
# SUBVERSION & TORTOISESVN GUIDE

VERSION: 1.2

DATE: DECEMBER 7, 2017

CONFIDENTIAL

# CONTENTS

# HISTORY OF CHANGE

| Section | Description | Version |
|---|---|---|
| All | [Add] Initial Version | 1.0 |
| All | [Modify] Peer Review | 1.1 |
| All | [Modify] Final Draft | 1.2 |
| | | |

# INTRODUCTION

This documentation includes information about working with both Subversion and TortoiseSVN.

## SUBVERSION OVERVIEW

Subversion (SVN) is a free, open-source version control system that facilitates document control for multiple users. SVN helps to solve issues regularly encountered with multi-user projects, such as resolving conflicting versions of the same document. A tree of files sits in a central repository (repo) that functions much like an ordinary file server, except it tracks every change made to the files and directories within. Subversion ensures multiple users can access and work on the same file simultaneously without accidentally overwriting others' edits.

TortoiseSVN is a Windows shell extension that allows access to SVN repositories within Windows Explorer.

## WHY USE SUBVERSION?

It is easy to accidentally overwrite changes if multiple users are working on the same file. Accidental overwrites of Darwin Information Typing Architecture or Document Information Typing Architecture (DITA) files results in confusion and rework.

By using a Subversion repository, we can ensure multiple users can access and work on the same file simultaneously without accidentally overwriting the file.

## WHAT THIS GUIDE COVERS

This guide provides general version control concepts and an overview of how Subversion works.

For more detailed information, refer to the free eBook *Version Control with Subversion*. This is found at: http://svnbook.red-bean.com/en/1.7/svn-book.pdf

To understand the basic concepts in depth, review the following sections:

- Chapter 1. Fundamental Concepts
- Chapter 2. Basic Usage

## REPOSITORIES

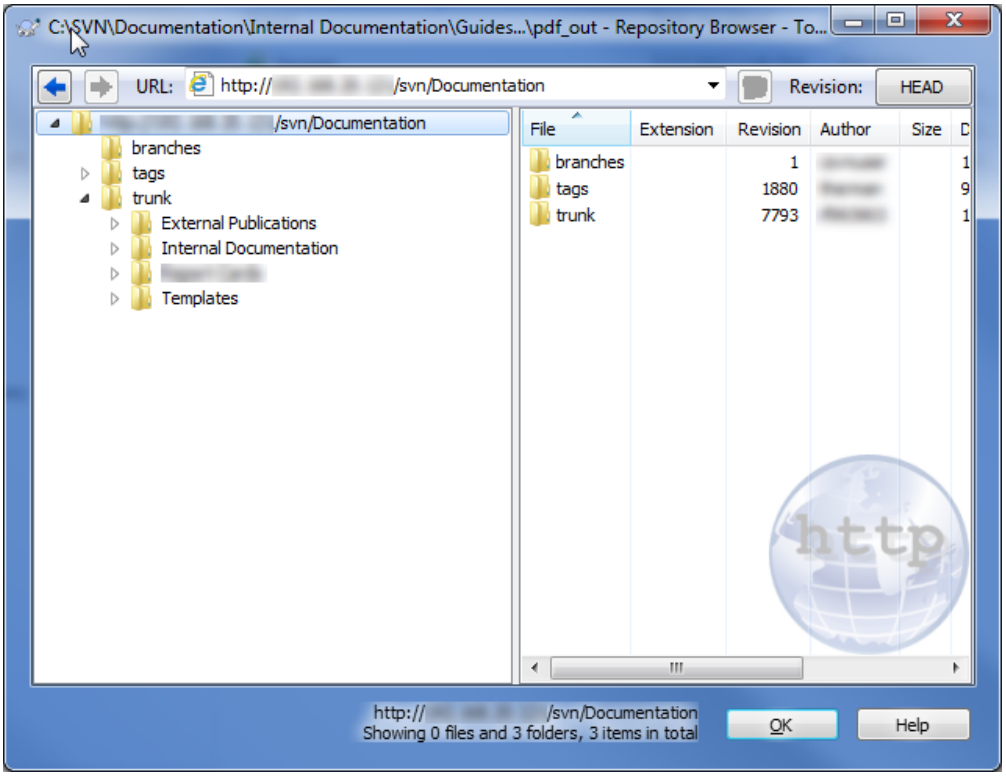
Subversion is a centralized system for version control. The heart of Subversion is the repository, a central data store. The repository stores information in a hierarchy of files and directories called a

file system tree. One or more users can connect to the repository and read or write to the files within.

Subversion differs from a standard file server because it records every change that occurs within the repository. Not only is every change to every individual file stored but also changes to the directory tree, such as the addition, deletion, and rearrangement of files and directories.

The standard view of the repository displays the latest version of the file system tree. However, previous states of the file system are also viewable, including who made changes to files and directories, and what those changes were.

**File System Tree**




## VERSIONING

There are different models of versioning used by different version control programs and file repository systems.

### LOCK-MODIFY-UNLOCK

Many version control systems use a lock-modify-unlock model. In these systems, the repository allows only a single user to modify the file at a time.

### EXAMPLE: LOCK-MODIFY-UNLOCK

Consider the modification of a History of Change DITA file. Every member of the team needs to update this file with the changes they have made.

In a lock-modify-unlock model, the first person to access the file must lock the file before making changes. By locking the file, other users cannot make any changes to it. All other users can read the file but must wait for the first user to finish making their changes and to unlock the file. After the first user unlocks the file, other users can take their turns editing the file.

## COPY-MODIFY-MERGE

Subversion can use a copy-modify-merge model instead of locking files. In this pattern, each user accesses the project repository and creates a personal working copy, which is a local reflection of the files and directories in the repository. Users can work in parallel, modifying their private copies. Finally, the private copies are merged on the server into a new, final version. Subversion assists with merging but ultimately a human being is responsible for making it happen correctly.

### EXAMPLE: COPY-MODIFY-MERGE

Consider the modification of a History of Change for republication.

User 1 and User 2 each create working copies of the release notes. They work at the same time to independently revise the History of Change file within their local copies. User 1 saves their changes to the repository first.

When User 2 attempts to save their changes, the repository informs them their History of Change file is out of date. This means the file in the repository is different since they last copied it. As a result, User 2 must merge any new changes from the repository into their working copy of the History of Change file. Once User 2 has integrated both sets of changes, they save their working copy back to the server.

If the changes made by User 1 overlap those made by User 2, this is a conflict. When User 2 merges the latest repository changes into their working copy, their file is currently in a state of conflict. User 2 sees both sets of conflicting changes and must manually choose between them. Note that Subversion cannot automatically resolve conflicts. Only humans can understand and make the necessary intelligent choices. Once User 2 has resolved the overlapping changes, after a discussion with User 1, they can safely save the merged file back to the repository.

Note: For more information on conflicts and resolving them, refer to **Conflicts**.

In these cases, communication is essential. If your changes conflict with changes made by another team member, you should immediately sit down together to resolve the conflict. Do not overwrite changes made by another team member without discussing it first.

## UNDERSTANDING SVN FOLDERS

A visible feature of TortoiseSVN is the icon overlays that appear on files and folders in your working copy. These show you at a glance which files and folders have been modified. Since TortoiseSVN uses a background caching process to gather the current status, it may take a few seconds before the overlay updates.

Note: TortoiseSVN and Subversion documentation refer to some icon overlays using different names. In these cases, the TortoiseSVN name is first, followed by the Subversion name.

## FOLDER ICONS

Table 1 Status Icon Overlays

| Status | Icon | Description |
|---|---|---|
| **Normal or In Subversion** |  | The Normal or In Subversion overlay represents the latest version from the repository |
| **Modified** |  | The Modified overlay represents:<br>• Modifications done locally to be committed to the repository.<br>• Modifications at the repository level that require you to update your local copy. |
| **Conflicted** |  | The Conflicted overlay represents the conflicted state. This occurs when a commit or update results in conflicts between your local changes and changes from the repository. It also shows the obstructed state, which can occur when an operation cannot finish. |
| **Read Only or Needs Lock** |  | The Read Only or Needs Lock overlay shows if a file contains the Needs Lock property. Subversion makes that file read-only until you get a lock on the file. |
| **Deleted** |  | The Deleted overlay represents the deleted state, where an item is scheduled for deletion |
| **Added** |  | The Added overlay represents the added state and tells you an item can be added to the repository |
| **Ignored** |  | The Ignored overlay represents an item in the ignored state, either due to a global ignore pattern, or the svn:ignore property of the parent folder. This overlay is optional. |
| **Non-Versioned or Unversioned** |  | The Non-Versioned or Unversioned overlay represents an item in the unversioned state. This is an item in a versioned folder, but which is not under version control itself. This overlay is optional. |

# SETTING UP TORTOISESVN

TortoiseSVN is a Windows shell extension that allows access to SVN repositories within Windows Explorer.

