This paper was originally presented at the Southwest Fox conference in Gilbert, Arizona in October, 2012. http://www.swfox.net

Advanced Topics in Mercurial: Taking it to the Next Level

Rick Borup
Information Technology Associates
701 Devonshire Dr, Suite 127
Champaign, IL 61820
Voice: (217) 359-0918
Email:rborup@ita-software.com

This session begins where Rick's Southwest Fox 2011 presentation on VFP Version Control with Mercurial left off. After a quick review of Mercurial's basic concepts and commands, this session explores more advanced topics drawn from the author's real-world, everyday experiences as a full-time VFP developer using Mercurial. This session is designed for VFP developers who want to improve their working knowledge of Mercurial as well as for those who have not yet started but want to begin using Mercurial in their everyday development work. You don't need any previous experience with Mercurial to benefit from this session, but for those who didn't get a chance to see it yet, the white paper from the 2011 introductory session is available for download from Rick's website at www.ita-software.com/foxpage.aspx.

Table of Contents

Reviewing the basics

This first section is a brief review of the basics for working with Mercurial. Key concepts and terminology are italicized – these are terms you should be familiar with before going on. If you're new to Mercurial or if these terms and concepts are unfamiliar to you, you may first want to go back and read my *VFP Version Control with Mercurial* paper from Southwest Fox 2011. It includes an extensive introduction to the subject and is available for download from my website at www.ita-software.com/foxpage.aspx.

Working with repositories

Mercurial is a *distributed version control system*, or *DVCS*. Unlike a centralized version control system, in which all developers share a central repository, developers using a DVCS each have their own *local repository* for every project.

Mercurial repositories are file-based. The files that comprise the repository for any given project are stored in a subfolder named *.hg*, which is located directly under the project's root folder. The files in the project's root folder, along with the files in any of its subfolders other than the .hg subfolder, are collectively referred to as the *working directory*, or working copy.

The local repository for a project is created by issuing the *init* command from the project's root folder. This action creates the initial folder structure and files for the local repository, and needs to be done only once for each project. The *add* command is then used to tell Mercurial which files in the working directory to place under version control in that repository.

The basic workflow when using Mercurial is that you make changes to the files in your working copy and then *commit* those changes to the local repository. Mercurial stores the revisions in the form of *changesets*, which are files containing the information necessary to describe the changes from one version of a file to the next version.

Changesets are identified by a *revision number*, which is an integer value scoped to the specific repository, and also by a *changeset ID*, which is a hexadecimal string that's unique across all repositories. Revision numbers and changeset IDs are generated automatically by Mercurial.

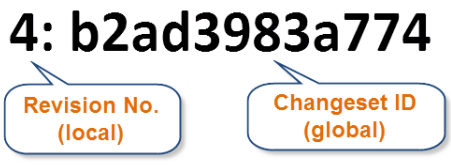


Figure 1: Changesets are identified by a revision number, which is local to the specific repository, and by a changeset ID, which is global.

Changesets are organized into a hierarchy within a repository. A revision from which a subsequent revision is derived is called a *parent* revision, and the derived revision is called the *child* of that parent. Due to merging, a revision may have more than one parent. A revision that has no child revisions is called a *head* revision. The most recent revision in a repository is called the *tip* revision. The tip is always a head, but not all heads are tips.

The commit command creates a new revision in the local repository, reflecting the change or changes made to one or more files in the working directory. The *update* command works in the other direction, modifying the files in the working directory to reflect their state as of a specific revision in the local repository. In common usage the word "update" means to make to more current, but in Mercurial the update process can modify a file to reflect either an older or a newer revision.

Developers using a DVCS can also create and share *remote repositories*. A remote repository is any other repository that is related to the same project as the local repository. Remote repositories can be accessed via the local file system or via http/https, depending on where they are located and how they are configured. Developers can choose to share changesets directly between their individual local repositories, or they may decide to use a central remote repository.

When working with one or more remote repositories, developers periodically *push* the changes from their local repository up to a remote repository in order to make them available to others. Other developers can then *pull* these changes into their own local repository and *update* their working copy by *merging* those changes. Developers working in a team environment need to periodically pull other developers' changes from the appropriate remote repository and merge them into their own working copy in order to stay in sync with the team.

Mercurial also provides a way to *clone* a repository. Cloning creates an entirely new local repository from a remote one. This provides a way for people to start working with a project entirely from scratch. A cloned repository has all of the change history the original has, resulting in an identical copy. After cloning a repository, the *update* command is used to create the files in the corresponding working directory.

One decision the developer or team needs to make for every project is which files to place under version control – i.e., which files to include in the repository – and which files to exclude. The general rule of thumb is to include all source code files, which are plain text files in most development environments, and to exclude binary files such as images and other non-source code resources. However, if you want other people to be able to clone your repository and get a full working copy, your repository needs to include everything necessary to build the app, which may include binary files such as images, external libraries, and others that might ordinarily be excluded.

Mercurial uses a file named *.hgignore* to store the list of files to be excluded. This file is stored in the root of the working directory. When you do a commit, Mercurial ignores the

files listed in the .hgignore file. The .hgignore file typically changes over the life of a project, so it should be included in the repository.

Mercurial provides a way to create *branches* within a repository. Branches are typically given names, although sometimes anonymous branches get created. Every repository starts out with a main branch named *default;* all changesets are stored in the default branch unless additional branches are created. Branches are typically used to facilitate multi-track development work, for example when it's necessary to support one or more release branches and also possibly multiple maintenance branches. Branches can also be used to facilitate experimentation, so that changes can be committed to the experimental branch without affecting the revision history of the main branch.

Using Mercurial with Visual FoxPro

Visual FoxPro developers have a special set of concerns when it comes to using a version control system. The majority of these concerns stem from the fact that VFP uses binary files to store the source code for things like forms and visual class libraries. Version control systems can't do merges on binary files, which defeats one of the principal benefits of using a version control system in the first place.

The basic choice for the VFP developer is whether or not to include the binary source code files in the repository. If you exclude them, you not only lose the ability to revert to a previous version but you also defeat the ability of other developers to build the project from a clone of your repository, since the repository does not contain all the necessary files. If you do include the binary source code files in the repository, you solve those two problems but are still left with the issue of how to share and merge changes among two or more developers, or even between two or more branches in your own repository.

The solution adopted by most VFP developers is to use a tool like SCCText, SCCTextX, or Christof Wollenhaupt's TwoFox to create text files from VFP's binary source code files. The first two create plain text files, while TwoFox uses XML. To the best of my knowledge, TwoFox is the only two-way tool, meaning it can re-generate the binaries from the XML.

If you use one of those tools to create text files from VFP's binary source code files, you should of course include these text files in the repository. Merge operations can then be performed on the text files. Even if you do use text files, you may also want to include the binary files in the repository. This may seem redundant, but including them at least gives you a way to revert to an earlier version, which can be important if the tools you're using don't offer a way to re-generate a binary file from its corresponding text file.

Files outside a project's root folder are not included in the repository. Visual FoxPro projects typically reference many files that are outside the root folder. When the project is built, the VFP Project Manager takes care of locating and bringing these resources into the build. However, these files are not included when you commit changes to the local repository so they're not automatically under version control and they won't be present in a cloned copy of your repository. Therefore, as a VFP developer you may want to consider

making copies within your project's root folder of external resources such as graphics files that other developers are not likely to have, or which they may have stored in a different relative location on their machine than where they are on yours.

Working with the TortoiseHg shell extension

Mercurial is a command line tool. Although you can work with it from the command prompt, most Windows developers will prefer using the TortoiseHg shell, which provides a familiar, Windows-based GUI interface to Mercurial's commands and features. The TortoiseHg shell for Mercurial is an implementation of the same shell used for TortoiseSVN and TortoseGit, so you may find it familiar if you've already used one of those.

TortoiseHg and Mercurial come bundled together in one installer package, available for download from http://tortoisehg.bitbucket.org/. When you use this installer the files for both TortoiseHg and Mercurial are written to the same folder, which is C:\Program Files\TortoiseHg on a 32-bit Windows machine.

There are three executable files in that folder you should know about.

hg.exe is the executable file for Mercurial itself. If you're working from the command prompt, you can run Mercurial commands by typing *hg* followed by the name of the command.

thg.exe is the executable file for TortoiseHg commands. TortoiseHg has its own set of commands, different from the *hg* commands for Mercurial. The TortoiseHg commands can be run from the command prompt using *thg* followed by the command name. Some of these commands open windows you can interact with.

thgw.exe is the executable file for the TortoiseHg Workbench. This is primary tool you'll use for interacting with Mercurial via TortoiseHg. The TortoiseHg Workbench is actually a container for many of the TortoiseHg graphical tools, some of which can be launched individually using the *thg* command.

You can launch the TortoiseHg Workbench either by typing *thgw.exe* from a command prompt, or by selecting *Hg Workbench* from the Windows Explorer context menu.

Figure 2 shows the TortoiseHg Workbench open in a typical configuration. Take a few minutes to become familiar with the various pieces annotated in this figure.

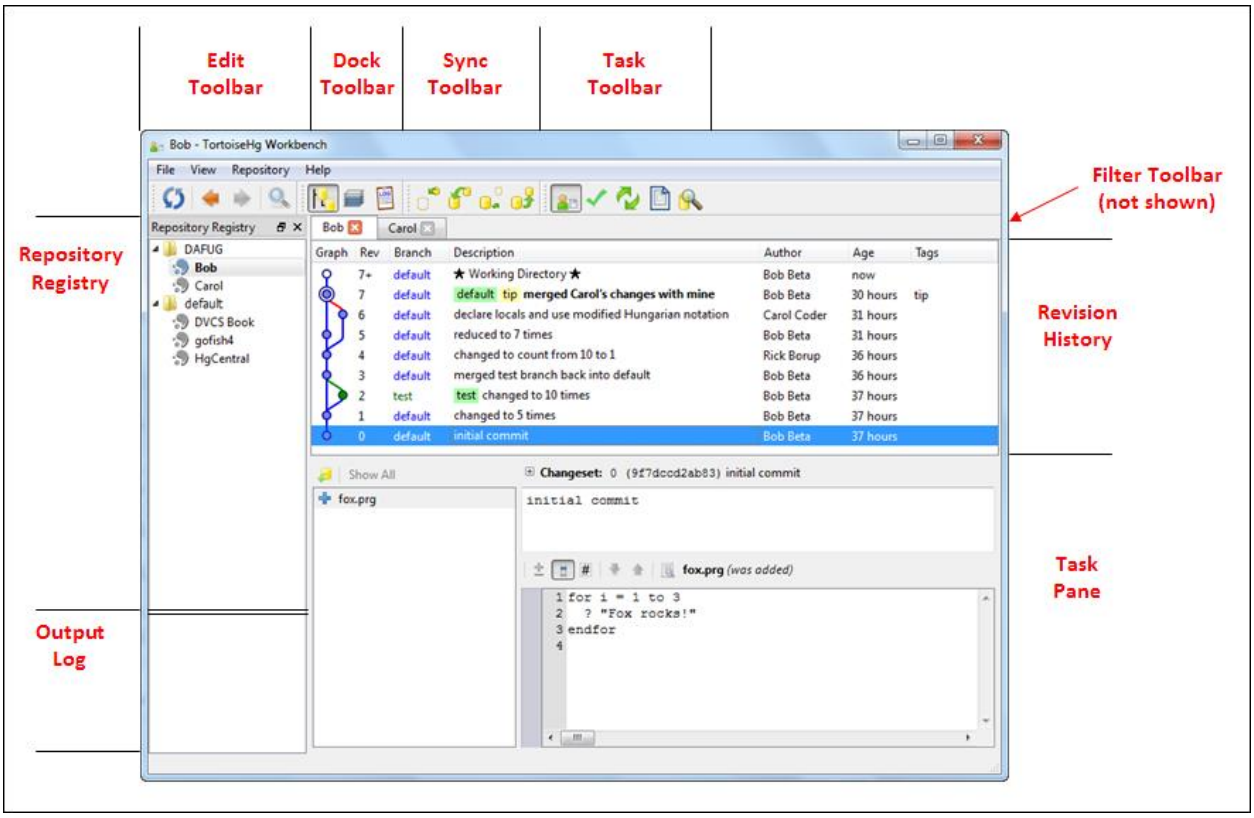


Figure 2; You'll be spending a lot of time in the TortoiseHg Workbench, so take some time to become familiar with its parts as annotated in this figure.

The tree view along the left side of the Repository Registry enables you to organize and group repositories for easy reference. You can add whichever repositories you want to the Repository Registry.[1] The Revision History in the upper main panel is a graphical representation of the revision history for the selected repository. The information displayed in the Task Pane in the lower main panel changes according to the task selected in the Task Toolbar (center top). The TortoiseHg workbench features a tabbed interface so you can have more then one repository open at the same time.

Configuring Mercurial and TortoiseHg

Before you begin working with Mercurial, there are a couple of configuration settings you'll want to change from their default values. In Mercurial there are two sets of configurations: global configuration settings are associated with the Windows user account and act as the default for all repositories operated upon by that user, while repository-specific configuration settings apply to individual repositories and can be used to override the user's global defaults.

[1] Despite its name, the Repository Registry has nothing at all to do with the Windows registry. The TortoiseHg Repository Registry information is stored in a file named thg-reporegistry.xml file, located in the TortoiseHg subfolder under common application data.

Global (user) configuration

Global configuration settings are stored in a file named mercurial.ini, located in the user's profile folder. The easiest way to set the values for this configuration file is to use the global settings dialog in the TortoiseHg Workbench, as shown in Figure 3. Open the global settings dialog by going to File | Settings on the main menu and selecting the global settings page. The global settings file can also be edited manually, either by opening it directly in a file editor or by clicking the Edit File button and using TortoiseHg's built-in editor.

At a minimum, you should set a value for the Username field in the Commit section. This establishes the name and email address that will be associated with changesets you commit. There is no hard and fast rule for the format to use, but the convention is your name followed by your email address inside angle brackets, as in *Rick Borup <rborup@ita-software.com>*. If you don't set a Username, Mercurial will prompt you for one every time you do a commit.

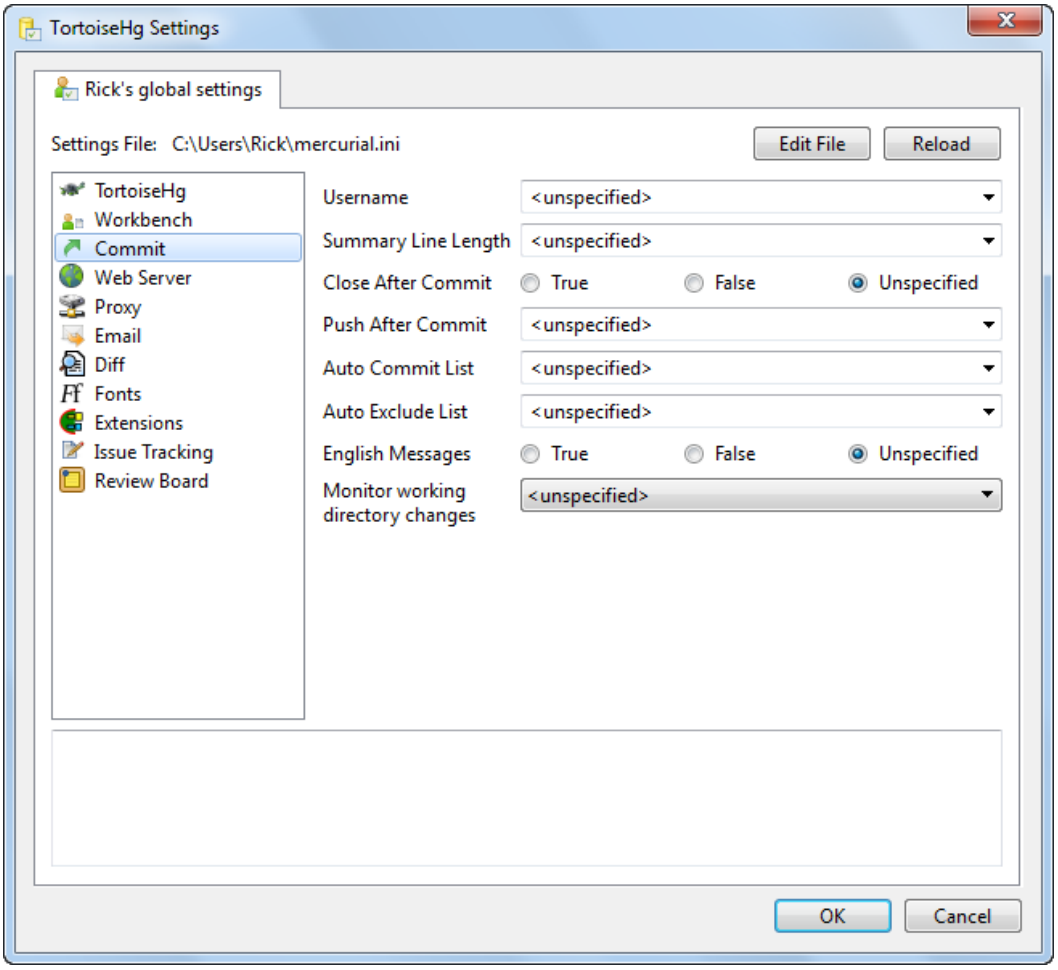


Figure 3: Global settings are stored in the *mercurial.ini* file in the user's profile folder.

TortoiseHg comes with a visual diff tool named kdiff, but if you prefer to use another one you can specify it in your global settings. Select the TortoiseHg group, then select the

desired tool from the drop-down list. The drop-down list is populated with entries for the appropriate tools TortoiseHg recognizes as being installed on your machine. I like to use BeyondCompare as my three-way merge tool and visual diff tool, as shown in Figure 4.

