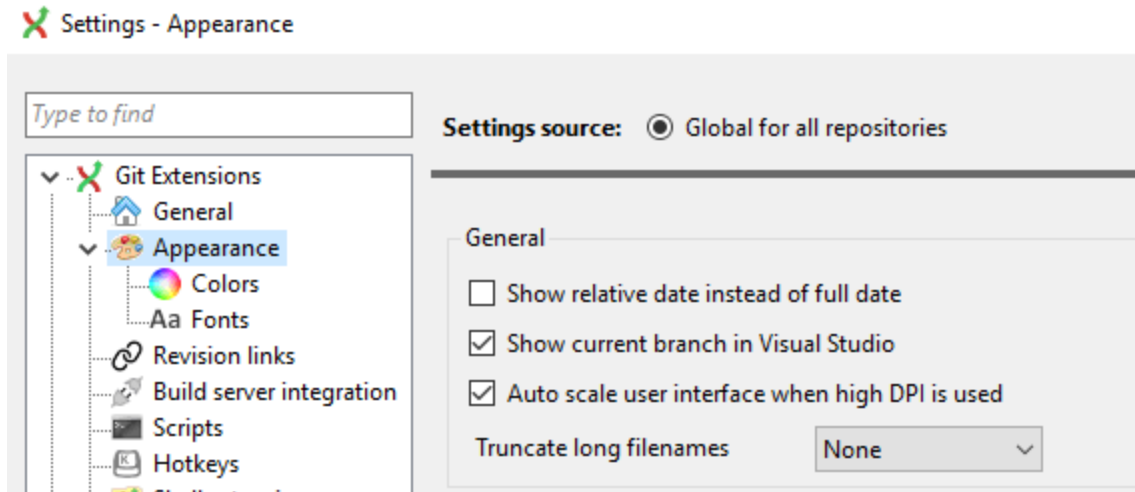


## 20.2 Toolbar

A Git Extensions toolbar allows you to perform the most common actions. The buttons can be customized, same functions as in the menu.



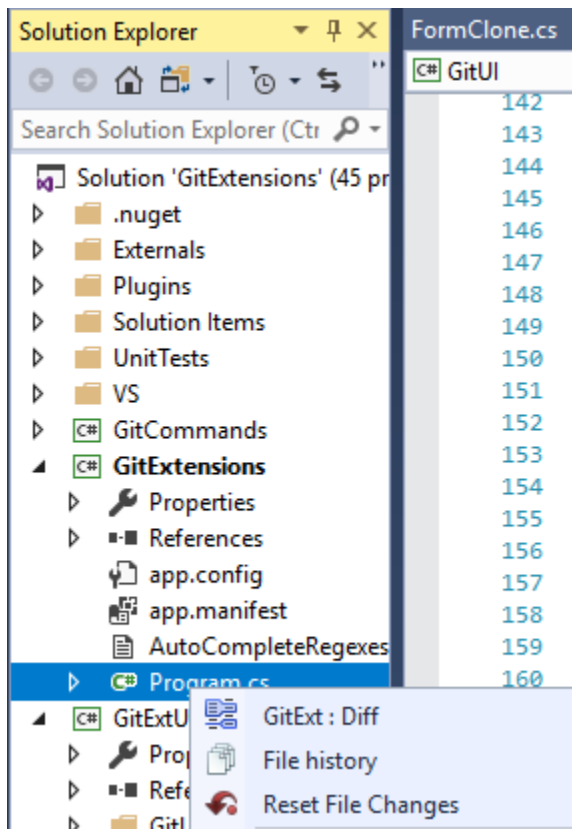
The current branch name can be shown in the commit button.



## 20.3 Context menu

Options in the context menu on files and in Solution Explorer:

- Diff changes to the commit index
- View the file history by choosing the 'File history' option.
- Reset the file changes to the last committed revision.