## INSTALLING TORTOISESVN

To set up TortoiseSVN:

1. Open https://tortoisesvn.net/downloads.html in your browser.
2. Click the correct download link for your system.
   Our current systems use the 64-bit version of Windows.

Downloading the incorrect version results in the installation process stopping unexpectedly.

3. Select Save File on the dialog window.
   The file is saved to your Windows Downloads folder.
4. Navigate to the Windows Downloads folder once the download completes.
5. Double-click the MSI installer file.
6. Follow the Setup Wizard instructions.
7. Restart your computer once the installation completes.

## SPECIFYING THE LOCATION FOR THE LOCAL REPOSITORY

To specify the local repository for TortoiseSVN:

1. Create a new folder on your C:\ drive called SVN.
2. Create a subfolder within the new SVN folder called Documentation.
   This is where you keep local copies of the repository.

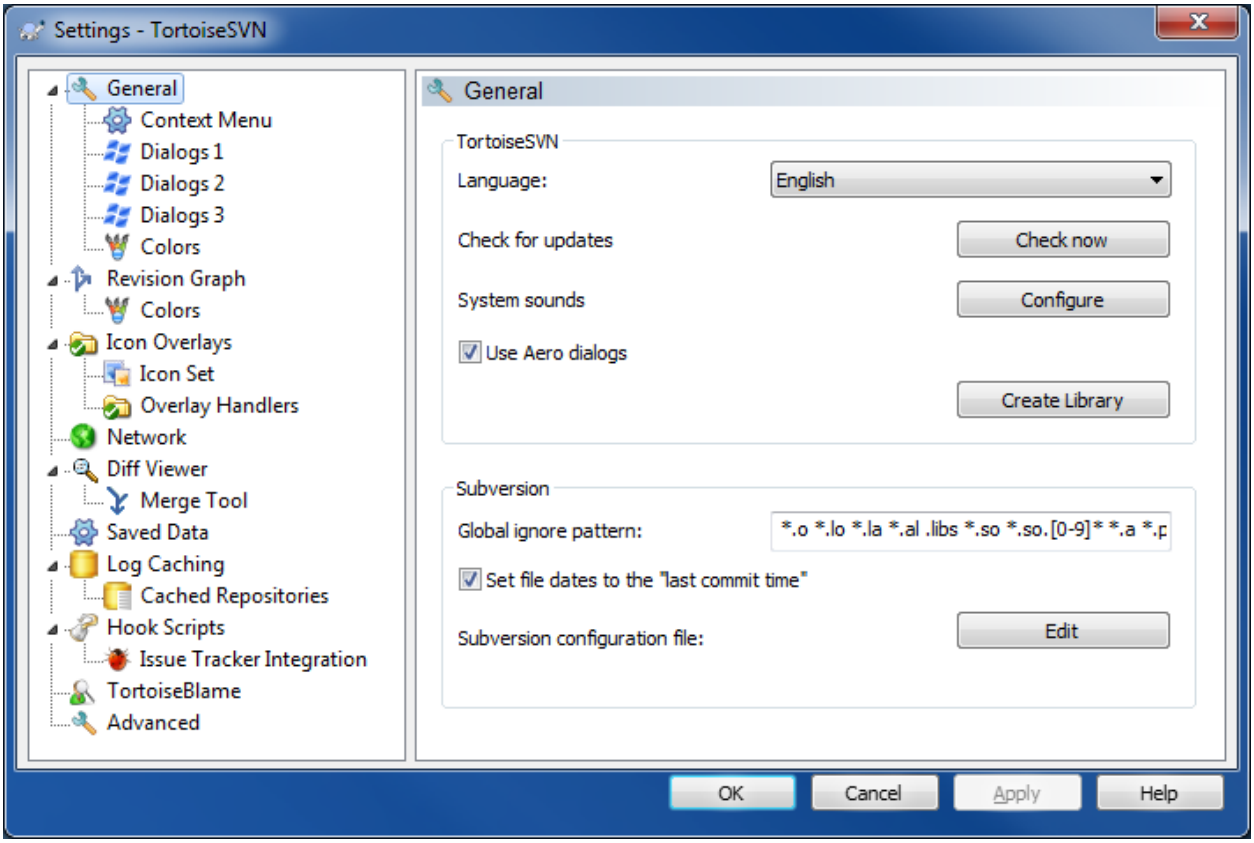
## CONFIGURING TORTOISESVN

When working with SVN, you do not want to upload all file types to Subversion. To prevent certain file types from being uploaded, you need to add file types to the global-ignore list. To do this:

1. Right-click anywhere in Windows Explorer to bring up the TortoiseSVN context menu.
2. Select TortoiseSVN > Settings.
3. In the Subversion section of the General tab, enter the following text at the end of the default Global Ignore Pattern field: *.~psfldr *.log *.tps *.backup.book *.backup.fm *.recover.fm *.xml.bak *.bak *.fm *.rtf
4. Click OK.

Note: This ignore pattern uses the asterisk as a wildcard character. TorotoiseSVN ignores the file name and checks only the file extension.

**Settings Window**




## CHECKING OUT THE REPOSITORY

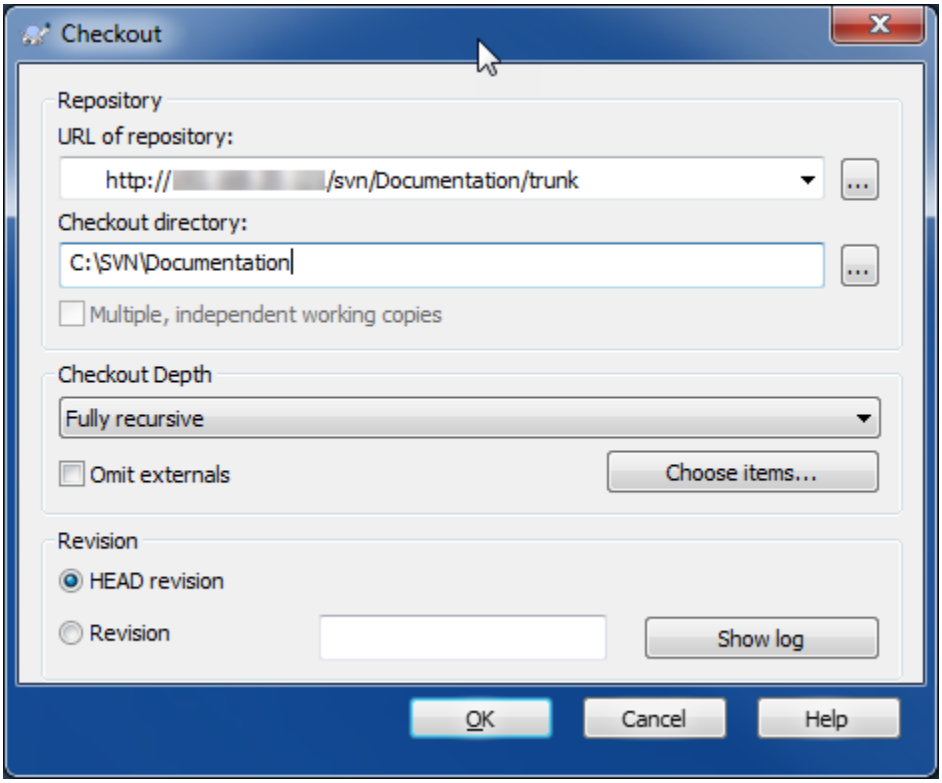
To work with files in Subversion, you need to "check out" the repository structure. Typically, this is only done once during the setup of TortoiseSVN on your computer. If you need to do major troubleshooting, deleting your existing files and repeating this step may be necessary.