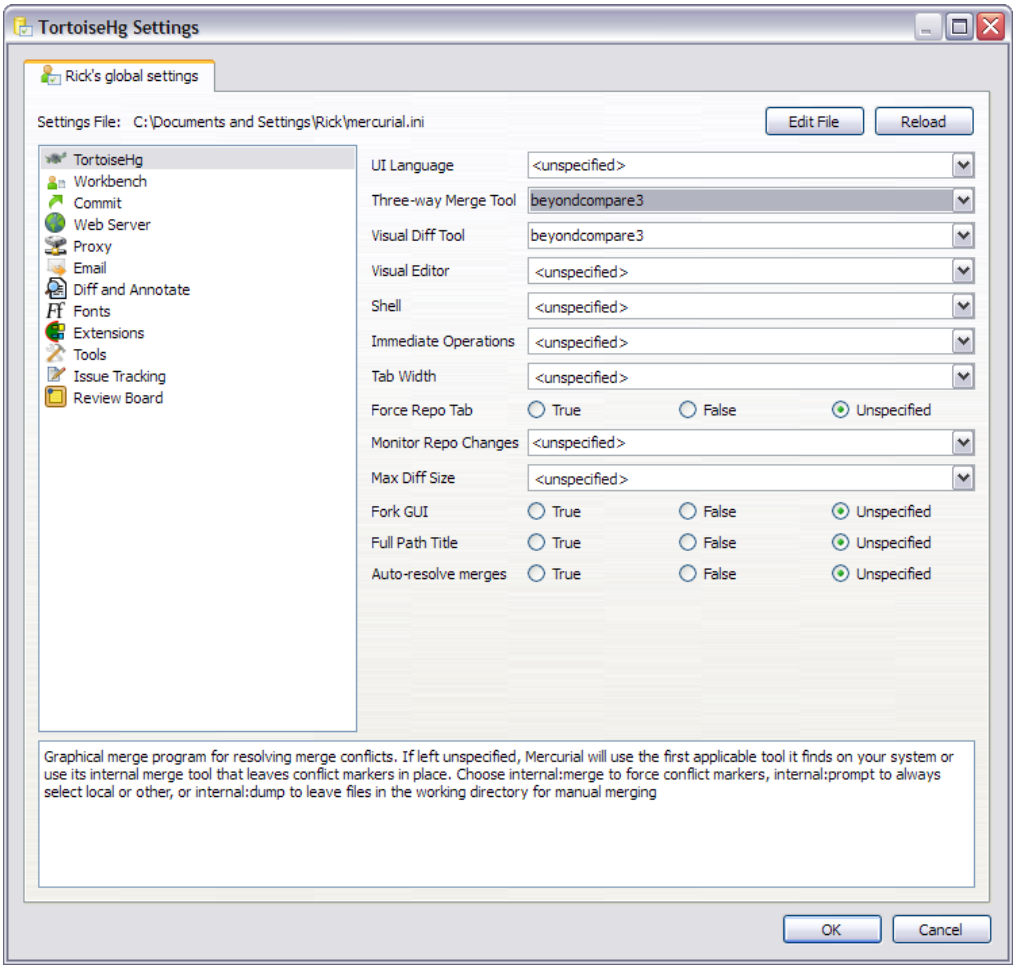


Figure 4: Set your preferred visual diff and merge tool in the TortoiseHg global settings dialog.

The mercurial.ini file is organized into sections containing lines with *name = value* pairs just like any other INI file. Clicking the Edit File button brings up TortoiseHg's own file editor where you can view or make changes. Figure 5 shows the mercurial.ini file resulting from the settings made above.

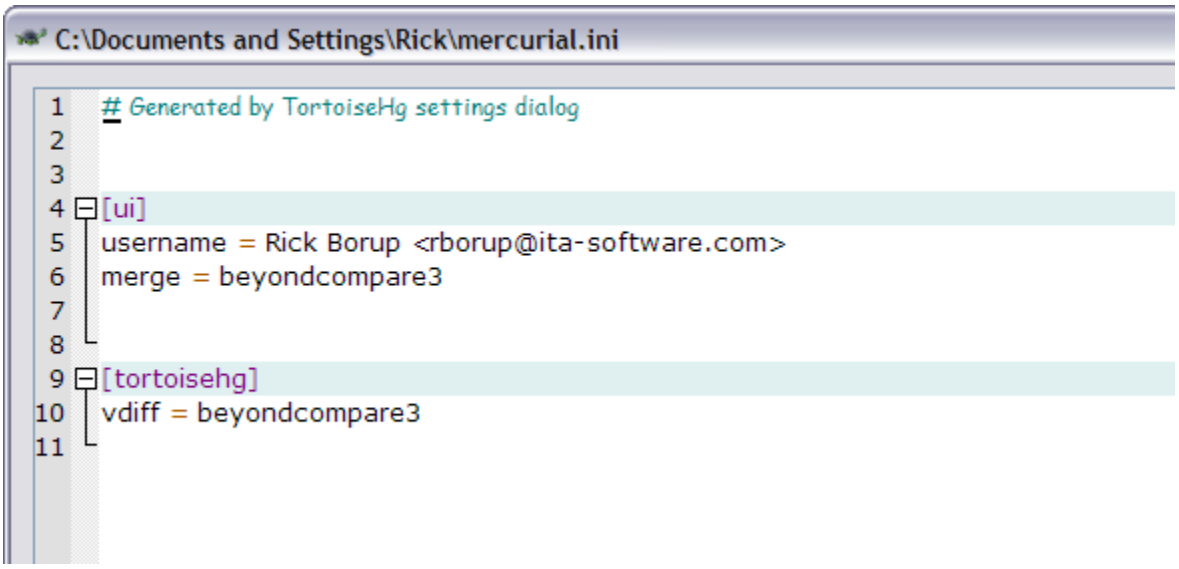


Figure 5: The mercurial.ini file is organized into sections just like any other INI file.

That's about all you need to do to get started. Take some time to explore the other options and settings available in the global settings dialog.

Repository configuration

Each local repository has a set of configuration settings that are specific to the repository itself. These are stored in a file name *hgrc,* which is stored in the .hg folder along with the repository files themselves.

When a repository is open in the TortoiseHg Workbench, the TortoiseHg Settings dialog has two tabs, one for the global settings and one for the repository settings. If specified, the repository settings override the corresponding global settings. If the repository settings are left unspecified, the global settings are used. Note the Edit File button in Figure 6. Like mercurial.ini, the hgrc file is a standard text file in INI file format, so it's easy to edit. However, it's about the *only* file in the .hg local repository folder you should ever consider editing manually.

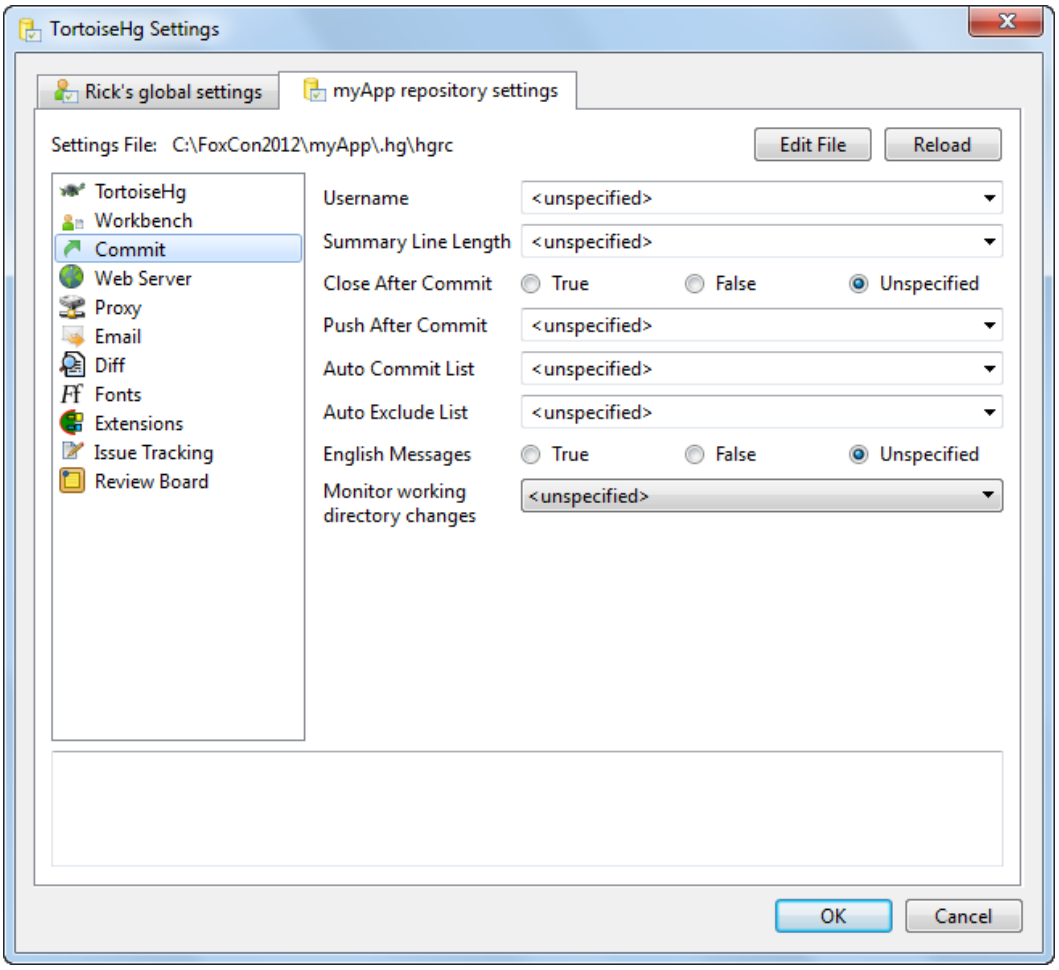


Figure 6: Repository settings are stored in the *hgrc* file located in the *.hg* subfolder. Note the two tabs, one for global settings and the other for repository settings.

TortoiseHg Workbench configuration

The TortoiseHg Workbench application can be configured according to your preferences.

Repository registry options

The Repository Registry, which occupies the left side of the Workbench app, can be made visible or hidden. By default, it shows the names of the repositories you've added to it, taken from the name of the corresponding project's root folder.

While the desired repository can usually be identified by its name within the tree view, it is sometimes helpful to be able to see the physical location of the repository on the local file system. The display can be changed to show the path to each repository by going to View | Repository Registry Options | Show Paths on the main menu. To accommodate the additional information, the column containing the repository registry can be made wider by dragging its right-hand border to the right. The resulting display is shown in Figure 7

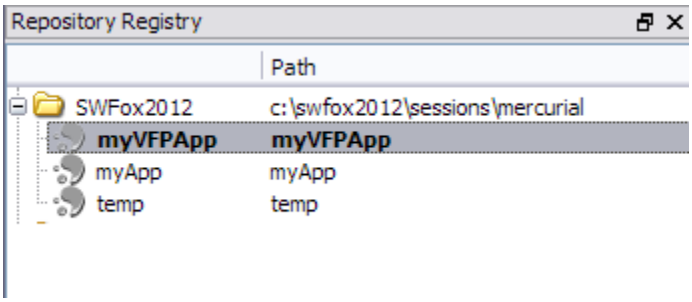


Figure 7: The Repository Registry can show the physical path to each repository.

Show output log

By default, the Repository Registry occupies the entire left column in the Hg Workbench window. However, there are two other panels which, when visible, are displayed in the area beneath the repo registry. One of these is the output log.

The output log displays the results of the operations performed by TortoiseHg, both successful and unsuccessful. The output log is hidden by default, since it's generally not necessary to view the result message from successful operations. However, when an operation is unsuccessful the output log is opened automatically to display the appropriate error information. Unless you know to look for it, it's easy to miss the output log and be left wondering what happened if an operation like a commit fails to complete successfully.

The Output Log can be made visible manually by choosing View | Show Output Log from the main menu. There is also a tool on the dock toolbar (see Figure 2) to show or hide the output log.

Choose log columns

By default, the revision history area of the TortoiseHg Workbench displays the columns shown in Figure 2, but additional information in available and can be displayed in optional columns. To choose the columns you want to display, right-click on the column headers area of the revision history to bring up a context menu, then mark or unmark the check boxes for the desired columns, as shown in Figure 8. Note that the "Workbench Log" in the title of this dialog refers to the Revision History, not to the Output Log previously discussed.

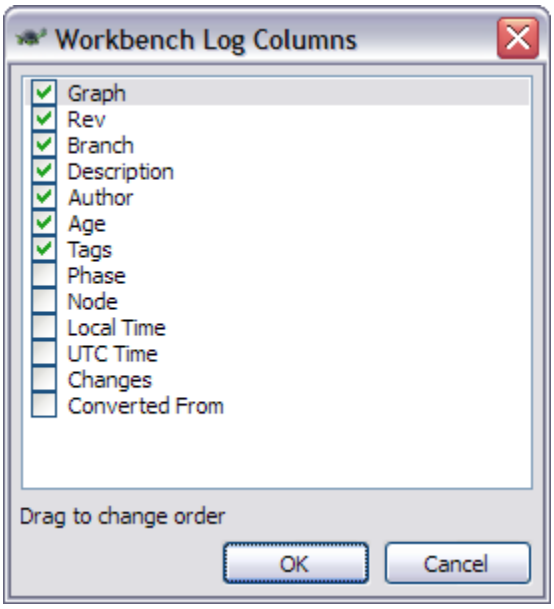


Figure 8: You can control which columns are displayed in the Revision History area of the Hg Workbench.

The TortoiseHg command line (thg)

Although TortoiseHg is primarily used to provide a graphical user interface to Mercurial via the Workbench app, it also has its own command set. These commands can be run from the command prompt by typing *thg* followed by the name of the command. You can see a complete list of these commands by typing *thg help*.

Even though run from the command prompt, many of the thg commands bring up a window you can interact with. Some of these windows are the same as, or essentially similar to, the various panes in the Tortoise Hg Workbench container. Among them are the thg commands for clone, commit, init, manifest, repoconfig, shellconfig, and sync.

The ability to control TortoiseHg from the command line opens up a great number of possibilities for customization of the interface. For example, it's possible to create a pad on the Visual FoxPro main menu with items corresponding to the various TortoiseHg actions you typically need to use when doing development work on a VFP project using Mercurial version control.

Running the *thg* command by itself—in other words, not followed by a command name—launches the TortoiseHg Workbench app. The same thing happens when you run the *thgw* command, so this is a source of potential confusion. Just remember that the two are not the same thing: *thg* is the TortoiseHg command line tool, which can be used with any of the valid TortoiseHg commands, while the only thing the *thgw* command does is launch the Workbench app.

Facilitating experimentation and parallel track development

If you're reading this paper you probably don't need to be convinced of the advantages of using a version control tool, but it may still be worthwhile to review some of those reasons here. Two of the main benefits are providing a way to revert to a previous version of a file if things go awry, and facilitating team development efforts by enabling the sharing and merging of changes among developers.

In addition to these benefits, a version control system affords developers greater freedom to experiment with changes without the risk of breaking something in the release version of a project. It also facilitates parallel track development in projects where for example one team of developers may be responsible for developing new features while another team is tasked with handling hot fixes for the release version. In some cases there may even be multiple release versions (1.0, 2.0, etc.), all of which need to be supported and maintained.

Mercurial provides several features for working with these kinds of situations.

Tags and bookmarks

A *tag* is a descriptive word or phrase that identifies a changeset in a way that's meaningful to the project's developers. Tags are often used to identify significant milestones in the development process, such as for "Release version 1.0" or "Release 2.0 Beta 1". When working on projects with a lengthy revision history, tags make it easier to identify and select the desired revision than if you had to rely on the revision numbers and commit messages alone. Tags can be used in place of revision numbers in many Mercurial commands such as update and diff.

Setting a tag in Mercurial is easy. Right-click on the desired revision in the Hg Workbench Revision History panel and choose *Tag* from the context menu. This opens the small dialog window where you can type in the desired tag as a word or short phrase. Figure 9 shows this dialog ready to add a tag for "Release version 9.1.101" to revision 1.

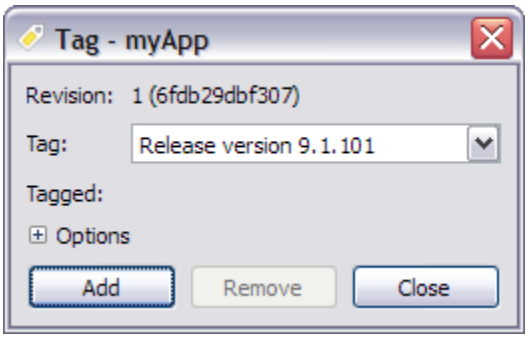


Figure 9: Tags are used to describe a particular revision with a meaningful word or phrase.

To complete the process, click the Add button. The dialog does not close automatically, giving you the opportunity to add other tags if you want to. Click the Close button when you're done.

One thing that's different about Mercurial than other version control systems is that creating a tag also creates a new changeset, which is added to the revision history. Figure 10 shows a revision history before any tags have been added.[2] Revision 1 is selected in preparation for adding a tag.

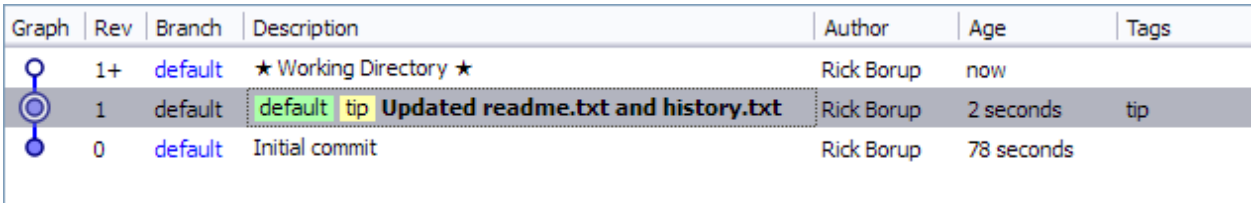


Figure 10: A revision history with revision 1 selected. No tags have yet been added, except for the *tip* tag which is added automatically by Mercurial and floats with the tip revision.

After adding the tag to revision 1, it can be seen in the revision history, which shows *Release version 9.1.101* highlighted in yellow (see Figure 11). Note that the highlighting is provided by TortoiseHg – it's not something I added for illustration purposes.