### 21.1 Git Extensions command line

Most features can be started from the command line. It is recommended to add `gitex.cmd` to the path when using from the command line. It is typically stored in the `C:\Program Files (x86)\GitExtensions` folder.



```
Commandline usage

Supported commandline arguments for
gitex.cmd / gitex (located in the same folder as GitExtensions.exe):

browse [path] [-filter=] [-commit=]
about
add [filename]
addfiles [filename]
apply [filename]
applypatch [filename]
blame filename
branch
checkout
checkoutbranch
checkoutrevision
cherry
cleanup
clone [path]
commit [--quiet]
difftool filename
filehistory filename
fileeditor filename
formatpatch
gitbash
gitignore
help (shows this dialog)
init [path]
merge [--branch name]
mergeconflicts [--quiet]
mergetool [--quiet]
openrepo [path] [-filter=]
pull [--rebase] [--merge] [--fetch] [--quiet] [--remotebranch name]
push [--quiet]
rebase [--branch name]
remotes
reset
revert filename
searchfile
settings
stash
synchronize [--rebase] [--merge] [--fetch] [--quiet]
tag
viewdiff
viewpatch [filename]
```



A screenshot of a terminal window titled "MINGW64:/f/dev/gc/gitextensions\_4". The window contains the following text:

```
ejgo@ejgo3 MINGW64 /f/dev/gc/gitextensions_4 (oriash93/_5109)  
$
```





## CHAPTER 22

## Appendix

## 22.1 Git Cheat Sheet

Action	Command
Create new repository	<code>\$ git init</code>
Create shared repository	<code>\$ git init --bare --shared=all</code>
Clone repository	<code>\$ git clone c:/demo1 c:/demo2</code>
Checkout branch	<code>\$ git checkout &lt;name&gt;</code>
Create branch	<code>\$ git branch &lt;name&gt;</code>
Delete branch	<code>\$ git branch -d &lt;name&gt;</code>
Merge branch (from the branch to merge into):	<code>\$ git merge PDC</code>
Solve conflicts (add <code>-tool=kdiff3</code> if no mergetool is specified)	<code>\$ git mergetool \$ git commit</code>
Create tag	<code>\$ git tag &lt;name&gt;</code>
Add files/changes (. for all files)	<code>\$ git add .</code>
Commit added files/changes ( <code>-amend</code> to amend to last commit)	<code>\$ git commit -m "Enter commit message"</code>
Discard changes	<code>\$ git reset --hard</code>
Create patch ( <code>-M</code> = detect renames <code>-C</code> = detect copies)	<code>\$ git format-patch -M -C origin</code>
Apply patch without merging	<code>\$ git apply c:/patch/01-emp.patch</code>
Merge patch	<code>\$ git am --3way --signoff c:/patch/01-emp.patch</code>
Solve conflicts (add <code>-tool=kdiff3</code> if no mergetool is specified)	<code>\$ git mergetool</code> <code>\$ git am --3way --resolved</code>
Stash changes	<code>\$ git stash</code>
Apply stashed changes	<code>\$ git stash apply</code>
Pull changes (add <code>-rebase</code> to rebase instead of merge)	<code>\$ git pull c:/demo1 master</code>
Solve conflicts (add <code>-tool=kdiff3</code> if no mergetool is specified)	<code>\$ git mergetool</code> <code>\$ git commit</code>
<b>22.1. Git Cheat Sheet</b>	<b>110</b>
Push changes (in branch <code>\$ git push c:/demo1 master master:&lt;new&gt;</code> )	<code>\$ git push c:/demo1</code>
Blame	<code>\$ git blame -M -w &lt;filename&gt;</code>
Help	<code>\$ git &lt;command&gt; -help</code>

Here are some default names used by Git.

Default names	
master	default branch
origin	default upstream repository
HEAD	current branch
HEAD^	parent of HEAD
HEAD~4	the great-great grandparent of HEAD

Git Extensions has a possibility to add functionality in external plugins. Some are distributed with the main program. Most plugins has settings in *Settings*. Most plugins also have UI forms accessible from the main menu in *Browse Repository*.

This list is incomplete.

### 23.1 Auto compile SubModules

This plugin proposes (confirmation required) that you automatically build submodules after they are updated via the GitExtensions Update submodules command.

**Enabled**

Enter true to enable the plugin, or false to disable.

**Path to msbuild.exe**

Enter the path to the msbuild.exe executable.

**msbuild.exe arguments**

Enter any arguments to msbuild.

### 23.2 Bitbucket Server

For repositories is hosted on Atlassian Bitbucket Server, the plugin cannot be used for bitbucket.org. For more information see: <https://www.atlassian.com/software/bitbucket/server>

This plugin will enable you to view and create pull requests for Bitbucket.