To check out the repository:

1. Right-click the C:\SVN\Documentation folder.
2. Select SVN Checkout from the context menu.
3. In the Checkout window, specify http://[ip address]/svn/Documentation/ in the URL of Repository: field.
4. Click the ellipsis box to the right of the URL of Repository field.
5. Select the Trunk folder.

Note: If you have not previously saved your Subversion ID and password for this repository within TortoiseSVN, you are be prompted for it now. Specify your ID and password, select the Save Authorization check box to save them, and click OK.

**Checkout Window**




6. Click OK.
7. Ensure the Checkout Directory field displays the location you wish to create your working copy in and click OK.
8. In the Checkout Finished! dialog window click OK.

TortoiseSVN populates the C:\SVN\Documentation folder with the filesystem tree and working copies of the files. The appearance of this folder differs from the repository folder as every file has a green check mark in the bottom left corner. These are TortoiseSVN status icon overlays, which are present in a working copy. The green check mark shows the file is unchanged from the version in the repository. For more information on status information icons, refer to **Folder Icons**.

# USING SUBVERSION

The following section explains the basic procedures in Subversion you are most likely to need. If you encounter an issue not included in this guide, refer to TortoiseSVN help documentation.

## SUBVERSION BEST PRACTICES

The Documentation Team SVN repository stores files for the Release Notes process as well as internal and external guides.

To reduce the volume of files in the repository, do not commit the following files to SVN:

- TopLeaf log files – XMetaL generates these files whenever it uses a TopLeaf template to generate output. The file name is always TL-log.txt. Ensure the checkbox beside this file is not selected when committing your change to the main repository and delete it from your local repository to prevent an accidental commit.
- Source documents – Business Analysts (BAs) provide these business requirements documents and other source files. Store these files on a network drive instead.

## UPDATES AND COMMITS

When you wish to work with an SVN version-controlled document, you must first ensure you update your working copy of the file to the latest version in the repository. "Update" describes the process of the TortoiseSVN client connecting to the SVN server and downloading the latest version of the document in a repository.

Throughout the workday, regularly commit the revised document back to the SVN repository as a new version of the source document.

### UPDATING WORKING VERSIONS

You should update your local version of the repository to the latest version stored in the repository each time before you work on it.

1. Right-click on a folder containing versioned files and folders, and select TortoiseSVN > SVN Update.

Note: If the repository contains new or updated files, the update process copies these files from the repository to your local version.

Be careful when updating your local version of the repository. Any changes made to versioned files being updated are removed. If you wish to update a folder that contains changed versioned files, commit your changes to the repository first, and

run the update. Also, note that entire files and folders can be deleted when updating the working version if they were deleted in the repository since you last updated.

## COMMITTING YOUR CHANGES

You should commit your changes to the central repository, at a minimum, every day before lunch, and before going home for the day. Ensure you leave yourself enough time to deal with any possible conflicts. Also, ensure every commit has a thoroughly descriptive log message to aid reverting to an earlier version.

1. Right-click on the file, subfolder, or repository folder that covers the entire set of modified files or folders, and select SVN Commit.
   The Commit window displays every changed file, including added, deleted, and unversioned (objects in an SVN folder not under version control) files.
   If you do not want a changed file to be committed, uncheck the file. If you want to include an unversioned file, select the file to add it to the commit.

Note: Double-click the file to review any changes made.

2. Enter a concise log comment that fully describes all your changes.
   Ensure your comment is accurate, descriptive, and complete so other team members can understand how the new version differs from the previous.
3. Click OK to save your comment.
4. Click OK in the confirmation window.

Note: If your commit conflicts with changes made by another user, you need to resolve the conflicts. For more information on resolving conflicts, refer to **Conflicts**.

## LOG MESSAGES

For every change you make and commit, you must provide a log message. Including a log message when committing a file to the SVN repository is essential for efficient version control. That way you and your team can track the changes made and why, and have a detailed audit log.

Your log message should be concise, describing the changes and why you made them. If you made multiple changes, write a comment for each one. If you find yourself writing a long list of changes, consider making several smaller commits.

To show the change performed and to allow filtering of content, each commit message should use one of the following prefixes:

Table 2 Log Message Prefixes

| Prefix | Usage |
|---|---|
| **[Add]** | Use this prefix if you are adding new content or information |
| **[Modify]** | Use this prefix if you are correcting or otherwise modifying content |
| **[Delete]** | Use this prefix if you are removing content |

Note: When renaming a file, SVN deletes the original version of the file and creates a new file with the new name. As a result, the comment must include both a [Delete] and [Add] message for the file.

For example, the following log messages display the use of prefixes:

```
[Modify] Changed XML files and output files
[Modify] Made changes to the RELEASE.ditamap and component XML files

[Modify] Made changes based on peer review
[Modify] Regenerated PDF due to changes

[Delete] Renamed Draft.xml to Final.xml
[Add] Renamed Draft.xml to Final.xml

[Add] Added trunk/RELEASE folder
[Add] Added all XML files and subfolders
[Add] Created rtf and docx outputs
```

Note: It is unnecessary to list the files affected in the commit message. For more information on viewing previous commits, refer to **Log Messages Window**.

## COMMITTING CHANGES TO REFERENCE GUIDE MATERIAL

DITA maps for a release notes publication may include file references to XML files in a reference guide.

Note: These are reference guide source files in Subversion, so any changes to the release notes are in the main reference guide too.

To ensure all changes to these essential files are properly documented, a specific syntax is used for the log message when committing changes to these files.

Note: It is essential that this syntax is followed and all changes to reference guide files are fully documented.

The standard SVN log message prefixes of [Add], [Modify], and [Delete] should be used based on the modification being made.

For example, the following log messages display the correct level of information for changes made to reference guide files:

```
[Modify] RELEASE, Line 1 is removed from the guide
[Modify] RELEASE, The client field is added to the guide
[Add] RELEASE, The client file is created to store items used in file generation
```

## COMMITS DURING THE REVIEW PROCESS

The Documentation Team follows a thorough review process for all documentation. Revisions are often made because of these reviews. When committing edits resulting from reviews, include the review stage in the log message.

Potential review stages to be listed in the log message include:

- Self edit review edits applied
- Peer review edits applied
- Internal review edits applied
- BA review edits applied
- QC edits applied
- PDF proof edits applied

Each file may have edits from these stages applied multiple times during the documentation cycle. For example, a business requirements document might have changes made during its individual peer review, and again when the compiled release notes are peer reviewed.

## STYLING LOG MESSAGES

It is possible to apply simple formatting to log messages when committing changes to the repository. The following commands apply styles to the text they contain:

- *text* - applies **bold** styling to the text
- text_ - applies underlining to the text
- ^text^ - applies *italics* to the text

# PROJECT HISTORY

The ability to view the history of a file or folder is an important feature of version control systems. It allows users to quickly view the changes made to a file. The Log Messages window displays the history of files and folders in a Subversion directory.

The Log Messages window can be opened from several locations, including:

- TortoiseSVN context submenu
- Property page
- Progress dialog window after an update has finished.
  Here, the Log Messages window only shows those revisions since the last update.