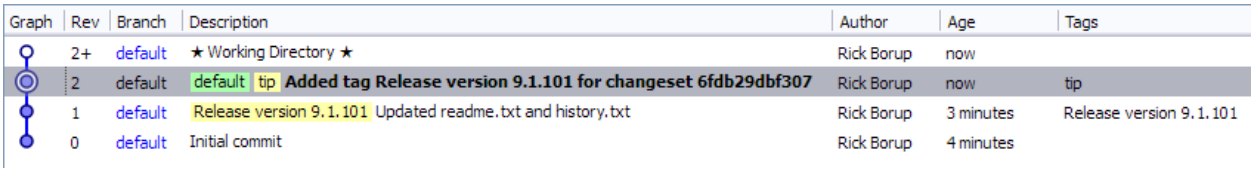


Figure 11: In Mercurial, adding a tag creates a new changeset.

The revision history also shows a new changeset, identified as revision 2. The description of this revision, which was automatically generated by Mercurial, includes the tag name supplied by the developer along with the changeset ID to which the tag applies. Note that revision 2 has now become the tip revision, and that the automatic *tip* tag has followed it.

Bookmarks are similar to tags, but they're less permanent and more portable. Bookmarks can be added, deleted, and moved around within your local repository as desired. Think of a bookmark as a sticky note you can affix to a revision. A bookmark is typically used to "mark your place" in a revision history or to record notes about a specific revision.

A bookmark associated with the tip revision follows the tip as you make new commits. Unlike tags, bookmarks are scoped to the local repository and are not included when you push changes to a remote repository.

[2] Mercurial automatically adds a tag named *tip* tag to the tip revision. This tag floats with the tip.

Branching

Branching provides a way of separating development work into two or more tracks. Mercurial has a built-in branching mechanism that enables multiple branches to co-exist within the same repository along with the ability to merge the changes from one branch into the code from another.

How much or when to branch is entirely up to you as the developer, subject to whatever constraints may be necessitated by the nature of your project or the requirements of your team. Some people create a branch for nearly every new feature and then merge it back in to the default branch for release. Others prefer to create a branch only when embarking on more extensive changes that are likely to span a longer period of time, in order that the work can be fully completed and tested without affecting the release branch.

Branching within a local repository

Branches within a given repository can be explicitly created by the developer, but can also be automatically created as the result of pushing changes to or pulling changes from a remote repository. Branches created by the developer should be given a meaningful name such as "test branch" or "release 2.0 dev branch", and are referred to as *named branches*.

Every repository has a default branch to which all changes are committed unless a new branch is created. When a new branch is created, changes are committed to that branch without affecting the revision history of the default branch. Most of the time, the intent is to pursue some line of development in the branch and then either discard it (if it was strictly a test) or merge it back into the default branch. The latter is done by first updating the working copy to the tip revision of the default branch and then merging in the changes from the branch. In some cases it may be necessary to create separate branches for different versions of a project and to maintain them indefinitely, with no intention of ever merging them. Either way, the branching mechanism is the same.

"Branching" using multiple local repositories

Another way of doing multi-track development is to use multiple local repositories, with each one starting out as a clone of the original. Cloning creates an entirely new local repository, so this is a completely different approach than creating a branch within the same repository. Conceptually, though, both approaches provide a way to split development work into two or more tracks.

After creating a clone of the main repository, run the update command to create the files in the working directory. Then do your experimentation or new feature development on the code in that working directory and commit your changes to the cloned local repository as you go along. If the purpose of the clone was strictly for experimentation or testing, you may decide to simply discard the whole thing when you're done. On the other hand, if the purpose was to develop and test a new feature for inclusion in the project, you can pull those changes back into the main repository when they're ready to go.

Patching

A patch is a file containing essentially the same information as a changeset, only in external form. Mercurial enables you to create and export patch files, which can then be shared with other developers who in turn can import them into their own local repository. The "other developer" doesn't need to be a different person – you might want to create a patch from one line of development (one repository) and import it into another line of development (another repository) for the same project, both of which you're working on yourself.

Creating and exporting patches

TortoiseHg makes it easy to create and export patches. From the TortoiseHg Workbench, select the desired revision in the revision history, right-click, and select Export | Export Patch from the context menu, as shown in Figure 12.

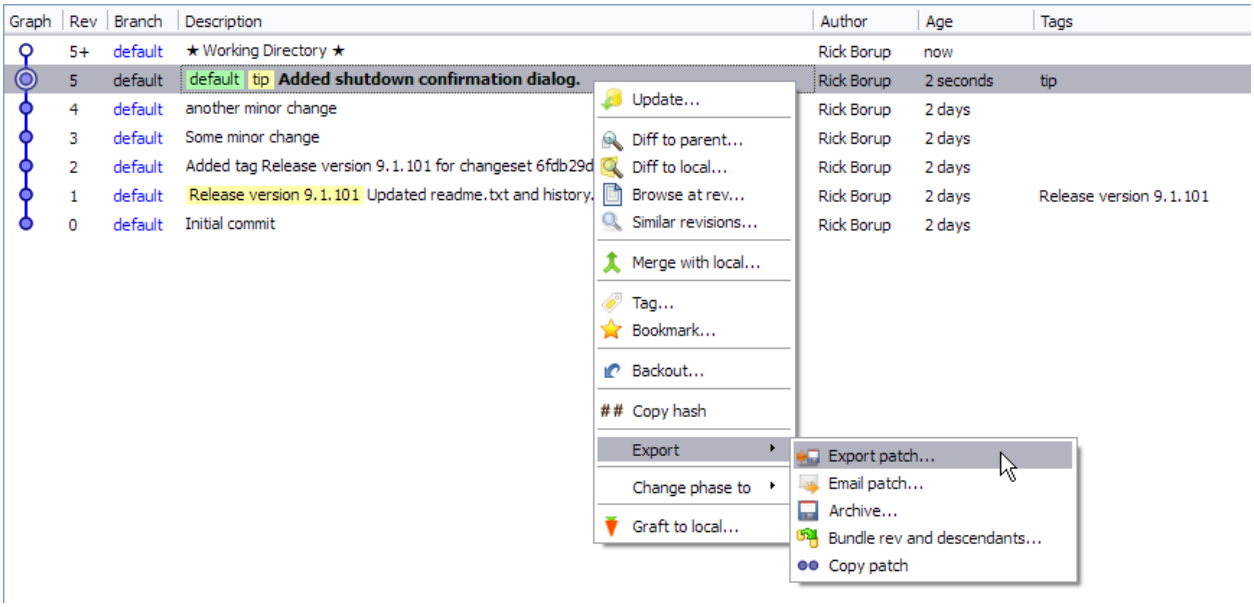


Figure 12: You can create a patch file from a revision via the context menu.

Patch files have a .patch file name extension and are written to the project's root folder. The default file name is the revision number, but you can change this to something more descriptive if you want to. Figure 13 shows the confirmation dialog displayed after creating a patch from revision 5 and writing it to a file named ShutdownConfirmation.patch.

Note that all changes associated with the selected revision are exported to the patch. If you want a patch to reflect only the changes to a particular file, make the changes to that file and do a commit to create a new revision. In other words, you can control the granularity of your patches by how frequently you commit changes to create new revisions.

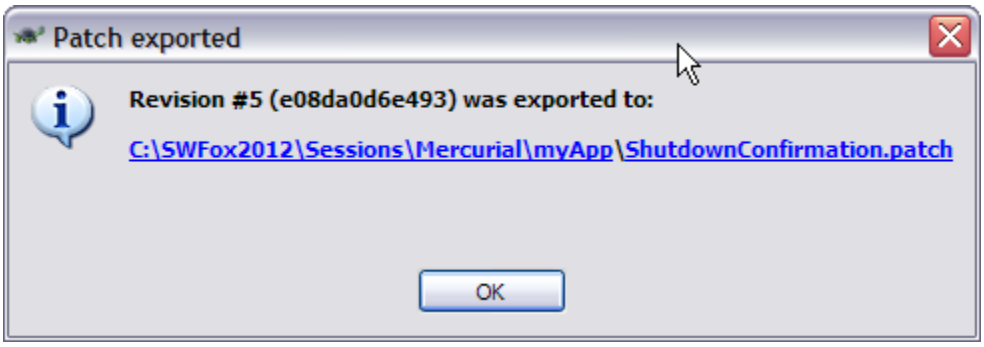


Figure 13: Patch files are written to the project's root folder and have a .patch file name extension.

Patch files are plain text files in unified diff format. This format is something you normally don't see when working with TortoiseHg, although you would see it routinely if you were working with Mercurial from the command prompt. Mercurial understands this format, so you don't have to, but here's what a patch file looks like if you're curious:

```
# HG changeset patch□# User Rick Borup <rborup@ita-software.com>□# Date 1348607565 18000□# Node ID
e08da0d6e493c39d8f4dedcded7bac1aefdaa0cb□# Parent  60ec1fed891cc520655a5ff5edf57aa84ce70bd2□Added shutdown confirmation
dialog.□□diff -r 60ec1fed891c -r e08da0d6e493 prgs/clsmyapp.prg□--- a/prgs/clsmyapp.prg Sun Sep 23 15:38:18 2012 -0500□+++
b/prgs/clsmyapp.prg     Tue Sep 25 16:12:45 2012 -0500□@@ -577,6 +577,10 @@□ *   Comments:
 *-----------------------------------------------------------------
 PROCEDURE Shutdown() as VOID
+IF MESSAGEBOX( "Are you sure?", mb_yesno + mb_iconquestion, ;
+                                "Exit application") <> idYes
+       RETURN
+ENDIF
 ON SHUTDOWN                 &&      Clear the ON SHUTDOWN command.
 CLEAR EVENTS           && Terminate the event loop.
 ENDPROC && Shutdown
```

Figure 14: The patch file is stored in universal diff format.

The four lines with the + sign represent the change in this revision, namely that these four lines were added. The rest of the information in the patch file is there to provide context.

In Figure 12 you can see there is a menu option to email a patch directly from TortoiseHg. In order to do this, you first need to configure the SMTP settings in the email section of the TortoiseHg configuration dialog so TortoiseHg knows which host, port, username and password to use to send the email. If you do this, note that your email password is stored as plain text in the mercurial.ini file (for global settings) or the hgrc file (for repository settings) on your local machine, which may or may not be security concern.

Importing patches

The flip side of creating and exporting a patch is importing it back into the project. This enables developers to create and share patches by passing them around so they can be applied wherever needed.

To import a patch using the TortoiseHg Workbench, open the desired repository and select Repository | Import from the main menu. This opens a dialog where you can select the patch by browsing to it on the local file system. With the desired patch selected, the Import

dialog gives you three ways to import the patch, as you can see in the drop-down list in Figure 15: Repository, Shelf, or Working Directory.

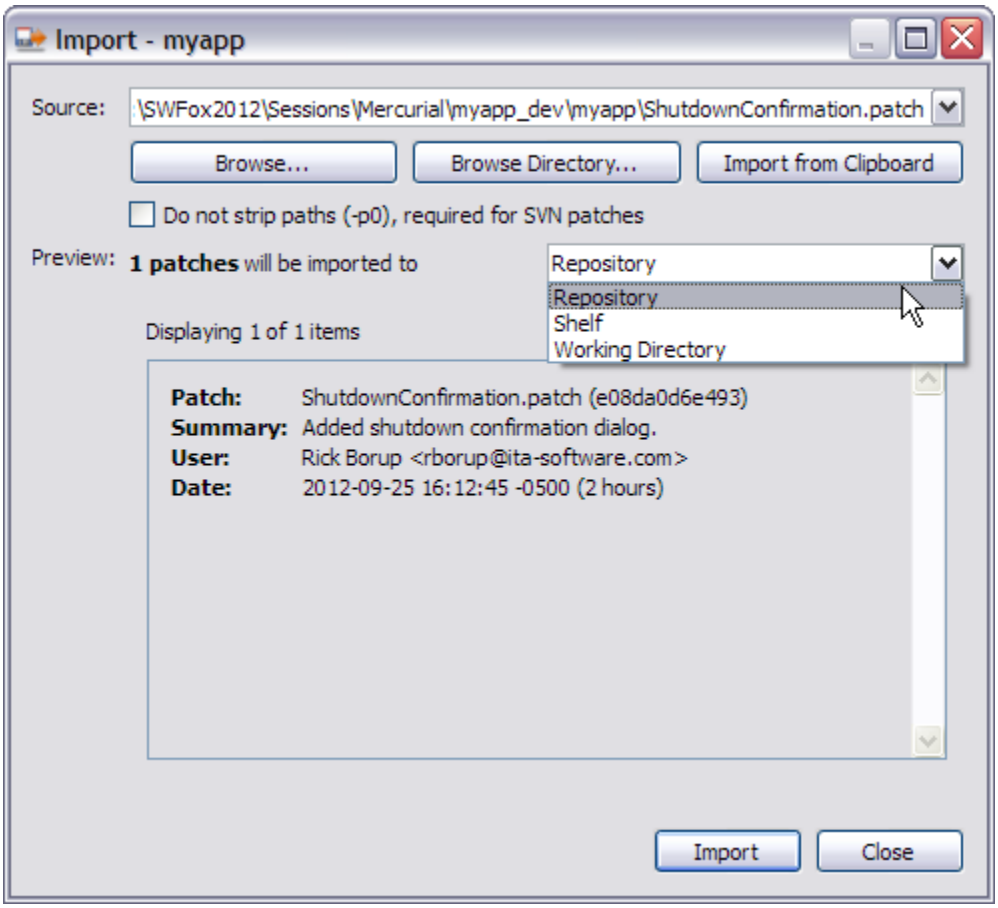


Figure 15: You can import a patch into the repository, into the working directory, or as a shelf patch.

If you import the patch into the repository, TortoiseHg applies the changes to the appropriate file(s) in the working directory and does an automatic commit, which creates a new revision in your local repository. The commit message is the one used for the revision from which the patch was exported. This information is embedded in the patch file header – you can spot the commit message “Added shutdown configuration dialog” in Figure 14.

If you import the patch into the working directory, TortoiseHg applies the changes to the appropriate file(s) in the working directory but does not do an automatic commit. This is the same as if you had made the changes yourself, and results in the file(s) being in a modified file state in the working directory. You then need to commit those changes to your local repository.

If you import the patch to the shelf, TortoiseHg creates a new shelf with the name of the patch file and stores the patch there. The file(s) in your working directory are not updated and no changes are committed to your local repository. Shelf patches are discussed in the next section.

The shelve tool

Shelving refers to the concept of setting aside a set of changes for possible later use, so named because it's equivalent to putting a physical object on a shelf. TortoiseHg provides the *shelve tool* for this purpose. The shelve tool is available only from TortoiseHg – it is not part of core Mercurial.

Open the TortoiseHg Shelve tool by selecting Repository | Shelve from the main menu. Shelving operates on uncommitted changes to files in the working directory. Files in a modified state in the working directory are listed in the upper left panel, as shown in Figure 16. This example shows an uncommitted change in the Shutdown() method of the clsMyApp program file. In the shelve tool, the change is displayed in standard diff format, same as for a patch. The change in this example are the four lines shown in green; the plus signs to the left of each of these lines indicate they were added.

The shelve tool window can be resized both vertically and horizontally. In addition, the center divider can be moved left and right, making it easy to adjust the display for maximum readability.

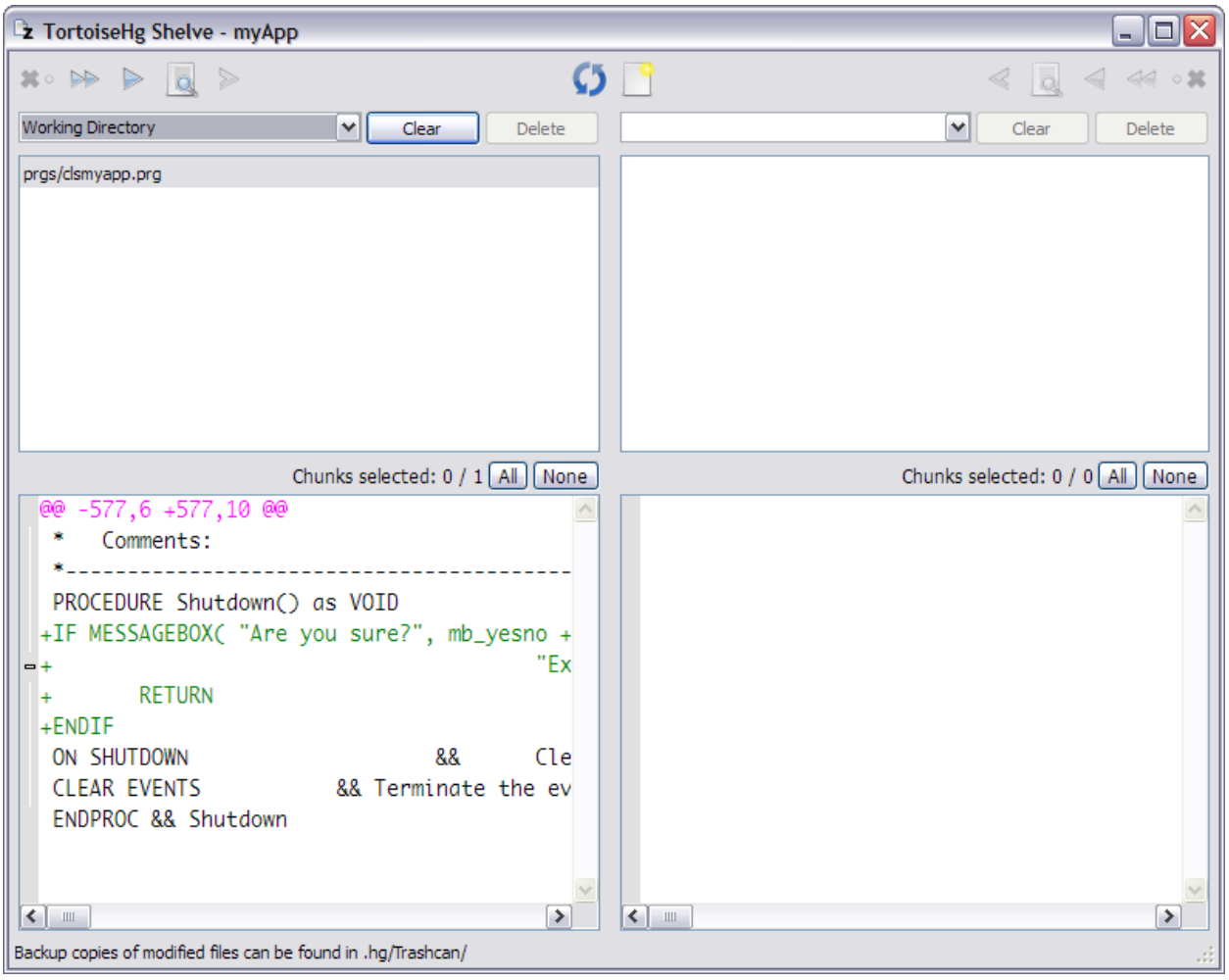


Figure 16: The shelve tool lists files with uncommitted changes in the upper left panel. The changes in the selected file are shown in the lower left.

