This paper was originally presented at the Southwest Fox conference in Gilbert, Arizona in October, 2011. http://www.swfox.net

# VFP Version Control with Mercurial

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

Mercurial is a distributed version control system (DVCS) well suited for use with Visual FoxPro application development. While distributed version control systems are based on a decentralized model designed to facilitate team software development, they are also useful for the independent developer who's working solo. Together with the TotoiseHg shell program for Windows, Mercurial offers VFP developers a powerful tool for managing their version control requirements. Come to this session and learn how to integrate Mercurial into your daily VFP development workflow.

 

# Table of Contents

Table of Contents ..... 2
Distributed Version Control System Concepts ..... 4
Installing Mercurial and TortoiseHg on Windows ..... 6
Getting to know Mercurial ..... 7
Key concepts and terminology ..... 7
Working from the command prompt ..... 9
hg – the Mercurial command ..... 9
Creating a local repository ..... 10
Doing the initial commit ..... 11
Making the first change ..... 13
Exploring differences ..... 14
Updating and backdating ..... 14
Introducing TortoiseHg ..... 15
Branching and merging within a repository ..... 18
Working with remote repositories ..... 23
Handling merge conflicts ..... 26
Special Considerations for VFP ..... 33
What to include in the repository ..... 33
What to ignore ..... 33
What to do about binary files ..... 34
The effect of recompiling all files ..... 36
Case sensitivity ..... 36
Integration with the VFP project manager ..... 36
Integrating Mercurial into your daily development workflow ..... 37
Step 1 – Create a local repository ..... 37
Step 2 – Enter your configuration settings ..... 39
Step 3 – Set up your .hgignore file ..... 40
Step 4 – Add files and do the initial commit ..... 40
Step 5 – Rinse and repeat ..... 43
Field notes from working with Mercurial ..... 44

 

Backups ..... 44
Portable repositories ..... 44
Use descriptive messages with every commit ..... 44
Tips, tricks, and advanced techniques ..... 45
Getting help ..... 45
Fixing Mistakes ..... 46
Revert ..... 46
Rollback ..... 47
Backout ..... 48
Previewing actions ..... 48
More about cloning ..... 50
Are you being served? ..... 51
Hosting solutions ..... 55
Tips ..... 55
Basic commands ..... 55
Working with a remote repository ..... 57
Mercurial extensions ..... 57
FAQs ..... 57
Resources ..... 58
Downloads for Mercurial and TortoiseHg ..... 58
References and support ..... 58
Summary ..... 59

# Distributed Version Control System Concepts

All version control systems (VCS) have in common the concept of a repository. The repository is the location where the VCS stores the information it uses to track the changes to a project's files over time.

You can think of a repository as a kind of filing cabinet where the records of changes are kept. While different version control systems may use different physical forms for their repository—be it file-based, a database, or whatever—conceptually it's always the same thing.

There are two distinct types of version control systems, centralized and distributed. As its name implies, a centralized version control system (CVCS) has a single central repository that is shared by all developers working on the project. In order to work on a file, a developer must check it out from the central repository. The checkout process creates (or updates) a copy of the file on the developer's local machine, and marks the file as locked in the central repository. Other developers cannot check out the same file until the first developer checks it back in. While this type of version control can be useful, the need to lock files often creates bottlenecks for team development.

Figure 1: A centralized version control system features a single central repository shared by all members of the development team.

In contrast, a distributed version control system (DVCS) does not require a central repository. Instead, each developer has their own local repository that resides on their own local machine. Along with the local repository is the concept of the working directory, sometimes also referred to as the working copy. The working directory is simply the name given to the set of folders and files with which the developer directly interacts during the development process. For VFP developers, the working directory is the project folder.

The concept of a local repository give rise to one of the best features of a DVCS, even for the solo developer: it's fast! Because commits and other interactions with the local repository take place on the local machine at hard drive speeds, there is no latency or dependency on a connection to a remote repository.

In a DVCS, collaboration among members of a team is facilitated by the ability for developers to interact with one another's repositories. For example, with appropriate authorization one developer can create a clone (an exact copy) of another team member's repository. The two developers can then begin collaborating on the project from the same starting point. The developer who creates the clone receives the current version of all tracked files in the working directory along with the project's complete version history in the local repository.

When a developer is ready to store a set of changes, they commit those changes to their local repository. Because this occurs on their local machine, it has no impact on other members of the team. When they're ready, developers can share their changes with other developers by allowing them to create a clone, to pull changes into their own local repository, or to push their changes to a central repository or even directly to another developer's local repository.