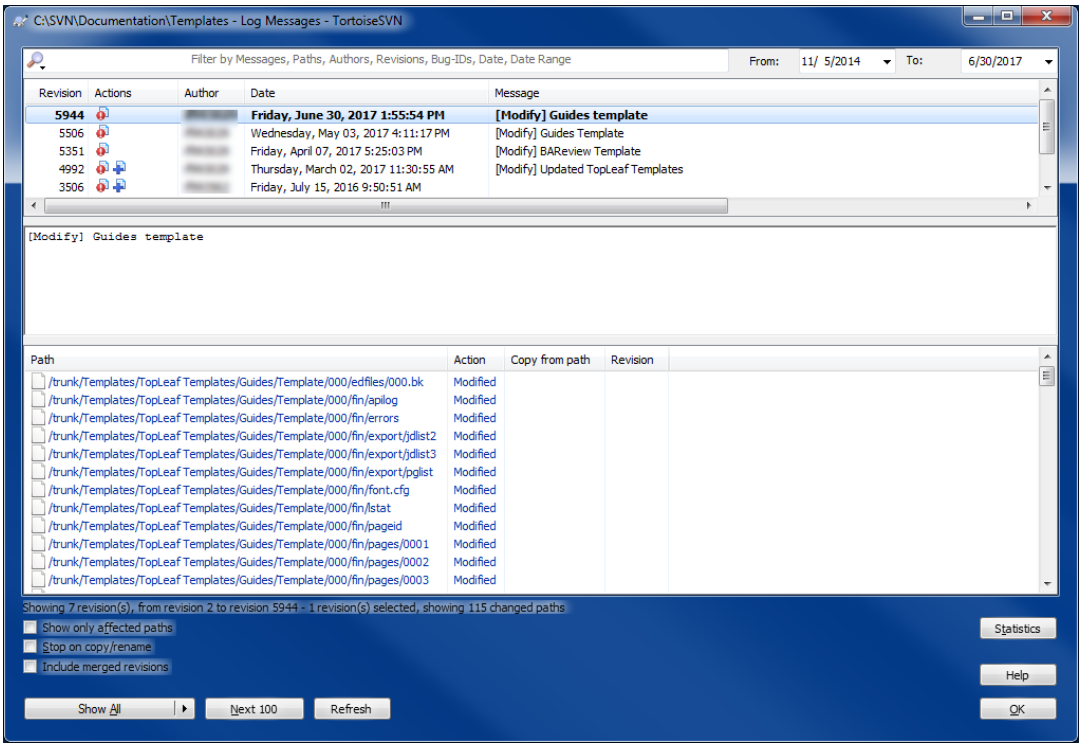
## LOG MESSAGES WINDOW

For every change you make and commit, you must provide a log message. That way you can locate the changes you made and why, and you have a detailed log.

Note: For more information on writing log messages, refer to **Log Messages**.

The Log Messages window retrieves and displays all of the logged messages. The following image displays the Log Messages window for a commit that included the addition, deletion, and modification of files.

**Log Messages Window**



The display is divided into three panes.

The top and bottom panes provide additional context menu commands which offer access to additional information about the project history. For detailed information on these context menus, refer to 4.9. Revision Log Dialog in the TortoiseSVN help documentation.

### LOG MESSAGES WINDOW – TOP PANE

The top pane displays a list of revisions where changes to the file/folder have been committed. The following information is displayed for each revision:

- The revision number
- The action performed in the revision
- The person who committed the revision
- The date and time of the commitment
- The start of the log message

The Actions column has four sub-columns and displays icons for each revision that summarize what actions were performed in the revision.

If a revision changed a file or directory, the Modified icon is displayed in the first sub-column. If a revision added a file or directory, the Added icon is displayed in the second sub-column. If a revision deleted a file or directory, the Deleted icon is displayed in the third sub-column. If a revision replaced a file or directory, the Replaced icon is displayed in the fourth sub-column.

Note: For more information on icons and icon overlays, refer to **Folder Icons**.

### LOG MESSAGES WINDOW – MIDDLE PANE

The middle pane displays the full log message for the selected revision.

### LOG MESSAGES WINDOW – BOTTOM PANE

The bottom pane displays a list of all files and folders that were changed as part of the selected revision. The previous image displays all of the files committed at the same time.

## ADDITIONAL OPERATIONS

This section covers a variety of additional common operations you may need to perform when using SVN.

### DIFFERENCES BETWEEN FILES

TortoiseSVN includes two tools for viewing differences between files:

- TortoiseMerge – displays differences between text files
- TortoiseIDiff – displays differences between image files

### VIEWING LOCAL CHANGES

To display the differences between your working copy and the latest repository version on the server:

1. Right-click the file.
2. Select TortoiseSVN > Diff.

### VIEWING DIFFERENCES FROM A REVISION

To display the differences between your working copy and a particular repository version:

1. Right-click the file.
2. Select TortoiseSVN > Show Log.
3. Right-click the desired revision.
4. Select Compare With Working Copy.

### VIEWING DIFFERENCES BETWEEN TWO REVISIONS

To display the differences between two revisions:

1. Right-click the file.
2. Select TortoiseSVN > Show Log.
3. Select the first revision to compare.
4. Hold down Ctrl and select the second revision to compare.
5. Right-click one of the selected revisions.
6. Select Compare Revisions.

### VIEWING DIFFERENCES BETWEEN IMAGES

To display the differences between two images:

1. Right-click the image file.
2. Select TortoiseSVN > Diff.
   The images are displayed side-by-side in TortoiseIDiff.

Note: For information on using TortoiseIDiff, refer to 4.10.4. Diffing Images Using TortoiseIDiff in the TortoiseSVN help documentation.

## CONFLICTS

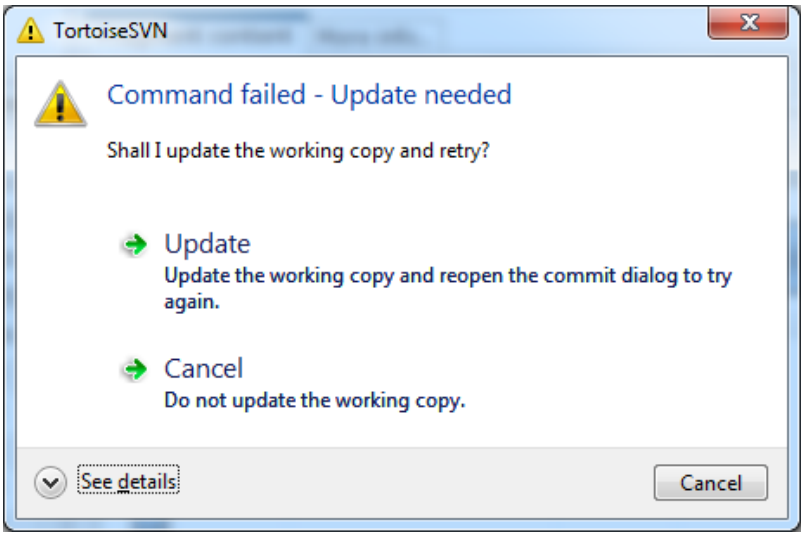
Conflicts occur when multiple users change the same file at the same time. There are two types of conflicts:

- File conflicts – A file conflict occurs if two (or more) users change the same few lines of a file.
- Tree conflicts – A tree conflict occurs if a user moves, renames, or deletes a file or folder which another user has moved, renamed, deleted, or modified.

Note: For more information on tree conflicts, refer to **Tree Conflicts Overview**.

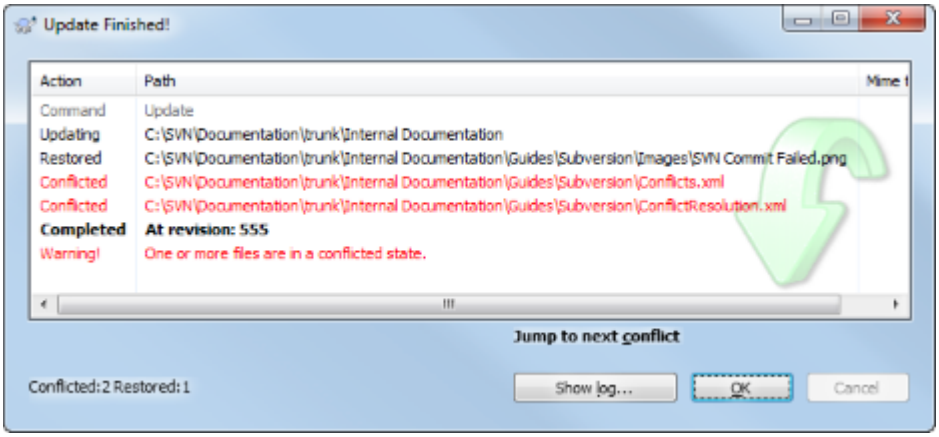
Conflicts occur when a user attempts to commit their changes to the repository. A pop-up window displays the following error message:

**TortoiseSVN Window – Command Failed**




When you run the TortoiseSVN Update operation, the following warning message is displayed:

**TortoiseSVN Update Finished! Window**




Note: For information on updating working versions of files from the repository, refer to **Updating Working Versions**.

After clicking OK, the folder containing the file in conflict contains several new, non-versioned files:

- Filename.mine – This is your file as it existed in your working copy before you updated your working copy. This file has your latest changes in it and nothing else. It contains no conflict markers.
- Filename.rOLDREV – This is the version of the file you checked out before you made any modifications to it.
- Filename.rNEWREV – This is a new version of the file which you just copied from the repository during the Update operation.
- Filename – During the update operation, conflict markers are inserted into this file to show the areas of conflict between the .mine and .rNEWREV versions of the file.