The shelve tool works on "chunks" of code, from which it creates patches that can be shelved. These are referred to as shelf patches. Although it's difficult to see in Figure 16, notice the small minus sign between two vertical lines along the left margin of the lower left pane where the code is displayed; these vertical lines mark the upper and lower boundaries of the chunk. Clicking the minus sign changes it to a plus and selects the code in that chunk, which is now displayed with a new background color as shown in Figure 17.

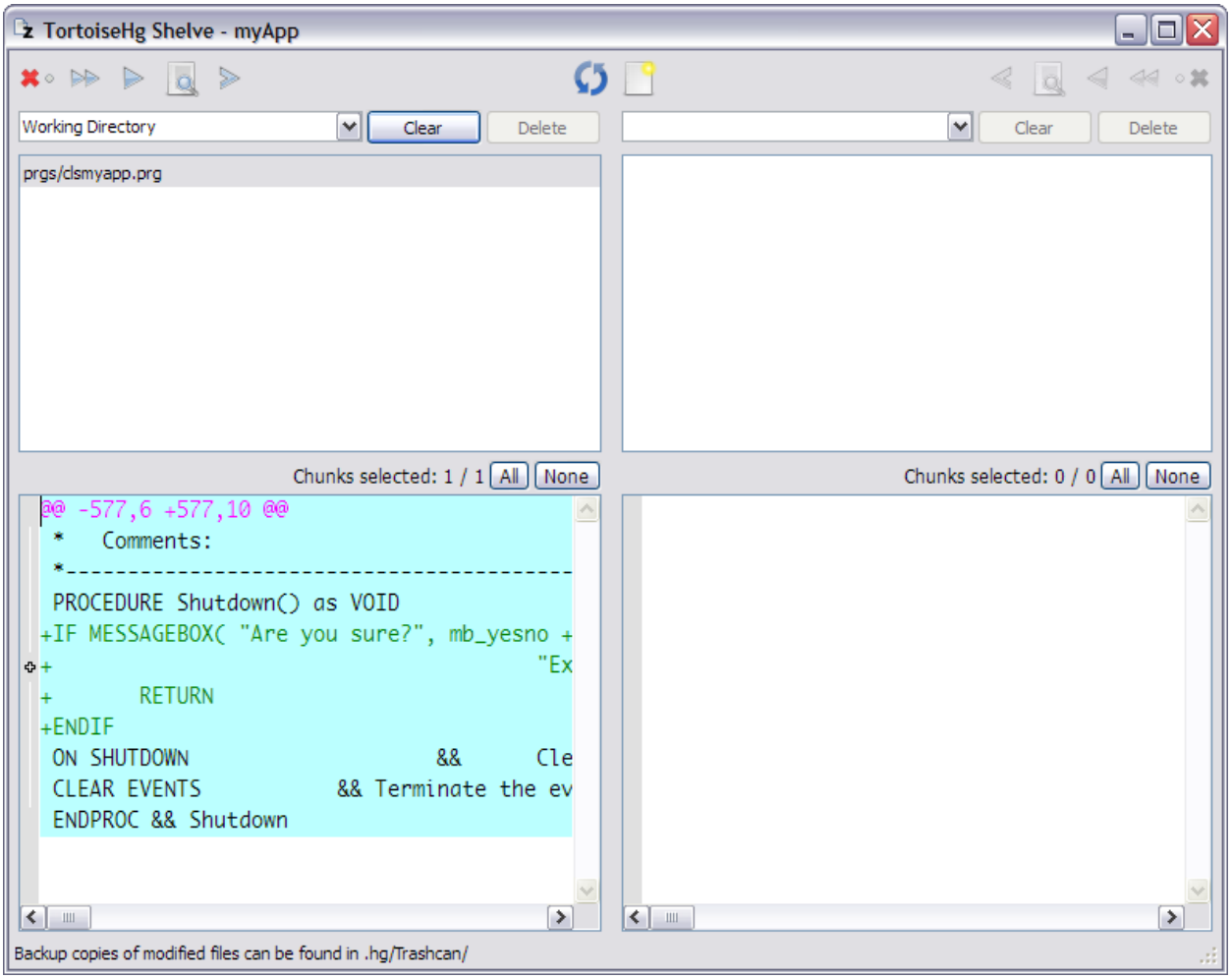


Figure 17: The selected chunk of code is displayed with a background color.

The shelve tool has two sides. The left side shows the uncommitted changes in the code, while the right side shows the existing shelf patches. Each side has its own toolbar at the top, which are mirror images of each other. The toolbar on the left side is used to create shelf patches from the selected chunks of code on the left, while the toolbar on the right side is used to apply selected shelf patches from the right to the code file on the left. Other

tools on these toolbars provide additional functionality. The toolbar in the center has two tools, one to refresh the display and the other to create a new shelf.

With the appropriate chunk selected on the left side, click the rightmost arrow on the left side's toolbar, as shown in Figure 18, to move the chunk to the right side and create a shelf patch.

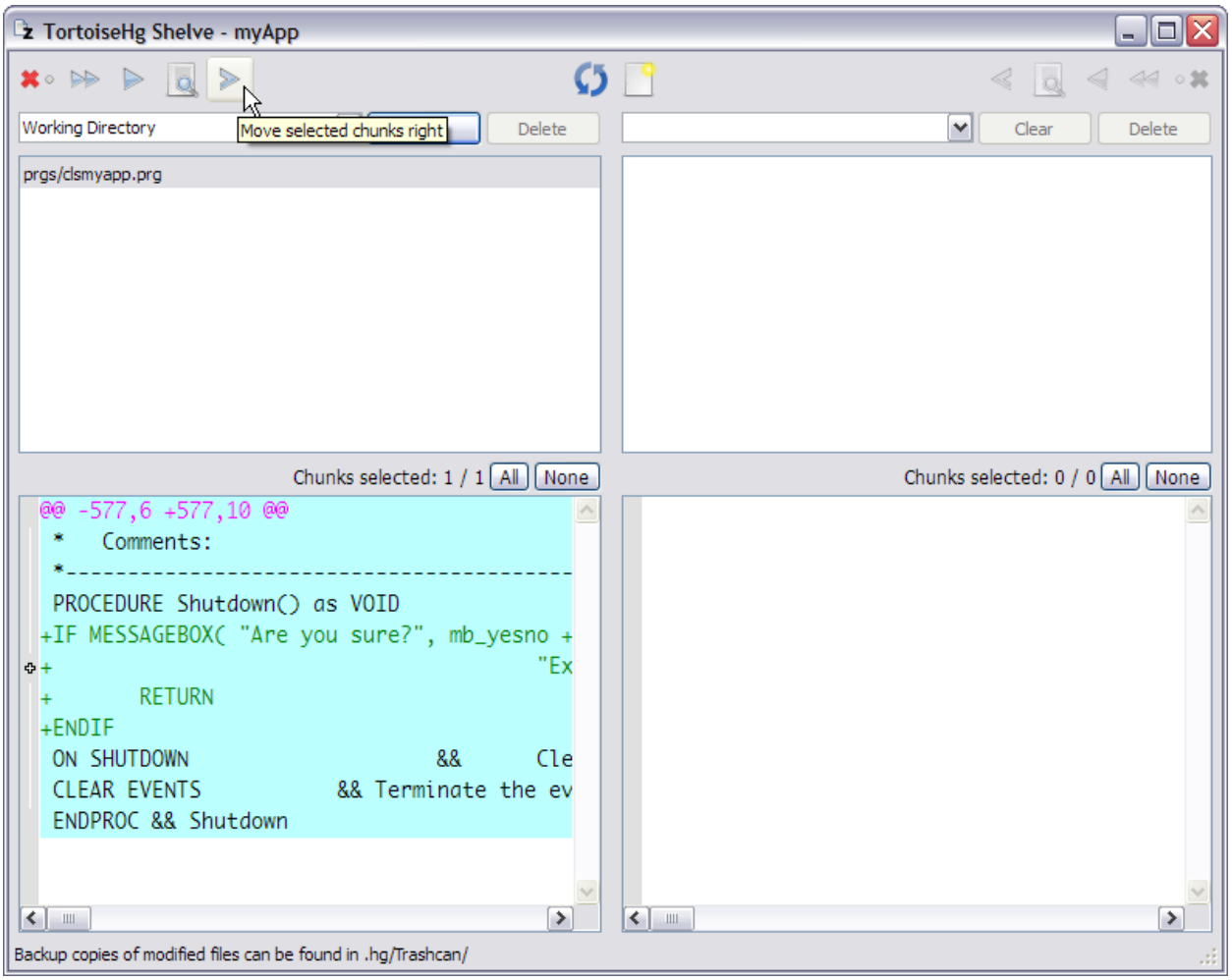


Figure 18: Click the right-most arrow on the left side's toolbar to create a shelf patch by moving the selected chunk of code to the right.

In this example the selected chunk was the only uncommitted change to the only modified file in the working directory, so the shelve tool prompts to remove all file changes from the left side. After replying Yes, the shelve tool now shows nothing on the left side and a new shelf patch on the right side, as illustrated in Figure 19. Because no shelf existed before creating this patch, a new shelf was automatically created and assigned a name, which can be seen in the field at the top of the right side.

The change is now stored in the shelf in the form of a patch, and the selected chunk has been removed from the original file. If the selected chunk was the only change to that file, the file reverts to an unmodified state as far as Mercurial in concerned. Either way, you can now go on to make other changes to the project as if the patch didn't exist.

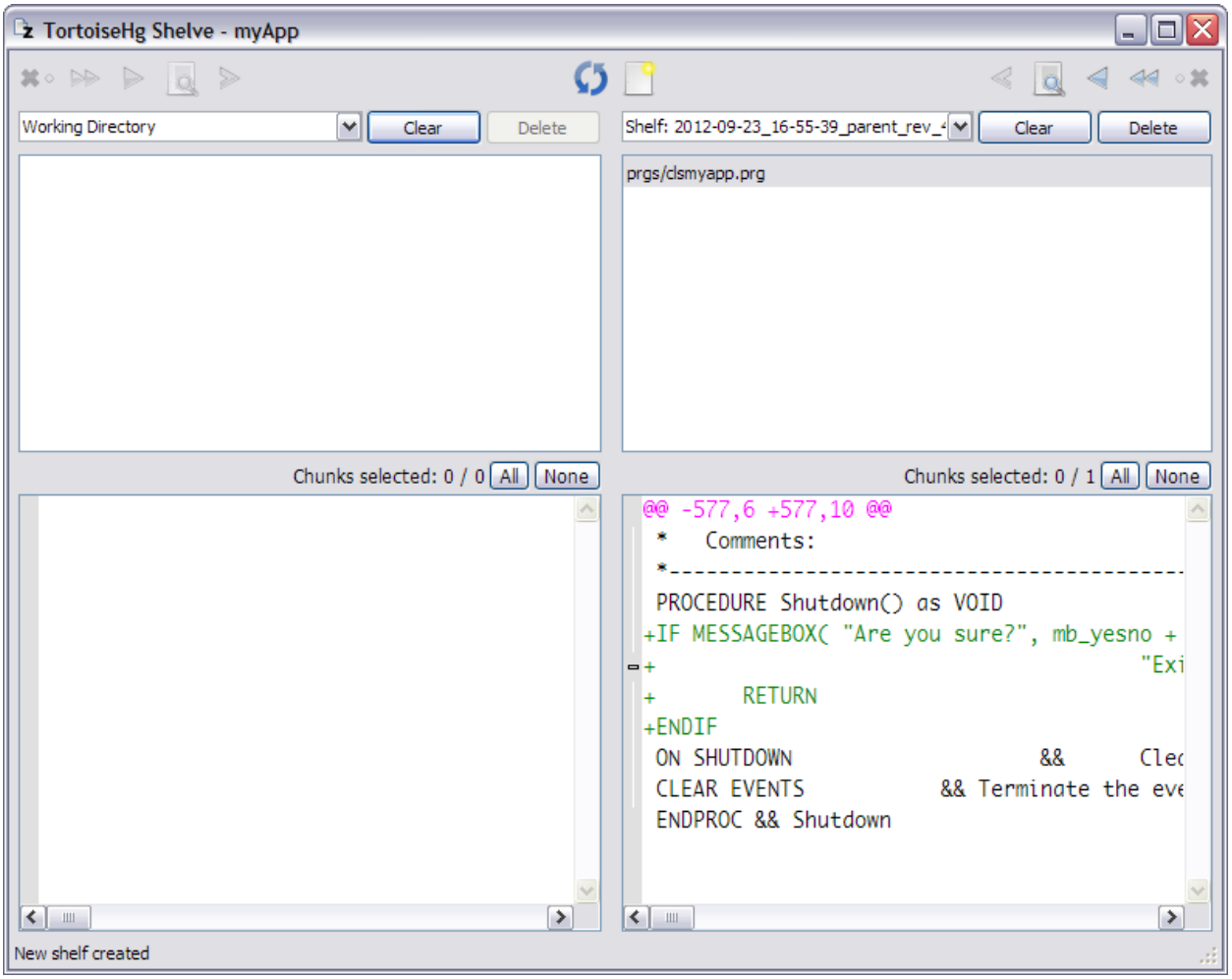


Figure 19: The newly created shelf patch is now shown on the right.

If and when the time comes to restore the chunk, you can reverse the previous process and apply the shelf patch back to the source code file. Note that if the source code in the affected area has diverged significantly from its state at the time the shelf patch was created, reapplying the patch may result in a merge conflict. If this happens, TortoiseHg displays a message as shown in Figure 20 and prompts you to edit the file and resolve the conflict.

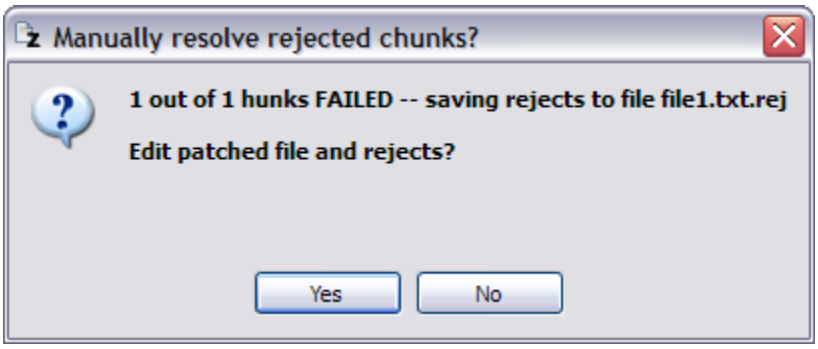


Figure 20: Re-applying a shelf patch can result in a merge conflict if the source code has changed significantly since the patch was created.

Close the shelve tool when you're done working with shelf patches, and open it again the next time you need it. Existing patches will still be there, ready for you to use. If more than one shelf is being used, you can select the desired one from the drop-down list at top right.

Mercurial queues

The ability to create, export, share, and import patches offers an alternative way for multiple developers to collaborate on a project without using a centralized repository. Some teams like to work this way because of its agility.

While it's relatively easy to work this way when the number of patches is small, the task can quickly become problematic as the number of patches grows. Mercurial comes with an extension called Mercurial Queues, or MQ for short, which is designed to deal with this "patch management problem."

MQ is designed to make it much easier to group, stack, track, and manage large numbers of patches. It's a fairly complex topic that's beyond the scope of this paper. There are two full chapters on the subject in the book Mercurial: The Definitive Guide, which is the best place to get started learning more about it.

If you want to experiment with MQ, note that it's implemented as an extension to Mercurial and must be enabled before it can be used. To do this, select the Extensions group in the TortoiseHg configuration settings dialog and mark the check box for "mq". A number of new commands and menu choices then become available.

Archiving

TortoiseHg enables you to create an archive of the files in your working copy, reflecting their state as of any point in the revision history. This can be useful not only as a backup but also as a way to send the project files to other developers who may not have access to the repository.

To create an archive, right-click on the desired revision in the revision history panel of the TortoiseHg Workbench and select Export | Archive from the context menu. The Archive

dialog provides several options. Start by specifying whether you want the archive to include all files or only the files that were modified in the selected revision. Then choose the desired archive format from among the several available choices. The popular zip format is probably the most useful for Windows developers.

TortoiseHg provides a default name and location for the archive. The default file name is based on the selected revision number. For example, if you select the zip format, the default file name for an archive from revision 8 is *myApp_8.zip*. If the revision has a tag associated with it, you can tell the archive tool to use the tag instead of the revision number, as in *myApp_Release 9.1.101.zip*. The default target location for the zip file is the *parent* of the working directory. Within the zip archive, the files are stored in a root folder named for the revision or tag, for example a folder named *myApp_8* or *myApp_Release 9.1.101*. You can override the archive's default file name and/or location if you want to.

The archive contains the files and folders from the working directory, along with a text file named .hg_archival.txt whose contents describe the revision from which the archive was created. The archive does not contain the .hg subfolder – in other words, it does not include the repository itself.

Working with remote repositories

As with any DVCS, the full power of Mercurial is only fully realized when you start working with remote repositories. In the simplest case, a solo developer can use a remote repository both as a backup to the local repo and also as a way of sharing changes when doing development work on the same project using two or more machines. In more complicated scenarios, multiple developers may push and pull changes from one or more remote repositories or even directly amongst themselves.

A remote repository is any repository other than the local one that is related to the same project. Therefore the definition of what constitutes a remote repository depends entirely on the perspective of the person making the determination: my local repository can be your remote repo, and vice versa. While developers can make their local repositories available to other developers directly, it's more common to use one or more centralized remote repositories.