Figure 2: In a distributed version control system (DVCS), each developer has their own local repository and can share changes directly with other developers and/or with a central repository.

Mercurial and Git are two popular DVCS. Git is popular with Ruby on Rails developers, among many others. Mercurial is popular with Python developers, partly because it's written in Python. Mercurial and Git are conceptually and functionally very similar, although their command syntax differs somewhat. Both are quite fast, and both offer integration with the Windows® shell via the Tortoise shell extension (TortoiseHg or

TortoiseGit, both of which derive from TortoiseSVN for Subversion). However, Mercurial appears to be growing as the preferred choice by developers working on Windows®.

Mercurial commands are straightforward and mostly intuitive. Mercurial can use http and https to clone, push to, and pull from a remote repository so no special or proprietary protocols are required.

## Installing Mercurial and TortoiseHg on Windows

The Web home of Mercurial is http://mercurial.selenic.com. This site includes a page of links for several variations of Mercurial installer for Windows®, some of which include the TortoiseHg Windows shell extension.

If you're installing Mercurial on Windows, however, you can go to the TortoiseHg site and download the most recent installer from there. Go to http://tortoisehg.bitbucket.org and download the current release. The installer includes both the Mercurial distributed version control software and the TortoiseHg Windows shell extension. At the time of this writing the current versions are TortoiseHg 2.0.4 and Mercurial 1.8.3, but it's likely there will be newer point releases by the time you read this.[1]

Note that there are different installers for 32-bit Windows and for 64-bit Windows, so be sure to download the appropriate one for your system. There is also a link to the release notes page, which you may want to browse through for information on what's new in the current release as well as what may be planned for future releases.

TortoiseHg and Mercurial are installed from a Windows Installer (.msi) file. After downloading, double-click on the file or right-click and choose "install" from the context menu. TortoiseHg and Mercurial are both installed to the C:\Program Files\TortoiseHg folder on your local machine.

Although there is an extensive set of configuration options available, very little is actually required to begin working with Mercurial and TortoiseHg. Global configuration settings are stored in a mercurial.ini file in your user profile (e.g., C:\Users\Rick), while project-specific configuration settings are stored in an hgrc file in the local repository. The basic configuration settings are discussed later on as they come up in examples.

[1] Updated versions of TortoiseHg and Mercurial are released approximately once a month on the TortoiseHg website. As I finish this paper in September 2011, the current versions are TortoiseHg 2.1.3 and Mercurial 1.9.2.

# Getting to know Mercurial

## *Key concepts and terminology*

Before you get started working with Mercurial, it's important to have a firm grasp on some of the basic concepts and terminology. Many of the concepts relate directly to the actual commands you use to interact with Mercurial.

The working directory is where you work on your project. For any given project, the working directory is simply the generic name that is used to refer to the project's root folder, its subfolders, and the files they contain. In Visual FoxPro, the working directory is your VFP project folder.

Each project that is placed under Mercurial version control has a local repository. In Mercurial, a project's local repository is located in a subfolder named .hg located directly under the root of the working directory. A project's local repository is where the history of changes to the project's files is stored.

Mercurial tracks files, not folders. It provides a mechanism for specifying which files are to be tracked and which are to be ignored.

The local repository stores information about the changes between successive revisions of the files being tracked. This information is called a changeset. Except for the initial commit, the local repository does not store complete copies of the files it is tracking; instead, it stores only the changesets, which are also sometimes referred to as deltas.

Changesets are identified by a revision number and a changeset ID. The revision number is an integer value starting at zero for the initial commit and incremented by 1 for each successive commit. The changeset ID is a SHA-1 hash displayed as a 12-character hexadecimal value. Revision numbers are unique only within a local repository, whereas changeset IDs are globally unique. Together they are usually written as the revision number, followed by a colon, followed by the changeset ID as illustrated in Figure 3.

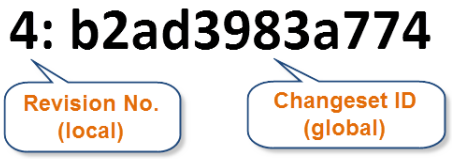


Figure 3: Changesets are identified by a revision number, which is local to the specific repository, and by a changeset ID, which is global.

When you've made some changes to a file or files in your working directory and are ready to have Mercurial store a record of those changes in the local repository, you do what's called a commit. A commit creates a new changeset in the local repository.