Note: Only Administrative users should resolve major file conflicts. When the conflict occurs, the team member who received the conflict warning message should contact the Documentation Team manager who will assign an Administrative user to resolve the conflict appropriately.

## USING TORTOISEMERGE

You can use TortoiseMerge to resolve conflicts in a file with the other user who was concurrently working on the file. Together, you decide which parts of each version of the file should go in the newly merged version.

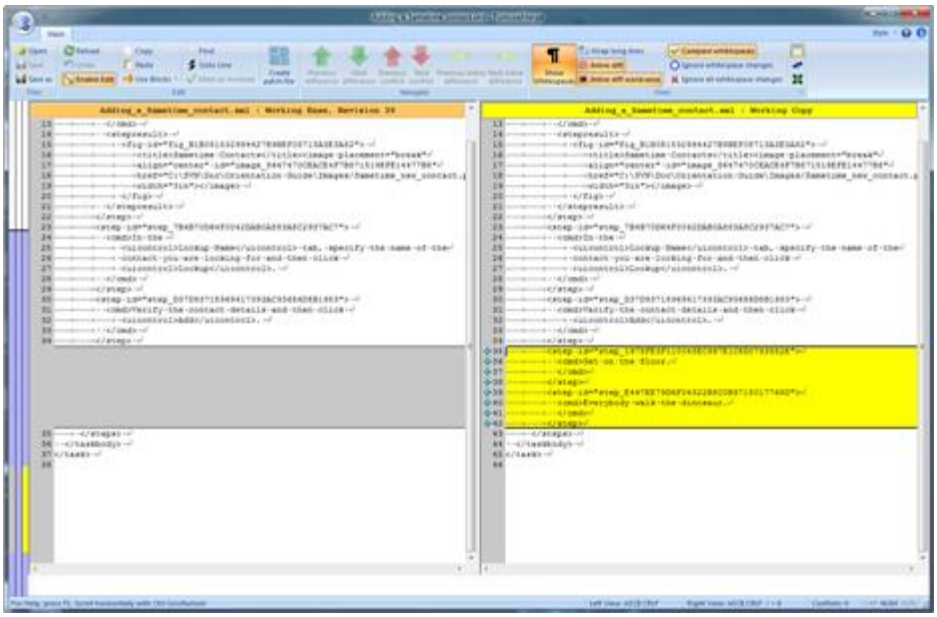
Note: It is essential you work on this together. Never overwrite or decide which changes should be kept on your own.

TortoiseMerge displays the differences between files in either a two-pane view or a three-pane view. Text is color coded within these panes:

- Added text is highlighted in yellow.
- Deleted text is highlighted in orange.
- Conflicts are highlighted in red.

## TORTOISE MERGE – TWO PANE VIEW

**TortoiseMerge Window – Two Pane View**



In the two-pane view, the left pane displays the differences between the other user's file and the base version. The right pane displays the differences between your file and the base version.

You can only edit the file in the right pane, which is the one you created.

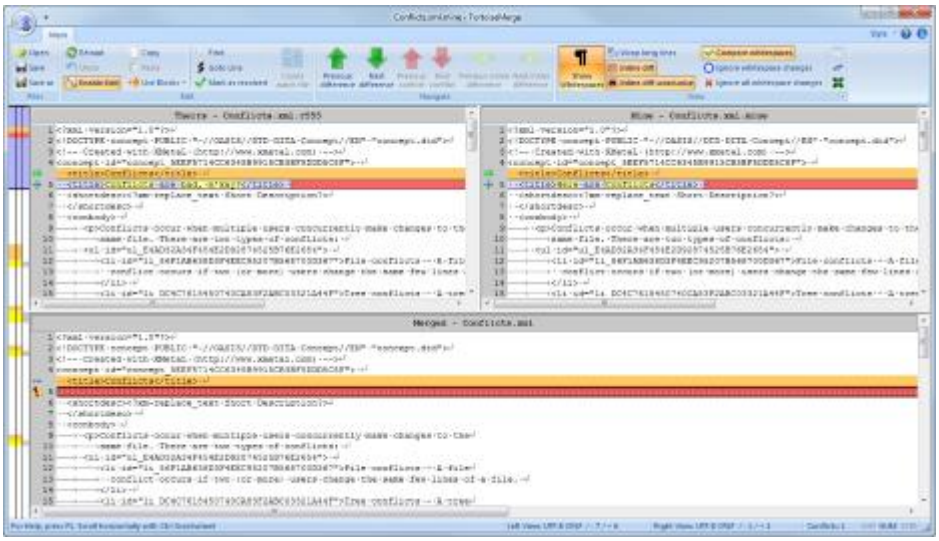
To apply changes made in the file belonging to the other user, displayed in the left pane, right-click on the changed lines and select Context Menu > Use Text Block From 'Theirs'. The changes from the file in the left pane are added to the file in the right pane.

To keep changes made in both documents to the same block of text, right-click on the changed lines and select Context Menu > Use Both Text Blocks (This One First) or Context Menu > Use Both Text Blocks (This One Last). Both sets of changes are made in the file in the right pane, with the changes made in the selected order.

Some lines in the output file can also be edited directly. Such lines are marked using a pencil icon.

## TORTOISE MERGE – THREE PANE VIEW

**TortoiseMerge Window – Three Pane View**



In the three-pane view, the left pane displays the differences between the other user's file and the base version. The right pane displays the differences between your file and the base version. The bottom panel displays the result of merging your version, their version, and the base version of the file with any resulting conflicts.

You can only edit the merged file in the bottom pane.

To apply changes made in the file belonging to the other user, displayed in the left pane, right-click on the lines in conflict and select Context Menu > Use Text Block From 'Theirs'. The changes from the file in the left pane are added to the merged file in the bottom pane.

To apply changes made in your file, displayed in the right pane, right-click on conflicted lines and select Context Menu > Use Text Block From 'Mine'. The changes from the file in the right pane are added to the merged file in the bottom pane.

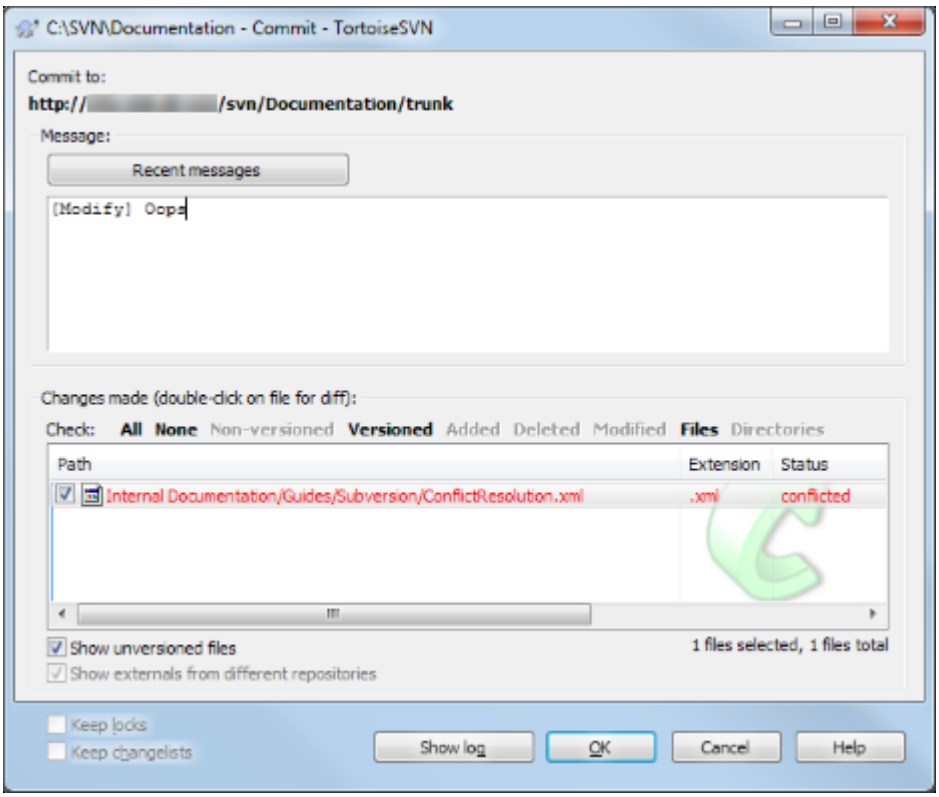
To keep changes made in both documents to the same block of text, right-click on the changed lines and select Context Menu > Use Both Text Blocks (This One First) or Context Menu > Use Both Text Blocks (This One Last) . Both sets of changes are made in the merged file in the bottom pane, with the changes made in the selected order.