There is no difference between a local repository and a remote repository, other than who uses it and how it's made available for sharing. This section discusses several ways in which remote repositories can be shared.

Accessing via the local file system

For developers working with external hard drives and/or networks with mapped drives, Mercurial repositories can be shared via the local file system. For example, a solo developer might create a folder structure on an external hard drive to serve as a location for remote repositories, or a team might do the same on a file server or network attached storage device.

The folder you use as the root for a remote repository can have any name – Mercurial doesn't care. It makes sense to me to give them the same name as the working directory for the project they're related to, but that's not a requirement. I use *HgCentral* as the name of the root folder for my remote repositories. I create an HgCentral folder wherever I want one, then create subfolders to hold the remote repository for each of the various projects I have under source control. This is simply my own convention – others are equally valid.

You can't push changes to a remote repository unless a repository already exists in the target location. In other words, you can't push changes to an empty folder. The remote folder must contain either a newly initialized .hg subfolder, or have an existing .hg subfolder containing a repository for the same project.

There are two ways to initialize an empty folder to serve as a remote repository. The first is to initialize a new repository either from the Windows Explorer context menu or by running the *hg init* command from the command prompt in that folder. Either way, a new .hg subfolder is containing the necessary structure for the new repository.

The other way to initialize an empty folder to serve as a remote repository is to create a copy an existing repository that's already in use for the desired project. This can be done either by using Mercurial to clone the existing repository into the new location, or by using Windows to copy the entire .hg subfolder from the local repository to the remote folder.

Of the two, the preferred approach would be to initialize a new repository and then push changes to it.

The .hg subfolder is usually the only thing in the root folder of a centralized remote repository. Centralized remote repositories do not need to have any files in the working directory because no development work is done there and all changes are pushed rather than committed.

Where to put the remote repository

For access via the local file system, a remote repository could be located on any drive accessible as a drive letter from the local machine.

In the simplest case, a remote repository can be created in a folder on the local hard drive. For example, I might create C:\HgCentral and under that have subfolders for Project A and Project B. Once repositories are initialized in those two subfolders, I can use TortoiseHg's *sync* tool to push Project A's changes to C:\HgCentral\ProjectA, and Project B's changes to C:\HgCentral\ProjectB. While this approach may be appropriate for the solo developer or team member wanting to use a remote repository as a local backup, it does not provide for sharing the repository with other developers or even with other machines.

The next step up might be to create HgCentral on an external hard drive. Although still not shareable by other machines or other developers (assuming the external hard drive is

connected to the local machine via a USB connection), this approach has the advantage of protecting the central repository from loss due a crash or corruption of the local hard drive.

When one is available, a network attached storage device (NAS) provides a convenient place for a centralized remote repository. Any machine with a mapped drive connection to the NAS can push to and pull from the repository, making this an appropriate solution for use in a team environment or for a solo developer working on the same project from more than one machine.

In situations where a local area network is in use, the file server or one of its shared resources is also a good place for the centralized remote repository. Both a file server and a network attached storage device have the advantage of being accessible by more than one user and more than one machine, while at the same time being physically located on a device other than the developer's hard drive or locally attached external hard drive.

Finally, the cloud provides opportunities to place a centralized remote repository on a virtual storage device. Not all cloud storage solutions are workable, because you need the ability to access the storage device as a mapped drive letter from your local file system. Services like Jungle Disk make this possible, with the cloud drive typically being mapped as J: to your local file system. Once in place, you can push to and pull from a repository on Jungle Disk just as if it were a on a local or network resource. One obvious advantage to the cloud solution is that the storage location is entirely off premise, insulating it from disasters such as fire or flood that could wipe out an entire office along with all its local and network resources.

A word about Dropbox

If you use Dropbox, you might be tempted to use your Dropbox folder as the location for a centralized remote repository. Dropbox is a fine service – I use it myself every day – but I'm not sure I'd be comfortable using it for this purpose.

Dropbox works by creating a special folder on your machine. Files placed in this folder or its subfolders are quickly replicated on Dropbox's servers and then pushed out to all other machines sharing the same folder. This makes it ideal for synchronizing the contents of folders among two or more machines.

I have two concerns about using it as a centralized remote repository for Mercurial. One is that each machine would actually be working with its own copy of the remote repository, namely the one in its local Dropbox folder, rather than interacting with a single shared repository in a common remote location. The other reason is that, although nearly instantaneous, there is still a small lag between the time a file is updated in a local Dropbox folder and the time it's refreshed on Dropbox's server, and from there before it's replicated to the local Dropbox folders on the other machines. I don't know what would happen if multiple developers begin pushing and pulling changes simultaneously in real time.

I don't have any empirical evidence to support either of these concerns, and it could turn out that Dropbox works fine as the location for a centralized Mercurial remote repository. I'd be interested to hear if anyone is successfully using it that way.

Accessing via http

Mercurial comes with a small built-in web server you can use to serve up your repositories via http. Although limited in its capabilities, this can be a viable solution for situations where developers want to work with a remote repository but don't have mapped drive-letter access to its location.

By default, pushing to the repository via http is not allowed. If you want to allow other people to push changes to the repository you need to change a couple of configuration settings. Open the TortoiseHg settings dialog from the Workbench app and select the repository settings page. Set *Push Requires SSL* to False, and set *Allow Push* to *, as shown in Figure 21.

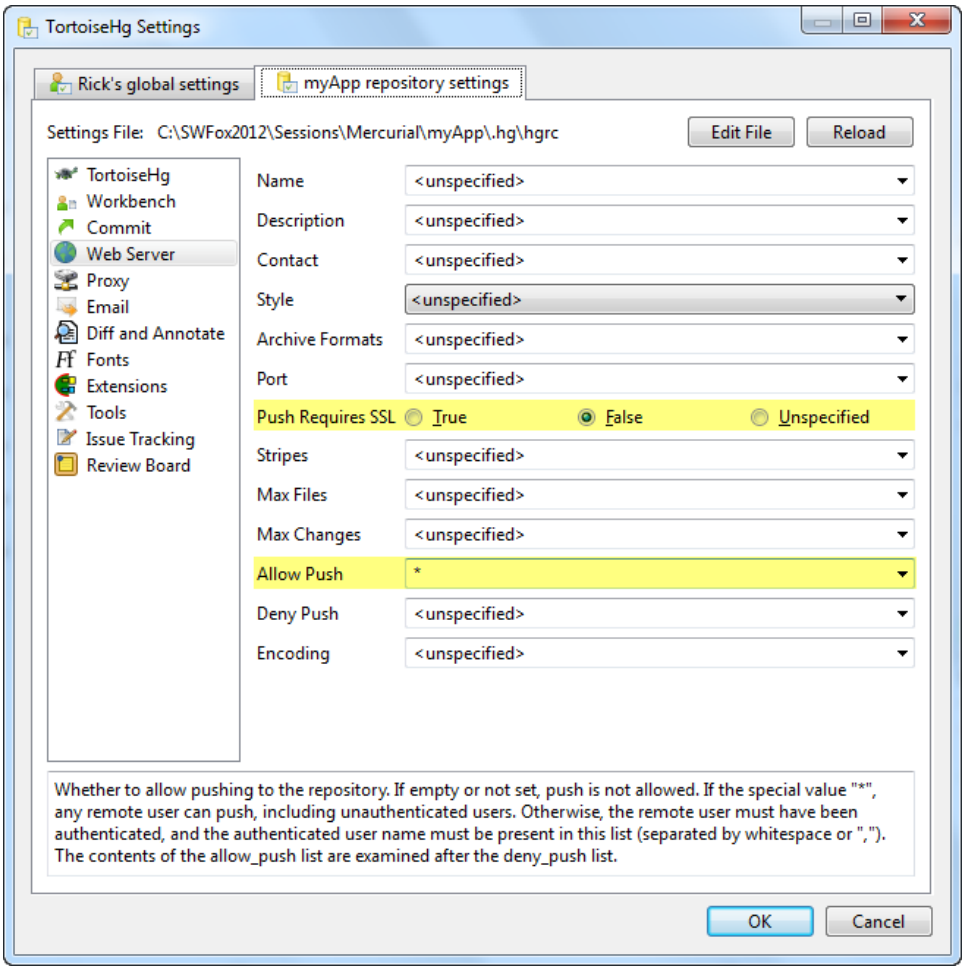


Figure 21: Change the repository configuration settings to allow people to push changes to the repository via the Web server.

Setting *Allow Push* to * opens up a big security hole, so unless your machine is isolated from the Web this setting is really only appropriate for testing in a secure environment. As you see from the notes displayed by TortoiseHg in the configuration settings dialog in Figure 21, you would normally allow only authenticated users to push changes.

You can start the Mercurial Web server by opening the desired repository in the Workbench app and choosing Repository | Web Server from the main menu, or by right-clicking on the working directory in Windows Explorer and selecting TortoiseHg | Web Server from the context menu. You can also launch the server by typing *hg serve* from the command prompt at the working directory.

When the server is running it's accessible over port 8000 by any authorized user who can access the host IP address. For personal testing, you can access the server from your browser at http://localhost:8000. The default page is the revision history, which looks like Figure 22.

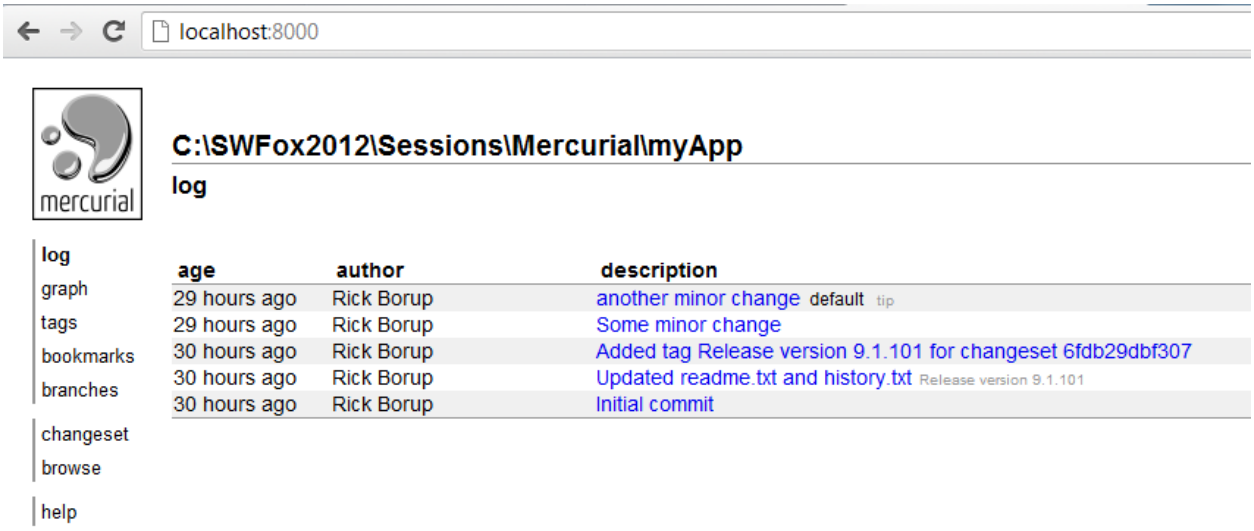


Figure 22: The Mercurial Web server is accessible over port 8000. The home page is the revision history.

The words under the Mercurial logo on the left side—log, graph, tags, etc.—are navigation links to other pages that display different information about the repository. The descriptions for each revision are also links; clicking on one displays details about that revision. Figure 23 shows the details for revision 3.

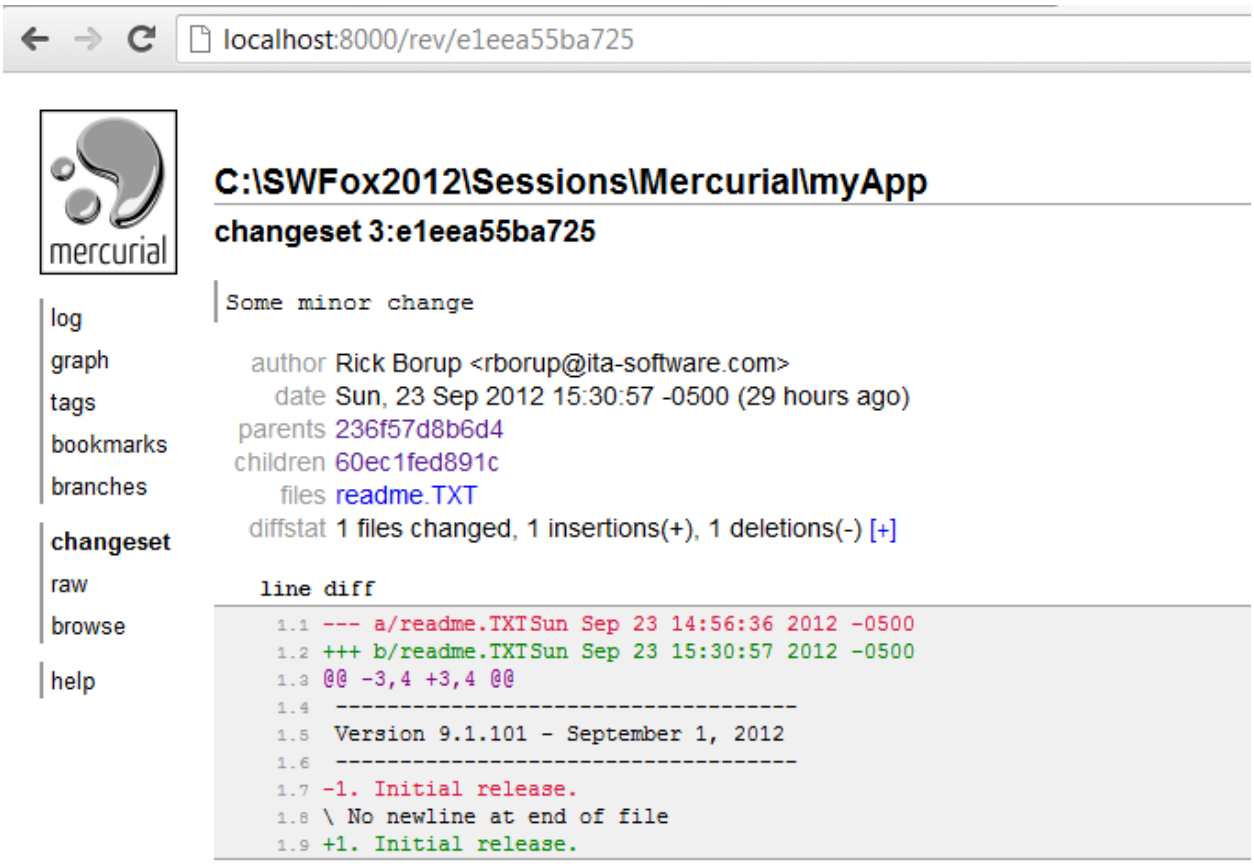


Figure 23: You can view details about a particular revision, including the diff information for the modified files.

You can use the TortoiseHg sync tool to push and pull changes from the repository via http in the same way you would using local file system access. Figure 24 shows the sync tool configured to access the repository on localhost via http.

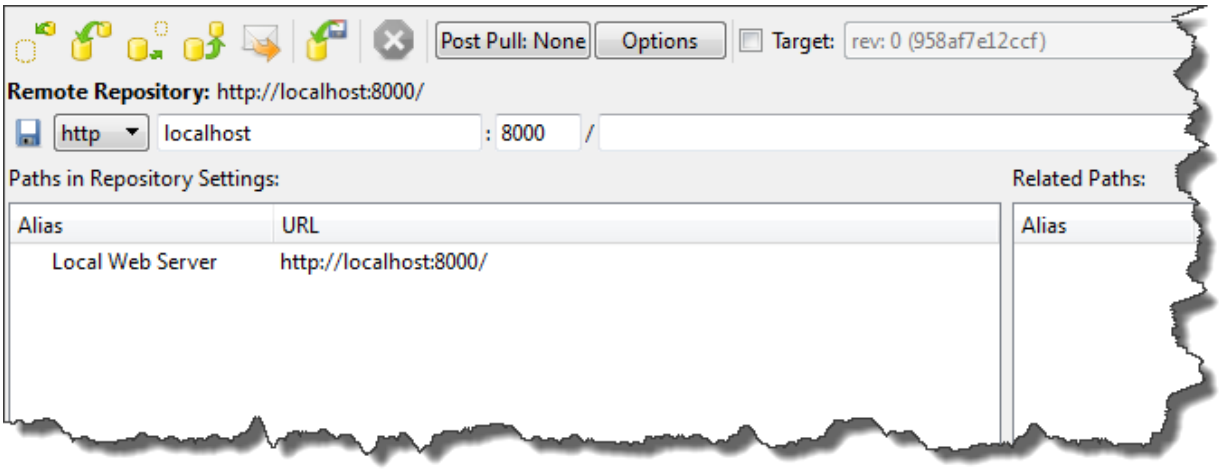


Figure 24: The TortoiseHg sync tool can be configured to access the repository on localhost, enabling push nd pull operations via http.

Using Bitbucket

Bitbucket is the most popular public service for hosting Mercurial repositories. It's public in the sense that anyone can open an account and create repositories, but that doesn't mean all repositories hosted there are publicly accessible. Your repositories can be kept private, even with a free account. Bitbucket also enables you to set permissions to control who can read, write, and act as an administrator on repositories you create.

To set up an account, go to https://bitbucket.org/ and follow the sign-up procedure. Bitbucket's free 5-user account is great way to get started. There are several tiers of paid accounts if you need to support more than 5 users.

Once you've set up your account, log in and create a repository. Give your repository a name and specify that you want to use Mercurial (the other choice is Git). There's also a field where you specify the language your source code is developed in. Surprise – Bitbucket knows about FoxPro!