**Bitbucket Username**

The username required to access Bitbucket.

**Bitbucket Password**

The password required to access Bitbucket.

**Specify the base URL to Bitbucket**

The URL from which you will access Bitbucket.

**Disable SSL verification**

Check this option if you do not require SSL verification to access Bitbucket Server.

## 23.3 Create local tracking branches

This plugin will create local tracking branches for all branches on a remote repository. The remote repository is specified when the plugin is run.

## 23.4 Delete obsolete branches

This plugin allows you to delete obsolete branches i.e. those branches that are fully merged to another branch. It will display a list of obsolete branches for review before deletion.

**Delete obsolete branches older than (days)**

Select branches created greater than the specified number of days ago.

**Branch where all branches should be merged**

The name of the branch where a branch must have been merged into to be considered obsolete.

## 23.5 Find large files

Finds large files in the repository and allows you to delete them.

**Find large files bigger than (Mb)**

Specify what size is considered a 'large' file.

## 23.6 Gerrit Code Review

The Gerrit plugin provides integration with Gerrit for GitExtensions. This plugin has been based on the git-review tool.

For more information see: <https://www.gerritcodereview.com/>

## 23.7 GitHub

This plugin will create an OAuth token so that some common GitHub actions can be integrated with Git Extensions.

For more information see: <https://github.com/>

**OAuth Token**

The token generated and retrieved from GitHub.

## 23.8 GitFlow

This plugin permit to manage your \_branching model: <http://nvie.com/posts/a-successful-git-branching-model/> with \_GitFlow: <https://github.com/nvie/gitflow> in GitExtension

You should have GitFlow installed to use this plugin.

The GitFlow plugin permit to : - init gitflow in your git repository - create your feature, hotfix, release or support branch - manage (pull, publish or finish) your existing gitflow branches

## 23.9 Gource

Gource is a software version control visualization tool.

For more information see: <http://gource.io/>

### **Path to "gource"**

Enter the path to the gource software.