## RESOLVING FILE CONFLICTS

The Commit window displays the files successfully committed and the files in conflict.

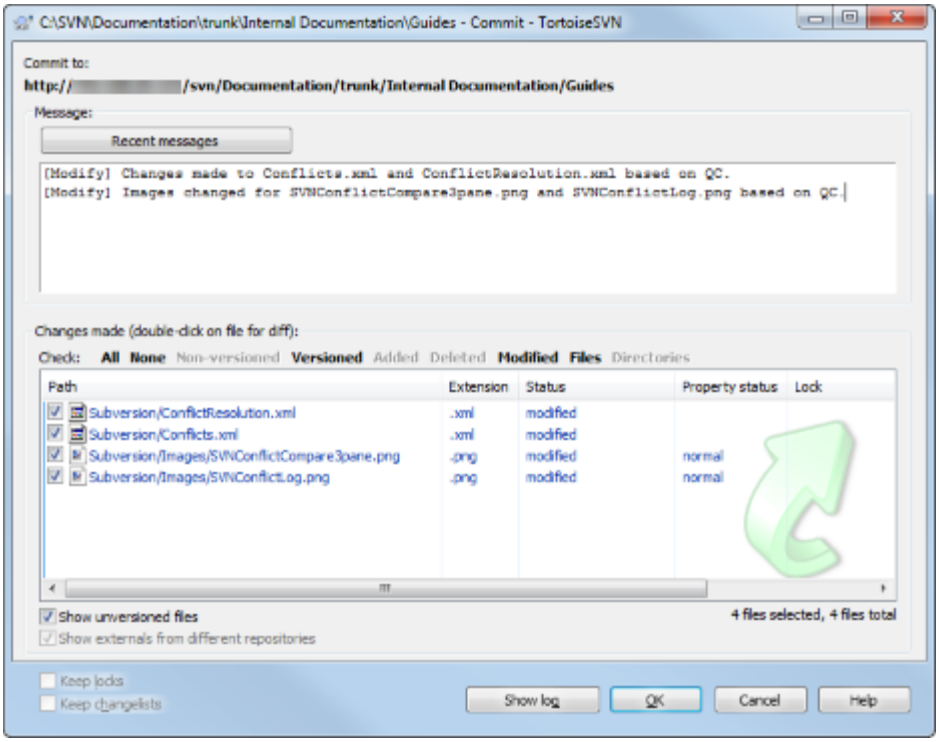
Note: Before starting this process, use the Log Messages window to find out which users made changes to the document and have them join you for this process.

**Commit Window**




1. Right-click the Filename version of the conflicted file, and select Edit Conflicts from the TortoiseSVN menu.
2. Use TortoiseMerge to decide which parts of the file should go in the merged version of the file with the other user who was concurrently working on the file. For more information on resolving conflicts with TortoiseMerge, refer to **Using TortoiseMerge**.
3. Resolve conflicts within TortoiseMerge. You can select to make the changes made by either user or directly alter the text within TortoiseMerge.
4. Select Merge from within TortoiseMerge.
5. Right-click the file.
6. Select TortoiseSVN > Resolved.
   This resolves the Conflicted status and removes the filename.mine, filename.rOLDREV, and filename.rNEWREV files allowing the changes to be committed.

**Commit Window**



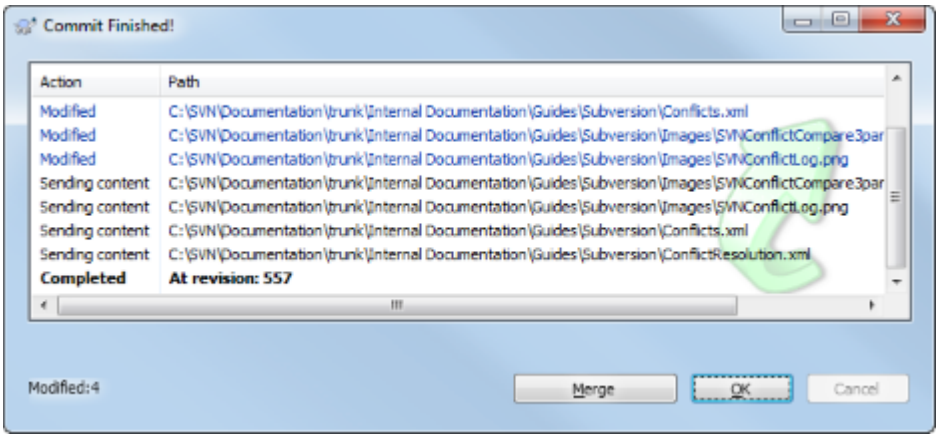

7. Commit the changes to the repository.

---

Note: For information on committing changes to the repository, refer to **Committing Your Changes**.

---

**Commit Finished! Window**



# ADMINISTRATION

This section details higher-level operations that not all users have the permissions to perform. More users may gain permissions as the team becomes more familiar with using Subversion and TortoiseSVN.

Note: Only Administrative users should perform operations in this section. If you require one of these operations and do not have permission to perform it yourself, speak with your manager to determine which users have the necessary permissions.

## REVERTING TO A PREVIOUS VERSION

There may be a case where you need to revert a local file to a previous revision:

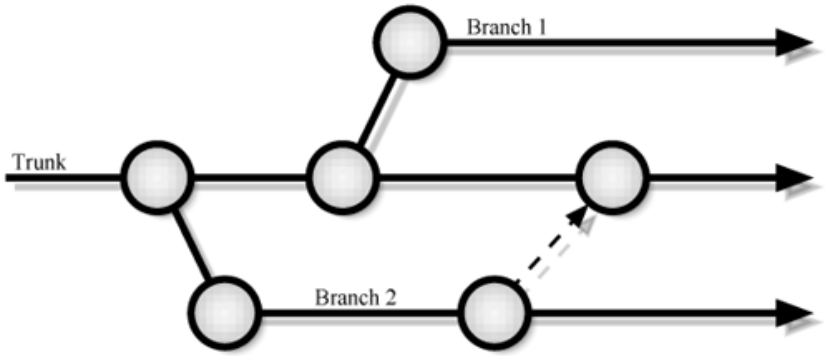
1. Right-click the file you wish to revert.
2. Select TortoiseSVN > Show Log.
3. Right-click the desired revision to revert in the top pane of the Log Messages window, and select Revert To This Revision.
4. Click Revert.
5. Click OK.

The file is reverted to the desired previous revision. The Modified icon overlay appears on the file until it is next committed to the repository.

## BRANCHES AND TAGS

One feature of version control systems is the ability to isolate changes into a separate line. This split-off line is known as a branch, and the main repository is referred to as the trunk. Branches are often used to try out new features without disrupting the main line of development with possible errors and bugs. As soon as the new feature is stable, the development branch can be merged back into the trunk.

**Branching Example**




Another feature of version control systems is the ability to mark particular revisions, such as a release version, to recreate a certain file. This process is called tagging.

Subversion does not have special commands for branching or tagging. Instead of making a complete copy in the repository, an internal link is created that points to a specific tree or revision. As a result, branches and tags are quick to create and take up almost no extra space in the repository.

## TAGGING

One feature of Subversion is the ability to mark particular revisions so a revision can be recreated. This process is known as tagging.