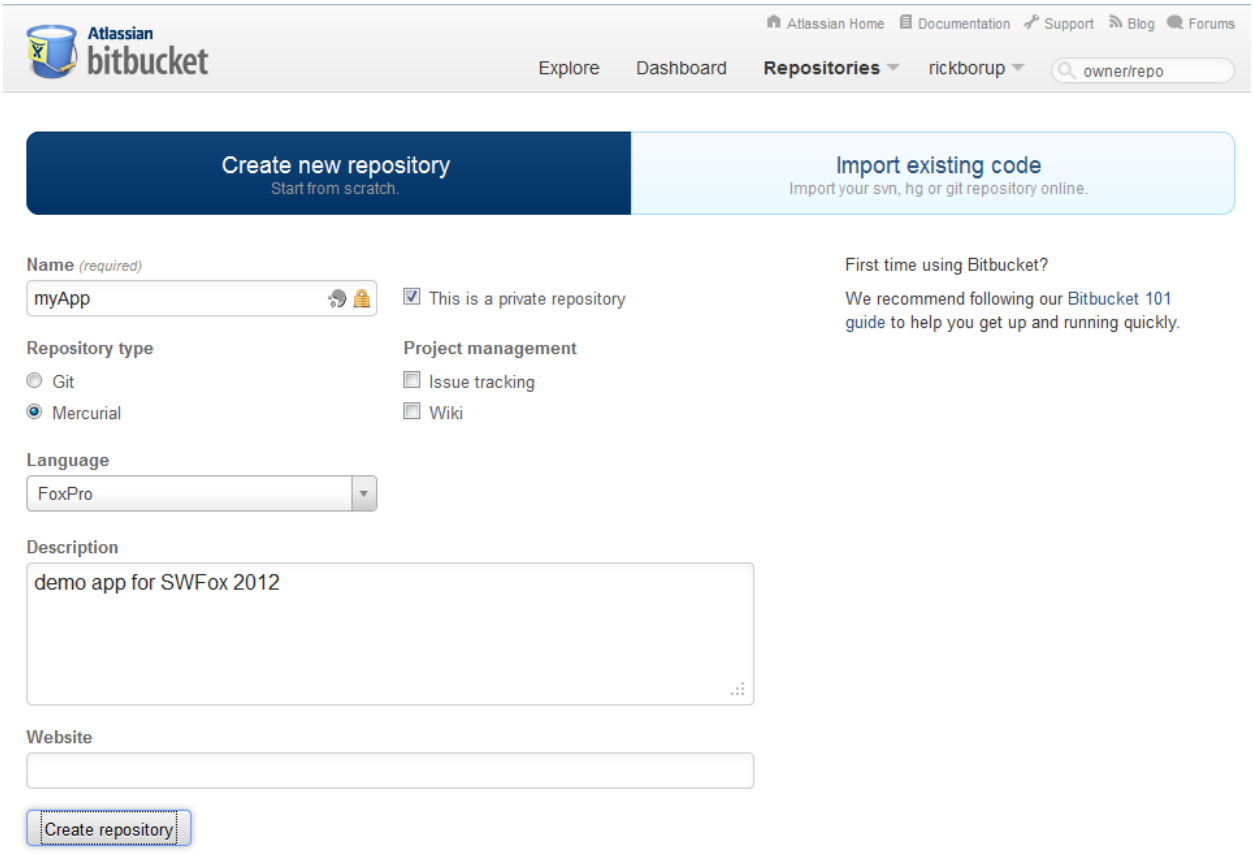


Figure 25: Specify you want Mercurial when creating a new repository on Bitbucket.

Figure 25 shows the Bitbucket web page for creating a new repository from scratch. Note the check box is marked to make this a private repository. Another way to create a new repository on Bitbucket is to import it from an existing one. This requires that the source repository be accessible via a URL, so it may not always be an option.

Once you've created the Bitbucket repository, use the TortoiseHg sync tool to add it to the list of remote repositories associated with your local repository. Bitbucket repositories are given a publicly addressable URL using secure http. In the sync tool, select *https* from the drop-down list and enter the URL as shown in Figure 26.

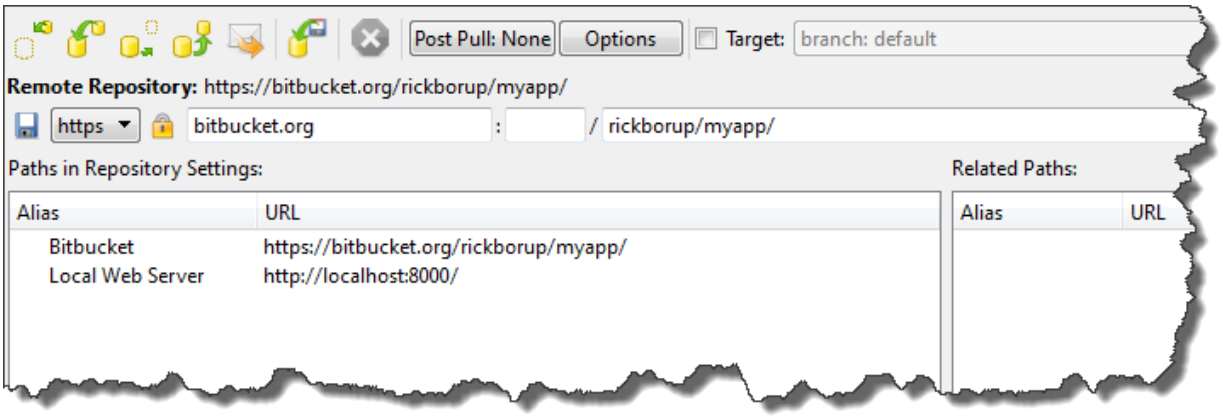


Figure 26: Add the Bitbucket repository to the list of remote repos associated with your local repository.

You can now push changes to and pull changes from the Bitbucket repository using the TortoiseHg sync tool. Bitbucket prompts you for your username and password each time you initiate an operation. You can avoid this by entering your username and password in the sync tool's security dialog, as shown in Figure 27.

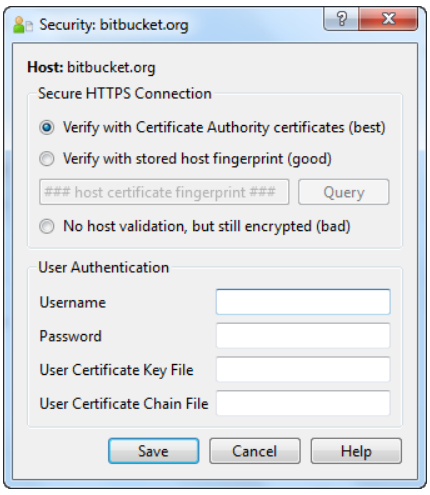


Figure 27: Enter your Bitbucket account username and password in the TortoiseHg sync tool's security dialog so you don't have to enter it every time you do a push or pull.

Troubleshooting

Mercurial is pretty easy to understand and to work with, but as time goes on the state of a repository and its relationship to the contents of the working directory can become complicated, which in turn can make it difficult to diagnose and resolve problems.

This section comprises a series of tips, tricks, and suggestions drawn from my real-world experience with Mercurial and TortoiseHg. They're loosely arranged into those that help you stay out of trouble and those that may help you get out of trouble once you find yourself in some.

Staying out of trouble

For all the benefits it unquestionably provides, a version control system does require that additional tasks be added to your daily software development workflow. One way to stay out of trouble is to establish a routine series of steps to incorporate your version control system, and then be sure you follow them all the time. Trouble often results from omitting a step without realizing you've done so.

Change > Commit > Push

The basic workflow for revisions you create is change > commit > push. After making changes to files in your working directory, commit those changes to the local repository whenever you feel it's appropriate, often when reaching a good stopping point in your daily work. If you're a solo developer and using a remote repository, you can push changes to the remote as frequently or infrequently as you want to since it doesn't impact anybody else. If you're working with a team and sharing changes via a centralized remote repository, it will be more important to push your changes to the central repository on a regular basis so other team members have access to the latest version of the source code. How often this needs to be done depends on how the team operates and how the development work is divided among its members.

Pull > Update > Merge > Commit

The basic workflow for getting changes made by others is pull > update > merge > commit. In a team scenario, or even if you're a solo developer using branches or working with multiple local repositories for multi-track development work, your routine should include a step to pull changes (if any) into your local repository and merge them into your working copy on a regular basis. Here again, how often you need to do this depends on the project and on the standard practices of the team.

Merge conflicts arise when two sets of conflicting changes have been made to the same chunk of code. Mercurial provides a way to resolve merge conflicts, so be sure you know how to use the appropriate tools to do so. Merging updates files in your working directory but does not automatically commit those changes back to the local repository, so you need

to remember to do a commit after doing a merge. Mercurial reminds you to do this, and the TortoiseHg merge wizard includes a commit step at the end, but it does not force you to.

After doing a merge, you might be tempted to hold off on the commit until after you've made your own changes. However, it's best to go ahead and do a commit immediately after the merge even if you intend to make new changes, because then you have a revision reflecting your starting point that you can revert to if necessary.

Preview your actions

I recommend always previewing what will happen before you do a pull or a push. Mercurial provides the *hg incoming* and *hg outgoing* commands for this purpose. These commands can be run by clicking the appropriate icons on the TortoiseHg sync toolbar. The *incoming* command shows you which revisions will be pulled into a repository, while the *outgoing* command shows which revisions will be pushed. Running these commands before performing the actual operation helps you avoid unpleasant surprises.

Getting out of trouble

It's not difficult to get confused. If you find yourself facing a situation where it's not clear what happened, which way to go, or what happened to your code, first take a step back and analyze the situation. The following general guidelines may help. They're followed by examples of some common problems, along with suggested solutions.

General guidelines, tools, and commands

Remember the basics: You modify files in your working directory. You commit changes to your local repository. You update your working directory from your local repository. Mercurial store changesets. You push changesets to and pull changesets from a remote repository. Revision numbers are local, changeset IDs are global.

Use the tools Mercurial makes available to help understand the situation. The graphical representation of a repository's revision history in TortoiseHg can be one of the best tools for analyzing the state of a repository and *how* it came to be in the state it's in. Here you can see the hierarchical relationship among revisions and branches, showing parent revisions, child revisions, and tips. The commit messages displayed in the description field of the revision history are one of the best tools for understanding *what* got changed in each revision. This is one reason it's important to encourage all team members to write meaningful commit messages.

Sometimes it becomes necessary to get additional information from other members of the team. The revision history shows who made each change, and when, but beyond that you may need to communicate directly to find out what was done and why. When source code conflicts are the issue, the *diff* command or the corresponding diff tool in TortoiseHg (either default one or the one you configured) are very helpful in discovering how two sets of changes may have impacted one another.

Revert, Rollback, and Backout

Use the tools Mercurial makes available to help you resolve problems and get out of trouble. Among these are the revert, rollback, and backout commands.

The *revert* command restores a file or directory to its unmodified state. It affects the working directory but not the repository. If no revision is specified, files are reverted to the state of the working directory's parent revision. Revert can be used to abandon a change or changes before committing them. Modified files are saved with a .orig extension before being reverting. Files can be reverted to an older revision by specifying a revision number. In TortoiseHg, the revert command is available on the context menu from the list of files in the commit dialog or the manifest.

The *rollback* command rolls back the last repository transaction, typically a commit. It affects the repository but not the working directory. Only the most recent transaction can be rolled back, and once completed there's no way to undo it. If the rolled back revision has already been pushed to a remote repository, or if other developers have already pulled that revision or cloned the repository, those repositories are not affected by the rollback and the revision has "escaped into the wild". For this reason, the rollback command is generally considered dangerous and should be used sparingly. In TortoiseHg, the rollback command is found on the Repository pad on the Workbench main menu.

The *backout* command backs out the effect of a previously committed revision. It affects both the working directory and the repository – files in the working directory are modified and a new changeset is created in the repository. Although useful in certain circumstances, the backup command introduces some complexities of its own. For example, a backup can sometimes cause merge conflicts which must be resolved before the new changeset can be committed. In TortoiseHg, the backout command is available on the context menu for the selected revision.

If you think you need to use any of these commands, be sure you understand how they operate. See the Mercurial documentation for more information.

Revision history tool

The *revision history* tools enable you to see which files got changed in any given revision and even what got changed within any given file. In the TortoiseHg Workbench, click on the desired revision in the revision history, then click the Revision History icon on the task toolbar as shown in Figure 28.

Figure 28: Display the revision history by clicking the icon on the task toolbar.

The task pane in the lower portion of the Workbench window then displays the revision history for the selected revision. The files modified by that revision are listed on the left. To see the history of changes for a particular file, right-click on the file name and select *File history* from the context menu. The file history dialog displays a list of the revisions that modified that file. Selecting any of those revisions displays the modifications that revision made to the file, as illustrated in Figure 29.

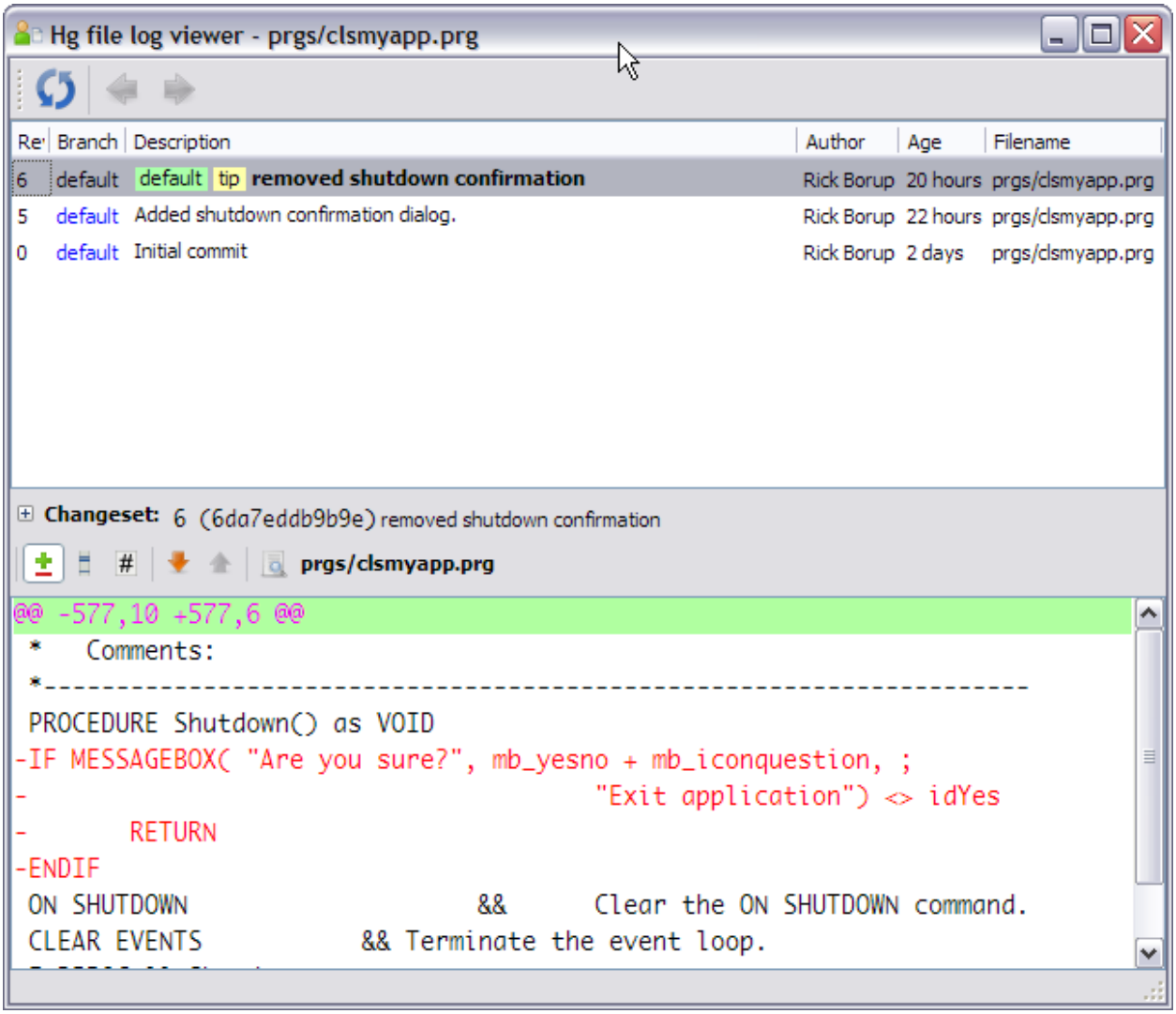


Figure 29: The file history dialog shows which revisions modified a file, and what changes were made.

Common problems and puzzlements

Problems tend to fall into three categories: your code gets hosed, your repository gets hosed, or you get confused – and sometimes all three. The point is, it's not unusual to get to a point where you want to throw up your hands and chuck the whole thing.

Don't despair: there's almost always a way out. This section is a collection, in no particular order, of some common problems and puzzlements I've encountered while working with (and learning about) Mercurial and TortoiseHg, along with explanations and suggested solutions as best I'm able to offer them.

Not a head revision

The "Not a head revision" condition is not really an error, but it looks like one due to the big red message with an exclamation point that appears in the revision history (see Figure 30).

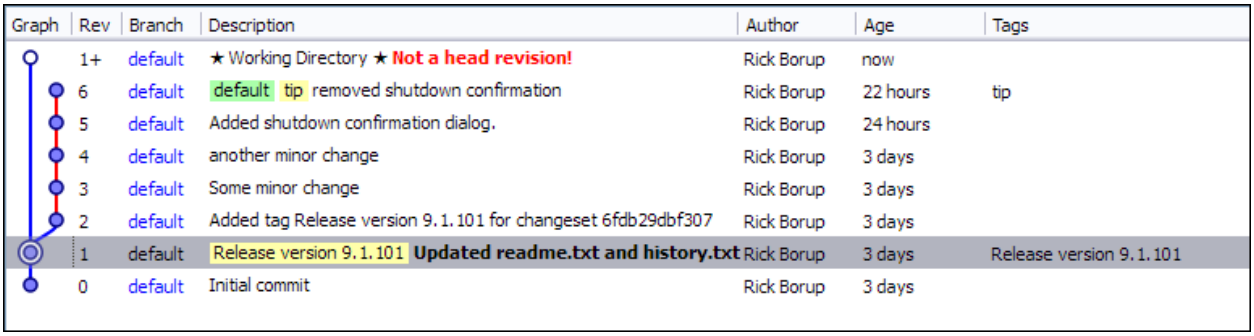


Figure 30: The "Not a head revision" message looks like an error, but it's really just informational.