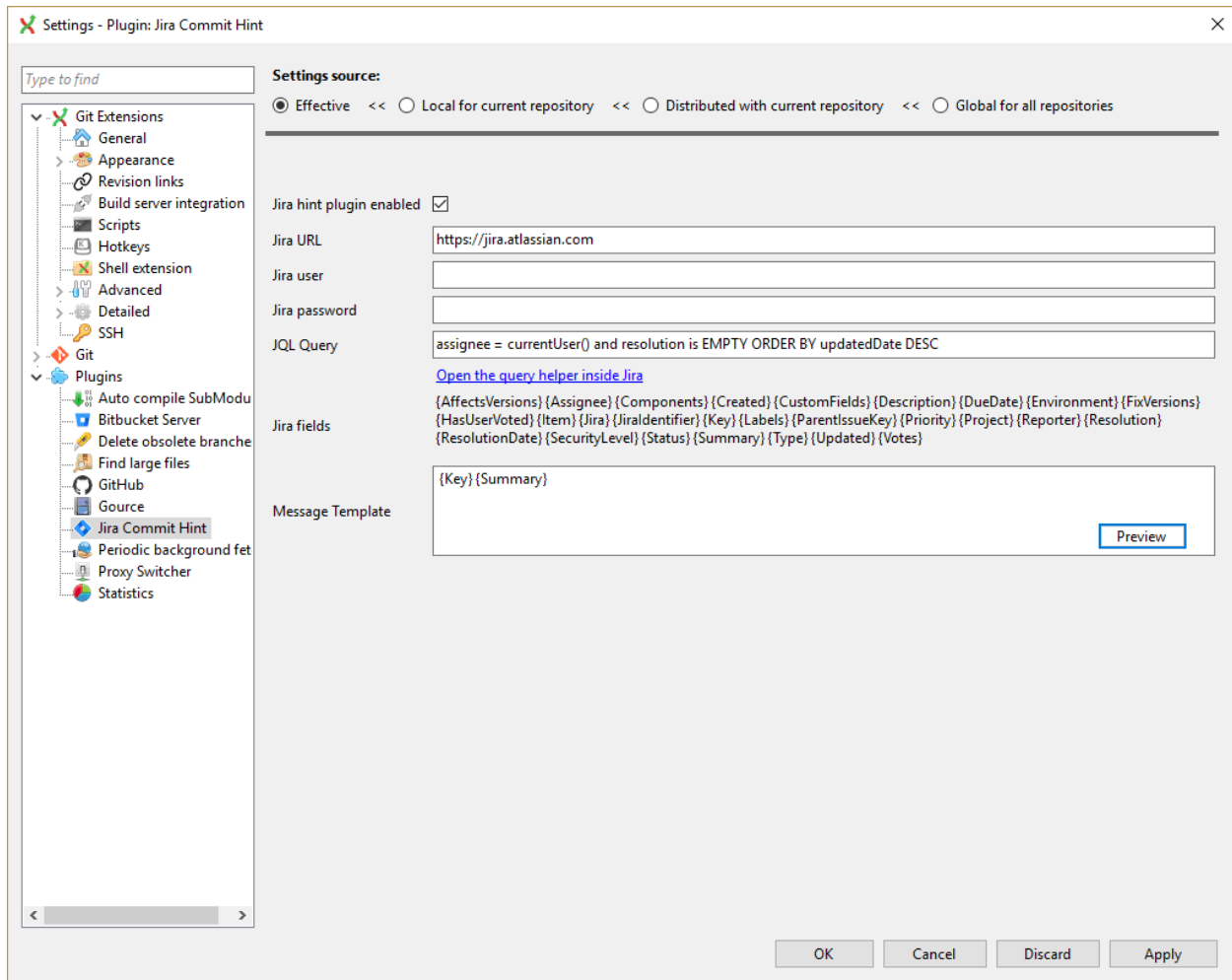
### **Arguments**

Enter any arguments to gource.

## 23.10 Impact Graph

This plugin shows in a graphical format the number of commits and counts of changed lines in the repository performed by each person who has committed a change.

## 23.11 Jira Commit Hint



Provides hints for Atlassian Jira issues in the commit form. For example, you can configure Key - Summary message for all your in progress tasks.

### **Jira hint plugin enabled**

Whether plugin enabled or not.

### **Jira URL**

Link to your Jira server.

### **Jira user**

Your username.

### **Jira password**

Your password.

### **JQL Query**

Query to Jira, results of which you want to show in “Commit Templates” in Commit Form. For more information see: <https://confluence.atlassian.com/jiracoreserver073/advanced-searching-861257209.html>

### **Jira fields**

Key words that you can use in Message Template.

**Message Template**

Result format to insert into message text box after some line from “Commit Templates” selected.

## 23.12 Periodic background fetch

This plugin keeps your remote tracking branches up-to-date automatically by fetching periodically.

**Arguments of git command to run**

Enter the git command and its arguments into the edit box. The default command is `fetch --all`, which will fetch all branches from all remotes. You can modify the command if you would prefer, for example, to fetch only a specific remote, e.g. `fetch upstream`.

**Fetch every (seconds)**

Enter the number of seconds to wait between each fetch. Enter 0 to disable this plugin.

**Refresh view after fetch**

If checked, the commit log and branch labels will be refreshed after the fetch. If you are browsing the commit log and comparing revisions you may wish to disable the refresh to avoid unexpected changes to the commit log.

**Fetch all submodules**

If checked, also perform `git fetch --all` recursively on all configured submodules as part of the periodic background fetch.

## 23.13 Proxy Switcher

This plugin can set/unset the value for the `http.proxy` git config file key as per the settings entered here.

**Username**

The user name needed to access the proxy.

**Password**

The password attached to the username.

**HttpProxy**

Proxy Server URL.

**HttpProxyPort**

Proxy Server port number.

## 23.14 Release Notes Generator

This plugin will generate ‘release notes’. This involves summarising all commits between the specified from and to commit expressions when the plugin is started. This output can be copied to the clipboard in various formats.

## 23.15 Statistics

This plugin provides various statistics (and a pie chart) about the current Git repository. For example, number of commits by author, lines of code per language.

**Code files**

Specifies extensions of files that are considered code files.



**Directories to ignore (EndsWith)**

Ignore these directories when calculating statistics.

**Ignore submodules**

Ignore submodules when calculating statistics (true/false).