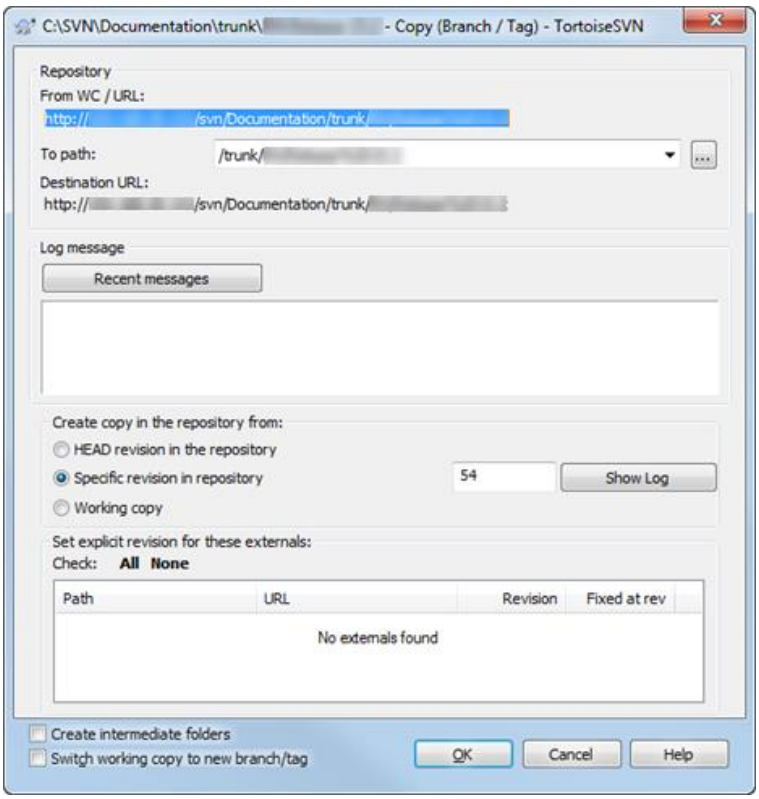
The Documentation Team tags documents whenever they are published. This allows us to identify which documents were published in a release, and when a document was last published.

Only Administrative users have the ability to access the Tag section of the repository. When the publication of a file is complete, the team member who published the document should contact the Documentation Team manager, who will assign an Administrative user to have the file tagged appropriately.

### CREATING A TAG

1. Update your working repository to ensure it is consistent with the most recent version in the trunk of the main repository.
2. From your working copy, right-click the folder you wish to tag.
3. Select TortoiseSVN > Branch/Tag.

**Copy (Branch / Tag) Window**




4. In the To Path field, specify the destination folder.
   The path should be C:\SVN\Documentation\tags\.

---

Note: Click the ellipsis (...) to browse to the desired destination folder if you are unsure.

---

5. Specify a log message.

   The log message for a tag differs from the standard log messages referenced in **Log Messages**. The log message always begins with [Add] as you are creating the tag. The message should specify the following information, separated by commas:

   - The name of the document published
   - The release number in which the document was published
   - The date the file was published in mm/dd/yyyy format

   For example, the following log messages display the correct level of information for tags:

   ```
   [Add] Guide Name, RELEASE, 1/22/2015
   [Add] Guide Name, RELEASE, 1/29/2015
   ```

6. Select the desired source for the copy.

- To copy the latest revision in the repository, select HEAD Revision In The Repository
- To copy a specific revision in the repository, select Specific Revision In Repository and specify the revision in the text field. Use this option only if changes have been made to the HEAD revision since the published revision.

Note: Click Show Log to display the log messages for all revisions of the selected folder.

- To copy your local working version, select Working Copy. Ensure your changes are committed to the main repository and perform this process again, selecting HEAD Revision In The Repository instead.

7. Click OK.

### OTHER WAYS TO TAG

You can also create a tag without having a working copy by using the repository browser:

1. Open the repository browser.
2. Select the folders to be tagged.
3. Hold down the Ctrl key and drag the folders to the desired location.

Note: You must hold down the Ctrl key while dragging the folders to create a copy. Otherwise, the folders are moved, not copied.

Or you can do it this way:

1. Drag the folders to the desired location using the right mouse button.
2. Release the right mouse button and select Copy from the context menu.

Note: You must select Copy to create the tag.

## TREE CONFLICTS OVERVIEW

A tree conflict occurs if a user moves, renames, or deletes a file or folder that another user also has moved, renamed, deleted, or modified. There are many situations that can cause a tree conflict, and all of them require different steps to resolve the conflict. These are considerably more complex to correct than file conflicts and should be avoided as much as possible.

To prevent tree conflicts, only some users have permission to rename or delete files and folders. If these conflicts occur, an Administrative user is required to resolve the conflict.

### TREE CONFLICT AVOIDANCE

To prevent tree conflicts, only Admin users should rename and delete files and folders.

### RENAMING FILES AND FOLDERS

1. Check out the desired file or folder to your computer.

Note: For information on checking out files and folders from the repository, refer to **Checking Out the Repository**.

2. Right-click the file or folder, and select SVN Rename from the TortoiseSVN menu.
3. Specify the new name.
4. Commit the file or folder.

Note: For information on committing changes to the repository, refer to **Committing Your Changes**.

Note: Subversion uses an uncommon method of renaming files or folders which requires additional actions. When a file or folder is renamed, a new file or folder is created and added to the repository with the new name. However, the original file or folder remains in the repository. For example, if you renamed the file Guide.ditamap to UserGuide.ditamap, ran a Commit, then ran a Checkout of Guide.ditamap to UserGuide.ditamap's containing a folder, both files would be downloaded from the repository. Once you've renamed the file or folder, you should delete the original from the repository.

### DELETING FILES AND FOLDERS

1. Check out the desired file or folder to your computer.
2. Right-click the file or folder, and select Delete from the TortoiseSVN menu. The file is deleted from your computer.
3. Commit the file or folder and ensure the check boxes are selected for deleting the desired items.

Note: For information on committing changes to the repository, refer to **Committing Your Changes**.

## RESOLVING TREE CONFLICTS

A tree conflict occurs when a user moves, renames, or deletes a file or folder another user has also moved, renamed, deleted, or modified.

When a file is deleted locally in Subversion, the file is also deleted from the local file system. As a result, it cannot show a conflicted overlay icon even if it is part of a tree conflict, and you cannot right-click on it to resolve the conflict. Use the Check For Modifications dialog instead to access the Edit Conflicts option.

TortoiseSVN can help find the right place to merge changes, but there is often additional work required to sort out the conflicts. Remember that after an update the working copy always contains the revision of each item as it was in the repository at the time of update. If you revert a change

after updating, it goes back to the repository state, not to the way it was when you made your own local changes.

### EXAMPLE: LOCAL DELETION WITH INCOMING EDITS UPON UPDATE

This conflict results when the following occurs:

- User 1 modifies RELEASE.ditamap and commits it to the repository.
- At the same time, User 2 has deleted RELEASE.ditamap or its parent folder, or renamed RELEASE.ditamap to RN.ditamap in their working copy.

When User 2 updates their working copy, a tree conflict results and the following events occur:

- RELEASE.ditamap is deleted from the working copy, but is marked with a tree conflict.
- If the conflict was caused by renaming the file rather than deleting it, the new filename (RN.ditamap) is marked as added, but does not contain the modifications made by User 1.

User 2 has to choose what to do with the changes made by User 1:

- In the case of renaming the file, they can choose to merge the changes made in RELEASE.ditamap into the renamed file RN.ditamap.
- For simple file or directory deletions, they can choose to keep the item with the changes made by User 1 and discard the deletion.
- They can mark the conflict as resolved without doing anything, which discards any of the changes made by User 1.

The conflict edit dialog offers to merge changes if it can find the original file of the renamed RN.ditamap. Depending on where the update was invoked, it may not be possible to find the source file.

### EXAMPLE: LOCAL EDIT WITH INCOMING DELETION UPON UPDATE

This conflict results when one of the following scenarios occurs.

Scenario 1 – Renaming the file:

- User 1 renames RELEASE.ditamap to RN.ditamap and commits it to the repository.
- User 2 modifies RELEASE.ditamap in their working copy.

When User 2 updates their working copy, a tree conflict results. In this scenario, it is a conflict of a single file and the following events occur:

- RN.ditamap is added to the working copy as a normal file.
- RELEASE.ditamap is marked as if it was added (with history) and has a tree conflict.

Scenario 2 – Moving or renaming the folder:

- User 1 moves the parent folder of RELEASE.ditamap or renames it, and commits it to the repository.
- User 2 modifies RELEASE.ditamap in their working copy.

When User 2 updates their working copy, a tree conflict results. In this scenario, it is a folder conflict and the following events occur:

- The moved or renamed folder is added to the working copy as a normal folder.
- The original folder is marked as added (with history) and has a tree conflict.
- RELEASE.ditamap is marked as modified.