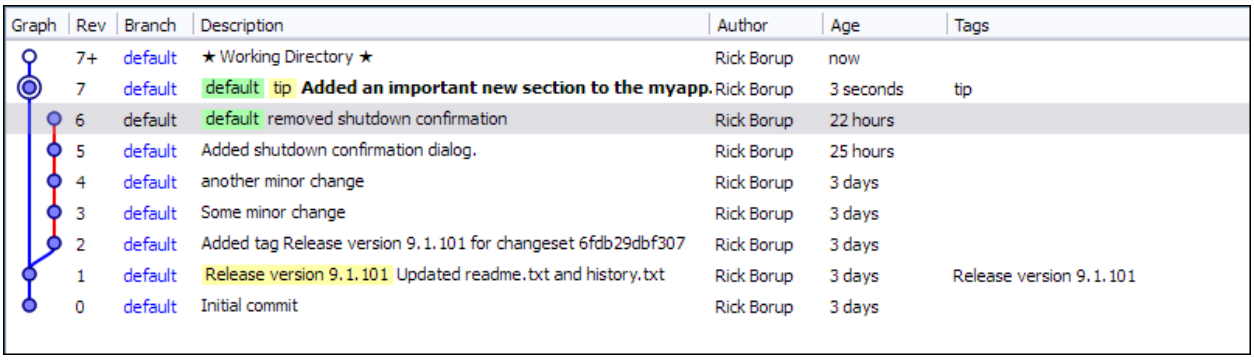
This message is simply telling you that the parent of the working directory is not a head revision – in other words, your working directory is probably not current. In the example in Figure 30, the condition was caused by updating the working directory back to revision 1. You can see this diagrammed in the revision history graph: starting from the Working Directory at the top, follow the left-most side down to the next lower node, indicating revision 1 is the current parent of the working directory. If you prefer reading it in the other direction, start from revision 1 and notice there is a fork in the graph. Following up the left-most side, you can see the working directory is the direct descendant of revision 1, while the line to the right shows that revisions 2 through 6 are also descended from revision 1.

What you do about this depends on what you're trying to accomplish. If you want to make your working directory current, just update it to the tip revision, which is revision 6 in this example. This might be the case for example if you updated to revision 1 only because you wanted to temporarily see what the files in revision 1 looked like but didn't intend to start a new line of development from that point. On the other hand, if you do want to start a new line of development from revision 1, go ahead and make your changes to the files in the working directory and do a commit. What happens at that point is the subject of the next section.

Two heads in the repository

Remember that a head revision is a revision with no descendants – i.e., one with no child revisions. There can be more than one head revision in a repository. Sometimes this

happens by accident, for example if you commit changes from a working directory whose parent is not a head revision. Continuing with the example above, a file was modified and the change was committed to the repository. Mercurial doesn't consider this an error, it simply creates a new head revision as shown in Figure 31. Note that there are now two head revisions, 6 and 7, both of which have revision 1 as their common ancestor.



Graph	Rev	Branch	Description	Author	Age	Tags
	7+	default	★ Working Directory ★	Rick Borup	now	
	7	default	default tip **Added an important new section to the myapp.**	Rick Borup	3 seconds	tip
	6	default	default removed shutdown confirmation	Rick Borup	22 hours	
	5	default	Added shutdown confirmation dialog.	Rick Borup	25 hours	
	4	default	another minor change	Rick Borup	3 days	
	3	default	Some minor change	Rick Borup	3 days	
	2	default	Added tag Release version 9.1.101 for changeset 6fdb29dbf307	Rick Borup	3 days	
	1	default	Release version 9.1.101 Updated readme.txt and history.txt	Rick Borup	3 days	Release version 9.1.101
	0	default	Initial commit	Rick Borup	3 days	

Figure 31: A repository can have more than one head revision. In the example, both 6 and 7 are head revisions.

It's perfectly normal to have more than one head revision in the repository when working with branches, because each branch will have at least one head revision of its own. In this example, however, you can see that all the revisions – including both heads – are in the default branch. Sometimes this is intentional and sometimes it is not. Because it's not really an error no solution is required, but the method for rejoining the two heads into one is the same in all situations: merge the changes from one line of development into the other.

A merge operation takes the changes from the "from" revision and merges them into the "to" revision. When choosing how to merge, the "to" side should be the one in the branch where you want development to continue after the merge.

In this example it makes sense to merge the older revision(s) into the tip, in other words to merge the changes in revision 6 (and it's ancestors, 2 through 5) into the tip at revision 7. To do this using TortoiseHg, select revision 6, right-click, and choose *Merge with local* from the context menu. This brings up the dialog in Figure 32, where you can confirm the "from" and "to" revisions are what you want.

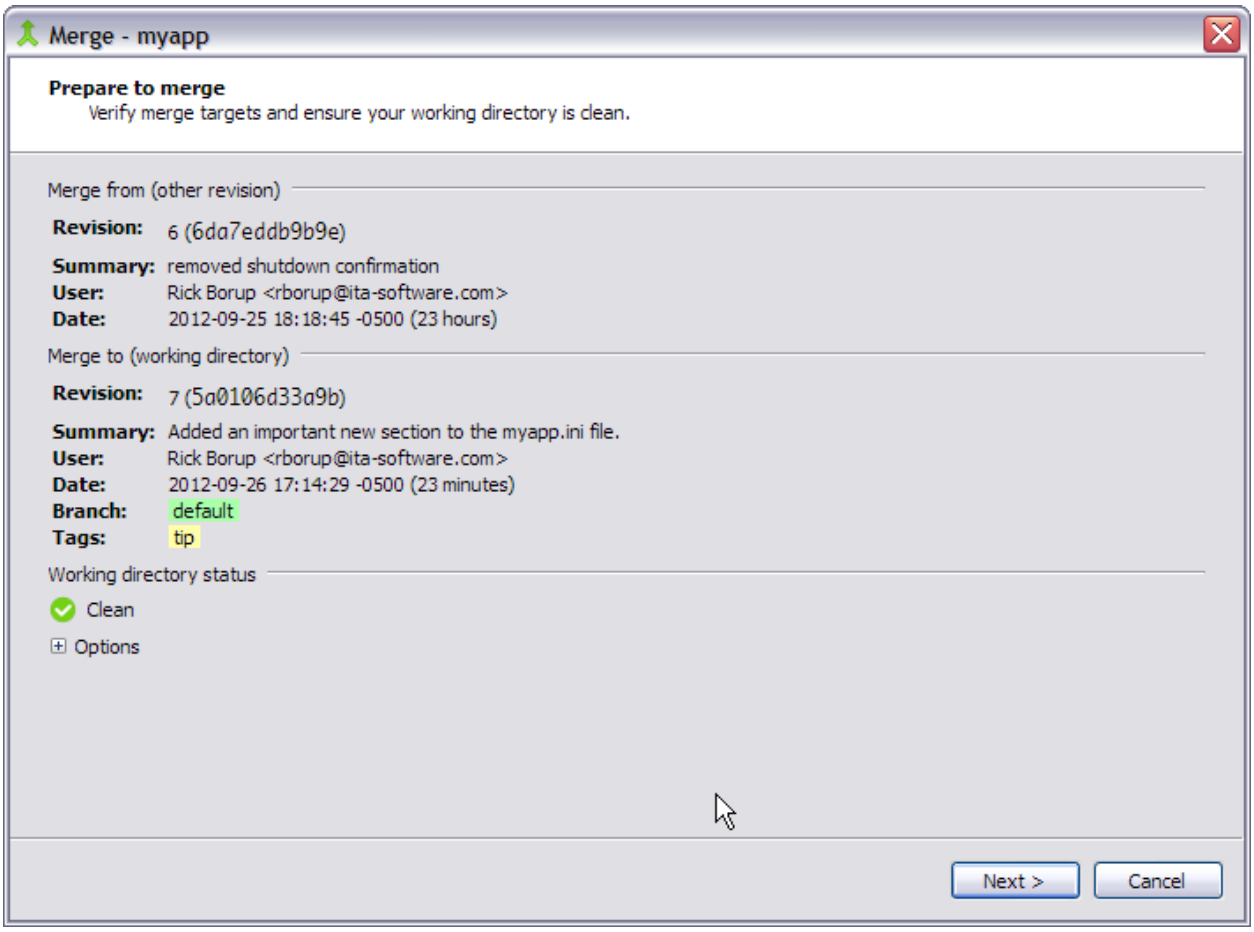


Figure 32: The *Prepare to merge* dialog gives you an opportunity to confirm the "from" and "to" revisions are the ones you want.

After confirming the merge is set up the way you want it, click the Next button to initiate the actual merging operation. The Merge dialog displays its progress as it goes along, as shown in Figure 33. In this example, the merge was successful.

Keep in mind that if there are conflicting changes to the same file, a merge conflict can result. Merge conflicts need to be resolved, sometimes manually, using the tools provided. In this example there were no merge conflicts.

Also keep in mind that a merge updates the files in the working directory, but does not automatically commit those changes back to the local repository. Mercurial reminds you that a commit is still required with the "…don't forget to commit" message, second from the bottom in Figure 33.

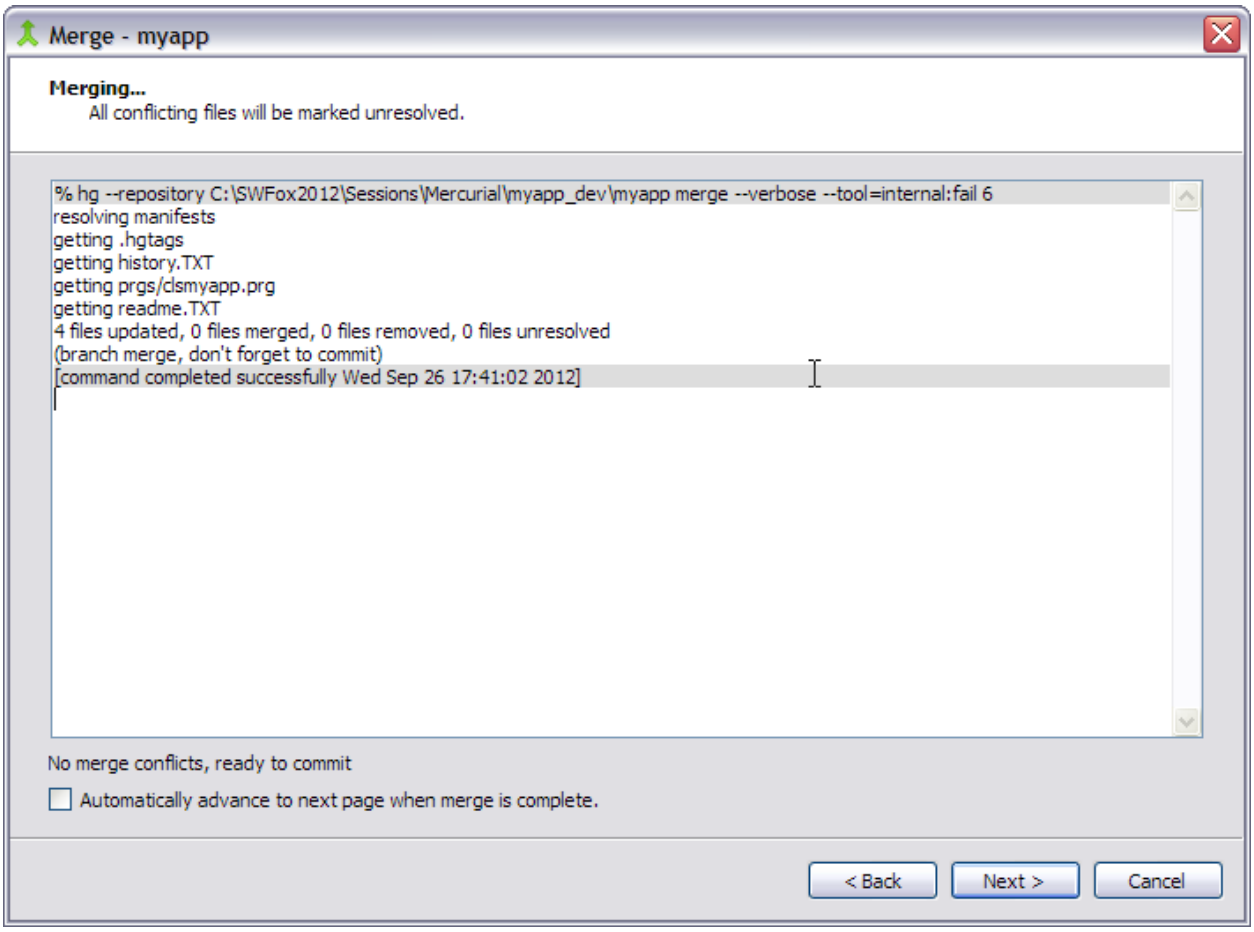


Figure 33: The Merge dialog displays its progress as it goes along and indicates whether or not the operation was successful.

Click the Next button to proceed. The Merge wizard prompts you to commit the modified files to the local repository and provides a place to enter the commit message, which defaults to “Merge”. Figure 34 shows this dialog superimposed on the main window, in which you can see the state of the revision graph before doing the commit. Notice that the graph shows revision 6 merged into revision 7, but 7 is still the tip and the working directory still has modified files that haven’t yet been committed.

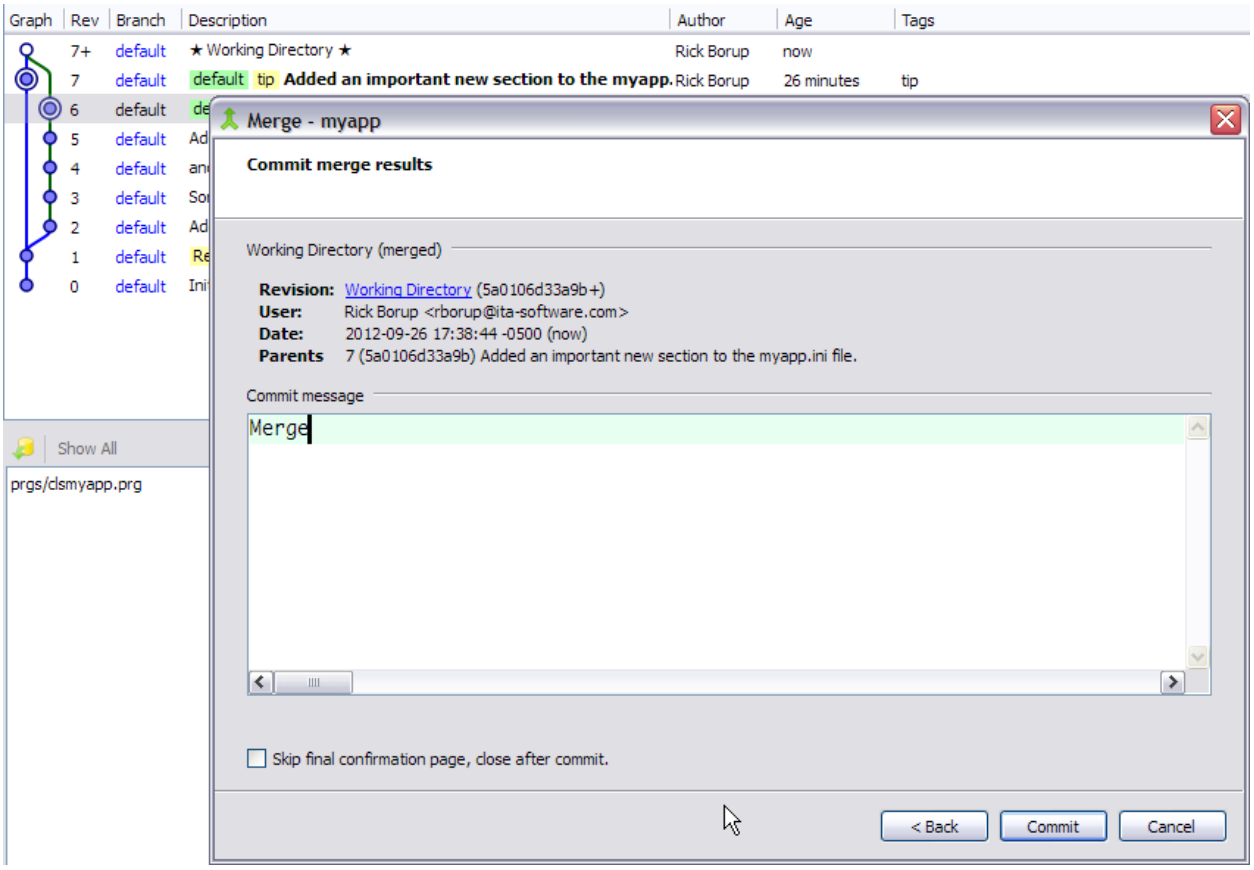


Figure 34: The last step in the Merge process is to commit the modified files from the working directory to the local repository.

As expected, committing the changes creates a new revision. By default, the Merge wizard displays a "Finished" dialog, although this is optional and can be suppressed by marking the check box in the previous step.

The revision history now shows revision 8 as the tip revision, and the graph shows where the merge took place.

Graph	Rev	Branch	Description	Author	Age	Tags
	8+	default	★ Working Directory ★	Rick Borup	now	
	8	default	default tip **After merging revision 6 into revision 7**	Rick Borup	1 second	tip
	7	default	Added an important new section to the myapp.ini file.	Rick Borup	42 minutes	
	6	default	removed shutdown confirmation	Rick Borup	23 hours	
	5	default	Added shutdown confirmation dialog.	Rick Borup	25 hours	
	4	default	another minor change	Rick Borup	3 days	
	3	default	Some minor change	Rick Borup	3 days	
	2	default	Added tag Release version 9.1.101 for changeset 6fdb29dbf307	Rick Borup	3 days	
	1	default	Release version 9.1.101 Updated readme.txt and history.txt	Rick Borup	3 days	Release version 9.1.101
	0	default	Initial commit	Rick Borup	3 days	

Figure 35: The revision history and graph reflect the merge.

The update / merge / commit cycle is a common activity in Mercurial, so it's a good idea to become comfortable with the concepts and familiar with the mechanics.

Forgetting to update after pulling

When you pull changes from a remote repository into your local repository, Mercurial does not automatically update your working directory unless you've explicitly configured it to do so.[3] If you don't update your working directory after pulling changes, and you then proceed to make changes and commit them, you can end up with two heads in the local repository.

If this happens, follow the procedure in the previous section to merge the changes and bring things back into one line of development.

Push creates new remote head

The "Push creates new remote head" error message occurs when you try to push changes to a remote repository from a local repository that is not in sync with the remote one. There are many ways this can happen. One is if you forget to pull, update, and merge from the remote repository before doing a push. Another is if you restore the local repository from a backup copy taken at an earlier state in the revision history.

Consider this scenario: You're working with a Visual FoxPro project. At regular intervals you commit your changes to the local repository and push them to a remote repository. Being a responsible developer, you also make periodic backup copies of your project files in case of emergency. Since the local repository is located in a subfolder of the project's root folder, the local repository is included in the backup. So far, so good.

Then something happens—a hard drive failure, sector errors, accidental erasure, or whatever—and you decide to restore the project's files to your local hard drive from the backup. As it turns out, the most recent backup is one revision older than the one you lost, but you know what you changed and you know how to reproduce those changes. You restore the files from the backup, re-enter the changes, commit to your local repository, and push to the remote repository.

This is when you get the error message, as shown in Figure 36.

[3] Select the Update option on the sync toolbar in TortoiseHg Workbench, or use the –u option with the hg pull command.

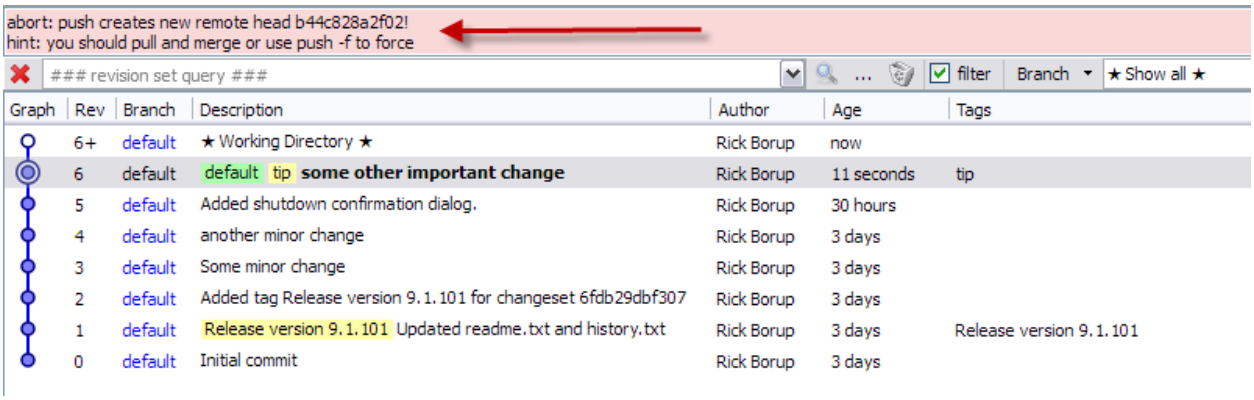


Figure 36: The "Push creates new remote head" error occurs when the local repository and the remote repository are out of sync.

There are a couple of way to avoid this error. One way is to start over with an empty folder on you local machine, clone the remote repository into a new local repository in that folder, and then update the working directory from there, thus re-creating the source files. This may or may be an optimal solution, depending on whether the repository contains everything needed to build the app or whether there are important files not under version control that aren't available elsewhere.

Another way to avoid this error is to pull changes from the remote repository immediately after restoring the local from backup, and then update and merge those changes into your working directory. This is typically the preferred approach, as it ensures the files in your working directory are up to date and in sync with the remote repository.

If you do find yourself facing this error, the solution is to pull changes from the remote repository, merge them with the local repository, resolve any merge conflicts, commit, and then push back to the remote. Use the *Incoming* icon on the sync toolbar was used to detect any incoming revisions. In this example, one incoming changeset was detected and is accepted (see Figure 37).

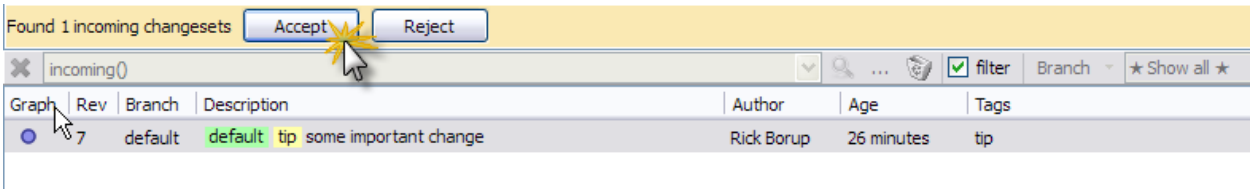


Figure 37: When "incoming" changesets are detected, click the Accept button to pull them into the local repository.

Clicking the *Accept* button triggers the pull operation, which updates the local repository but does not affect the working directory. The revision history at this point shows that the parent of the working directory is still revision 6, while local repository now contains a new tip at revision 7, as illustrated in Figure 38.

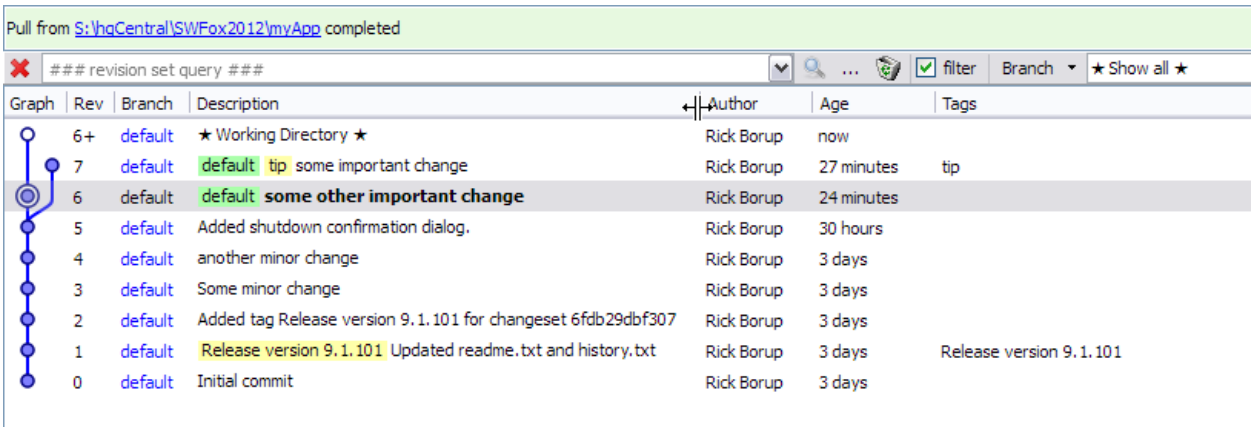


Figure 38: After accepting the incoming changesets, the revision history shows two heads in the local repository.

At this point you're back in familiar territory, as this situation is the same "two heads need to be merged into one" that we've seen before. In this case we want to merge the tip (revision 7) into the mainline (revision 6). Although not shown here, the merge is this example caused a merge conflict which needed to be resolved. After resolving it, the commit was successfully completed and the revision history now shows revision 8 as the new tip (see Figure 39).

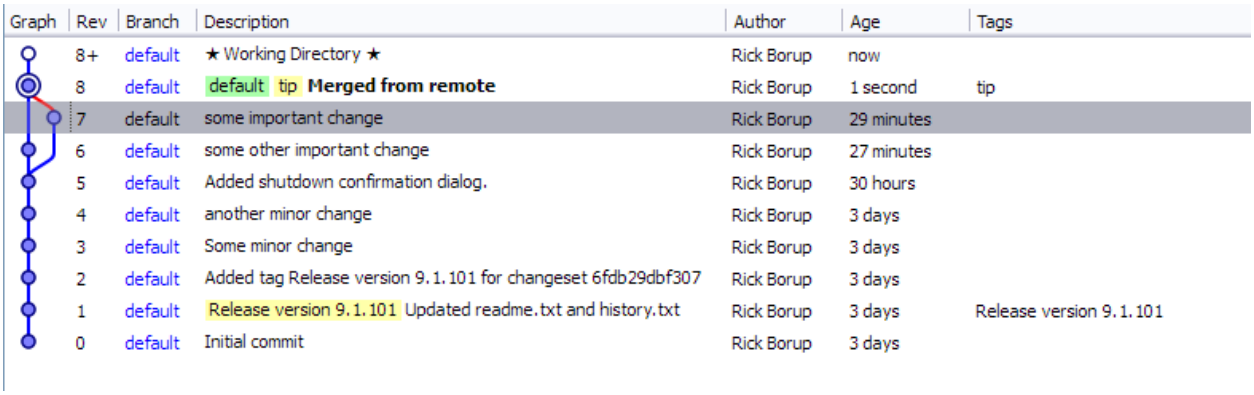


Figure 39: After merging and committing, the revision history shows everything is back in order.

You're now ready to push the changes back up to the remote repository. The "outgoing" tool shows two changesets ready to be pushed, namely revisions 6 and 8.

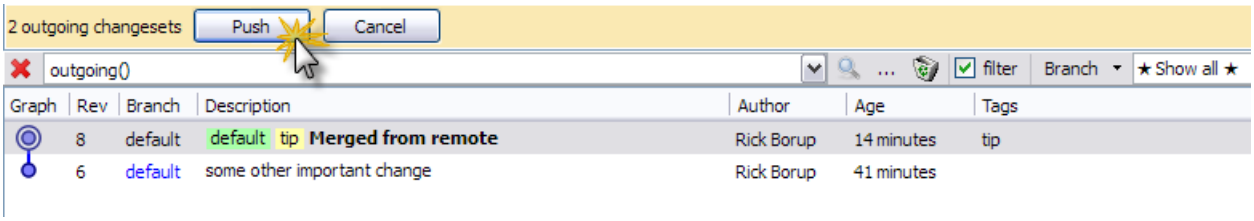


Figure 40: Click the Push button to push the outgoing changesets to the remote repository.

Click the Push button to initiate the push operation. This time, no errors are detected and the push is successful.

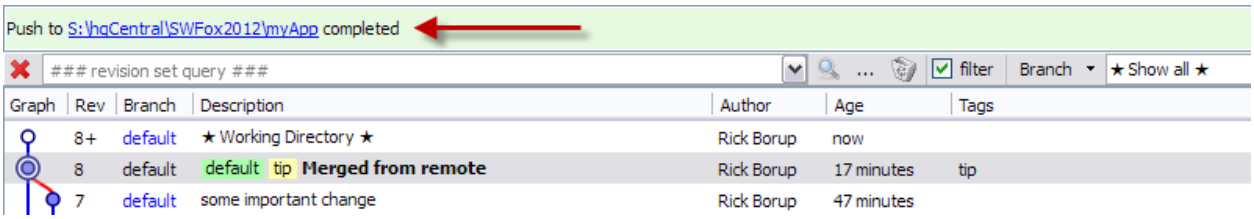


Figure 41: The push can now be successfully completed.

Miscellaneous things to know or to watch out for

— Adding an exclusion to the .hgignore file does not stop Mercurial from continuing to track matching files that are already in the repository. For example, suppose you initially add the binary file *myForm.scx* to your repository. Later, you decide you no longer want to have the binaries under version control, so you add **.scx* to the list of exclusions in the .hgignore file. New .scx files will not be added, but myForm.scx will continue to be tracked because it's already in the repository. To stop tracking myForm.scx, you need to tell Mercurial to "forget" it. There's an icon on the Workbench context menu to forget a file.

— The TortoiseHg Workbench usually refreshes its display automatically after an operation is performed, but sometimes it doesn't. If the display doesn't seem to reflect what you believe is the correct representation of the state of the repository or the working directory, press F5, or choose View | Refresh from the main menu, or click the *Refresh* button on the appropriate toolbar when one is present.

— Adding a remote repository to the list in the TortoiseHg Workbench sync panel is not as intuitive as it could be. When selecting a local folder, there does not appear to be any way to bring up an Explorer dialog to navigate to the desired folder. You can type the entire drive and path in by hand, but that's tedious and prone to error. An easier way is to drag the desired folder icon from Windows Explorer and drop it onto the field in the sync panel.

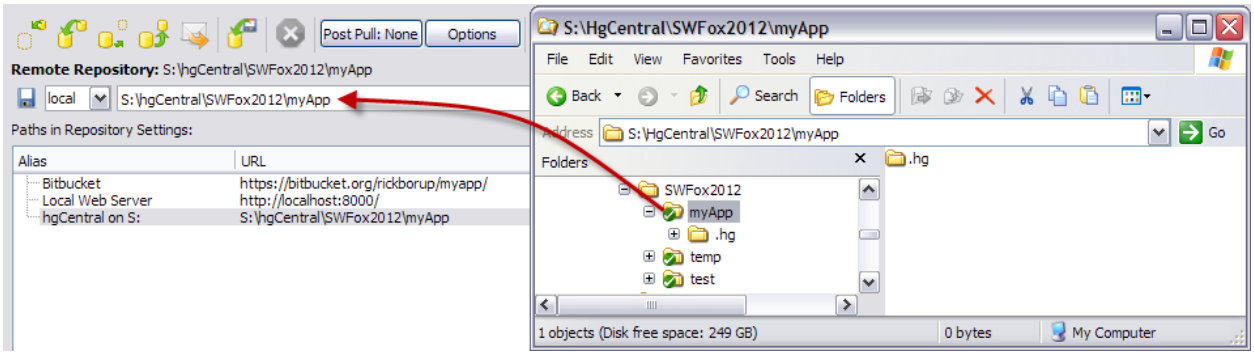


Figure 42: To add a remote repository to the list, you can drag a folder icon from Windows Explorer and drop it in the remote repository field on the Workbench sync panel.

— The context menu for remote repositories in the TortoiseHg Workbench sync panel includes an option to remove the selected repository from the list. This doesn't seem to work. Even after confirming you want to remove it, the selected repository remains visible in the list and is still present in the repository's hgrc configuration file, which is where the list gets its data. The only way I've found to remove it from the list is to edit the hgrc file and remove the unwanted line from the *paths* section, as shown in Figure 43.

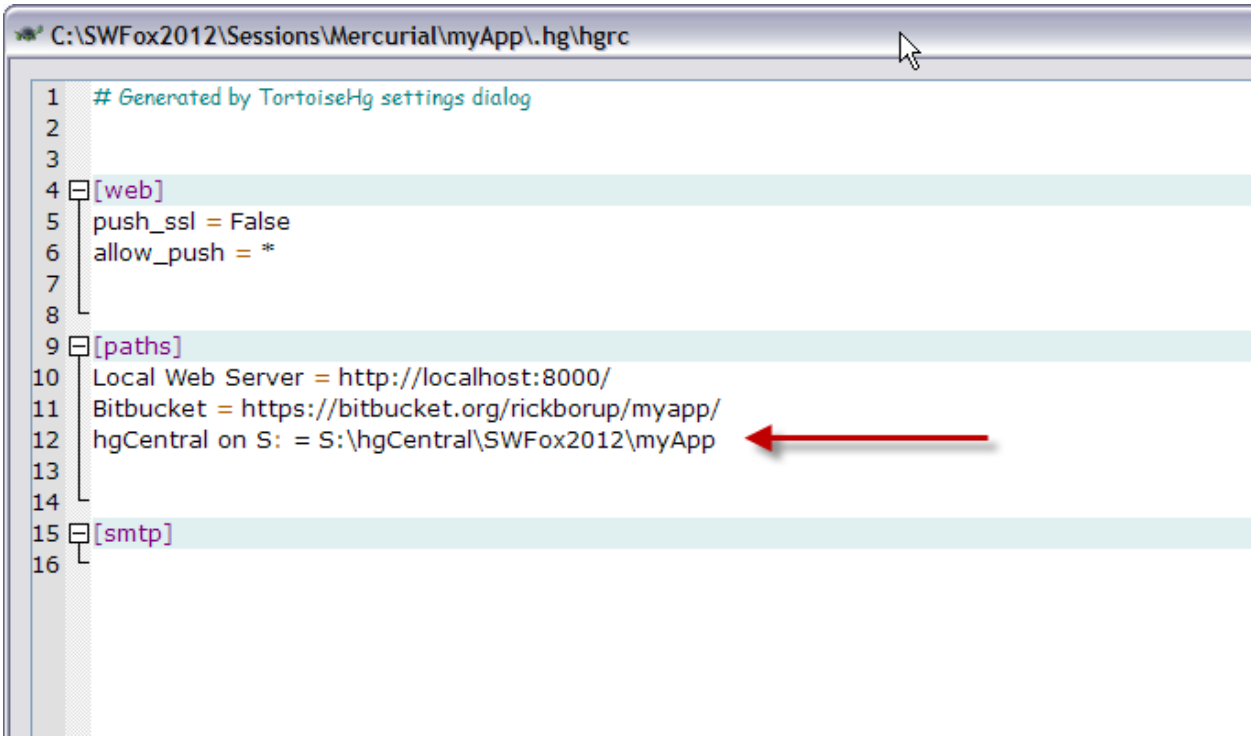


Figure 43: To remove a remote repository from the list displayed in the TortoiseHg Workbench sync panel, edit the repository's hgrc file and remove the line from the [paths] section.

Summary

Mercurial is a powerful distributed version control tool. When combined with the graphical interface provided by TortoiseHg, the two make an excellent DVCS solution for developers working on a Windows platform. Both Mercurial and TortoiseHg are free, and there are plenty of resources for getting started and for further exploration. A few of them are listed below. Good luck, and stay committed!

Resources

A Beginner's Guides to Mercurial
http://mercurial.selenic.com/wiki/BeginnersGuides

Mercurial: The Definitive Guide
http://mercurial.selenic.com/wiki/MercurialBook
and also online at http://hgbook.red-bean.com/read/

TortoiseHg documentation:
http://tortoisehg.bitbucket.org/manual/2.5 (URL changes with version number)

Email listserve:
https://lists.sourceforge.net/lists/listinfo/tortoisehg-issues