User 2 has to decide whether to accept the reorganization performed by User 1 and merge their changes into the corresponding file in the new structure, or revert the changes made by User 1 and keep their local version of the file.

To merge their local changes with the reorganization, User 2 must first find out to what filename the conflicted file RELEASE.ditamap was renamed to, or where it was moved in the repository. This can be done using the log dialog. The changes must be manually merged as there is no way to automate or even simplify this process. Once the changes are made, the conflicted path is redundant and can be deleted. In this case, use the Remove button in the conflict editor dialog to clean up and mark the conflict as resolved.

If User 2 decides the changes made by User 1 were incorrect, they click the Keep button in the conflict editor dialog. This marks the conflicted file or folder as resolved. However, the changes made by User 1 need to be removed. Again, the log dialog helps to track down what was moved.

### EXAMPLE: LOCAL DELETION WITH INCOMING DELETION UPON UPDATE

This conflict results when the following occurs:

- User 1 renames RELEASE.ditamap to RN.ditamap and commits it to the repository.

- User 2 renames RELEASE.ditamap to RNYYY.ditamap and commits it to the repository.

When User 2 updates their working copy, a tree conflict results and the following events occur:

- RNYYY.ditamap is marked as added with history.
- RN.ditamap is added to the working copy with a status of normal.
- RELEASE.ditamap is marked as deleted and has a tree conflict.

To resolve this conflict, User 2 has to find out what filename the conflicted file RELEASE.ditamap was renamed to in the repository. This can be done by using the log dialog.

Next, User 2 has to decide which new filename of RELEASE.ditamap to keep:

- RN.ditamap by User 1
- RNYYY.ditamap by User 2

After User 2 has manually resolved the conflict, the tree conflict has to be marked as resolved in the conflict editor dialog.

### EXAMPLE: LOCAL MISSING WITH INCOMING EDIT UPON MERGE

This conflict results when the following occurs:

- User 1 is working on the trunk and modifies RELEASE.ditamap and commits it to the repository.
- User 2 is working on a branch and renames RELEASE.ditamap to RN.ditamap and commits it to the repository.

Merging trunk changes made by User 1 to the branch working copy of User 2 results in a tree conflict and the following events occur:

- RN.ditamap is already in the working copy with a status of normal.
- RELEASE.ditamap is marked as missing with a tree conflict.

To resolve this conflict, User 2 has to mark the file as resolved in the conflict editor dialog, which removes it from the conflict list. They have to decide whether to copy the missing file RELEASE.ditamap from the repository to the working copy, whether to merge the changes made by User 1 to RELEASE.ditamap into the renamed RN.ditamap, or whether to ignore the changes by marking the conflict as resolved and doing nothing else.

**Note:** If you copy the missing file from the repository and mark it as resolved, your copy will be removed again. You have to resolve the conflict first.

### EXAMPLE: LOCAL EDIT WITH INCOMING DELETE UPON MERGE

This conflict results when the following occurs:

- User 1 is working on the trunk and renames RELEASE.ditamap to RN.ditamap and commits it to the repository.
- User 2 is working on a branch and modifies RELEASE.ditamap and commits it to the repository.

Merging the trunk changes made by User 1 to the branch working copy of User 2 results in a tree conflict and the following events occur:

- RN.ditamap is marked as added.
- RELEASE.ditamap is marked as modified with a tree conflict.

User 2 has to decide whether to accept the reorganization performed by User 1 and merge their changes into the corresponding file in the new structure, or revert the changes made by User 1 and keep their local version of the file.

To merge their local changes with the reorganization, User 2 must first find out what filename the conflicted file RELEASE.ditamap was renamed to, or where it was moved in the repository. This can be done by using the log dialog for the merge source. The conflict editor only shows the log for the working copy as it does not know which path was used in the merge. The changes must be manually merged as there is currently no way to automate or even simplify this process. Once the changes have been made, the conflicted path is redundant and can be deleted. In this case, use the Remove button in the conflict editor dialog to clean up and mark the conflict as resolved.

If User 2 decides the changes made by User 1 are incorrect, they click the Keep button in the conflict editor dialog. This marks the conflicted file or folder as resolved. However, the changes made by User 1 need to be manually removed. Again, the log dialog for the merge source helps track what was moved.

### EXAMPLE: LOCAL DELETION WITH INCOMING DELETION UPON MERGE

This conflict results when the following occurs:

- User 1 is working on the trunk and renames RELEASE.ditamap to RN.ditamap and commits it to the repository.

- User 2 is working on a branch and renames RELEASE.ditamap to RNYYY.ditamap and commits it to the repository.

Merging the trunk changes made by User 1 to the branch working copy of User 2 results in a tree conflict and the following events occur:

- RNYYY.ditamap is marked with normal (unmodified) status.
- RN.ditamap is marked as added with history.
- RELEASE.ditamap is marked as missing and has a tree conflict.

To resolve this conflict, User 2 has to find out to what filename the conflicted file RELEASE.ditamap was renamed to in the repository. This can be done by using the log dialog for the merge source. The conflict editor only shows the log for the working copy as it does not know which path was used in the merge.

Next, User 2 has to decide which new filename of RELEASE.ditamap to keep:

- RN.ditamap by User 1
- RNYYY.ditamap by User 2

After User 2 has manually resolved the conflict, the tree conflict has to be marked as resolved in the conflict editor dialog.

### OTHER TREE CONFLICTS

There are other cases that are labelled as tree conflicts because the conflict involves a folder rather than a file. For example, if you add a folder with the same name to both the trunk and branch, and try to merge, you get a tree conflict. If you want to keep the folder from the merge target, mark the conflict as resolved. If you want to use the one in the merge source you need SVN to delete the one in the target first and run the merge again. With anything more complicated, you have to resolve it manually.

## TROUBLESHOOTING

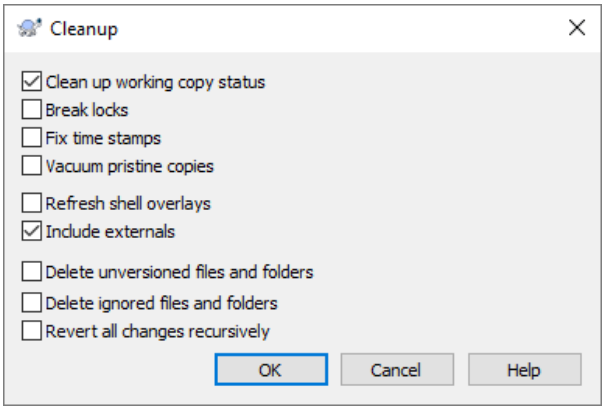
### USING THE CLEANUP TOOL

Sometimes an SVN Update fails to update or restore the repository. When this happens, you may receive an error in the Update window requesting you run the Cleanup command. To run a Cleanup, follow the steps below:

1. Right-click the Documentation folder and click Tortoise SVN > Clean up.

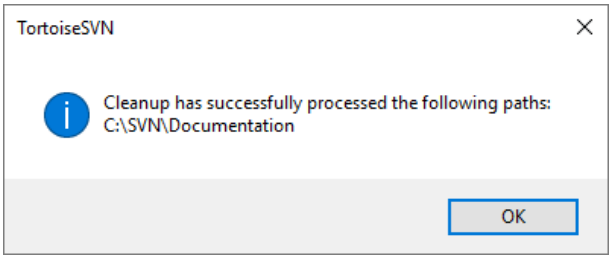
2. The Cleanup dialog window appears.

### Cleanup Window




3. Accept the default settings and click OK.
4. A success message dialog window appears when complete.

### Cleanup Success Window




## NUCLEAR OPTION: REMOVING THE REPOSITORY AND STARTING AGAIN

The last resort when trying to troubleshoot SVN issues is to remove the entire SVN repo and start over. To delete your local SVN repo, follow the steps below:

1. Open Windows Explorer and navigate to C:\SVN\Documentation.
2. Highlight all folders at this location (there should be five).
3. While holding the Shift button on your keyboard, press Delete.
4. A confirmation message asks if you wish to delete the files permanently. Click Yes.
5. Once the files are removed, right-click within the Documentation folder and click SVN Checkout. This retrieves the latest SVN files and folders from the server and places them into the Documentation folder.