Sometimes you need to change things in the opposite direction, i.e., to apply changes stored in the local repository to the files in working directory. In Mercurial this is called an update. Unless the working directory and the local repository are already in sync, an update changes the contents of one or more files in the working directory.

The word update can be a source of confusion when first learning Mercurial. In common usage the word means "to make more current", but in Mercurial an update can work either forward and backward in time. This means that while you can update a working directory to a more recent changeset from your local repository – for example, a changeset you pulled from another developer – you can also update a working directory back to an older changeset. It might help to think of the latter as a backdate instead of an update, but in Mercurial you use the update command in both situations. The important thing is to remember that an update changes the working directory by synchronizing its contents with a specified changeset from the local repository.

A remote repository is any repository other than the local one. In Mercurial, a remote repository is what you clone, push to, or pull from. A remote repository can be anywhere – on the local machine, on a network, or an online resource accessed via the Internet.

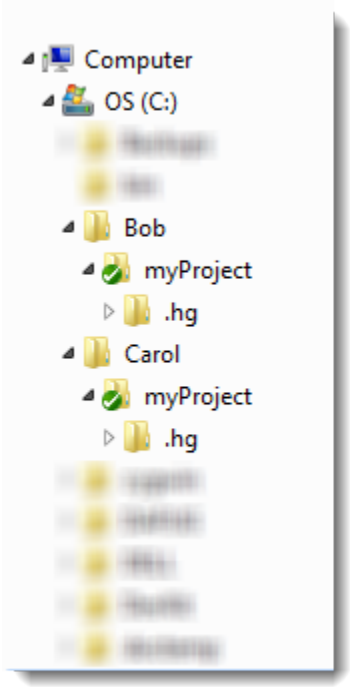


Figure 4: A project's local repository is stored in the .hg folder underneath the root of the working directory. In this example, both Bob and Carol are working on a project called "myProject". Each of them has their own working directory and underneath that, their own local repository. Because I need to run this presentation from a single machine, both folders are on my local hard drive, but you can think of them as being on two different machines. Either way, from Bob's point of view Carol's is a remote repository and vice-versa.

Changesets have a parent-child relationship within a repository. The changeset from which another changeset was derived is called the parent, and the derived changeset is called the child. A changeset that does not have any children is called a head revision. The most recent head revision is called the tip.

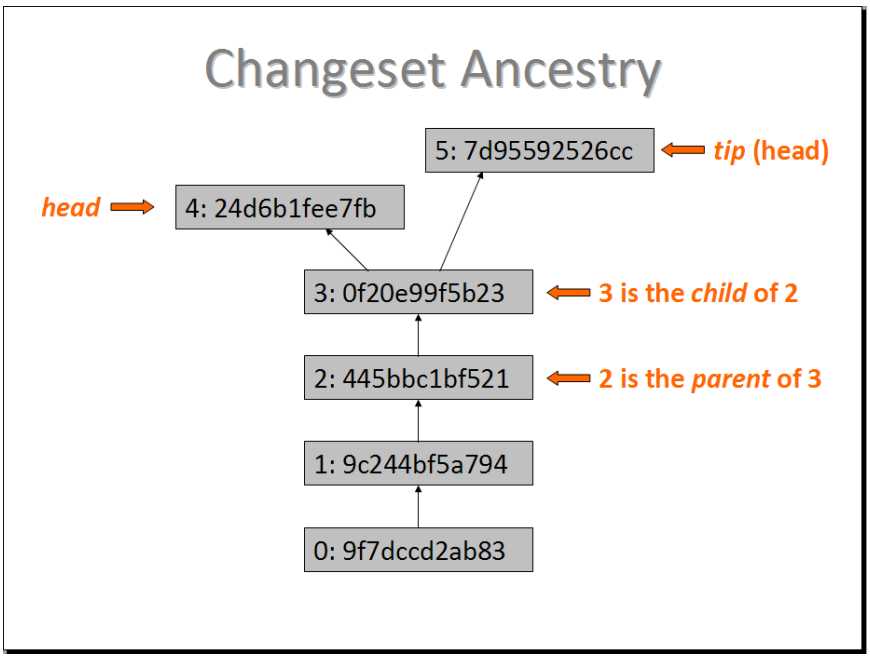


Figure 5: Within a repository, changesets have an ancestry based on a parent-child relationship.

## *Working from the command prompt*

Mercurial is a command-driven tool. Therefore the most direct way to use it is from the command prompt. While you will most likely end up preferring the TortoiseHg shell, which enables you to interact with Mercurial from a GUI environment, the command line is the best way to learn Mercurial. Once you learn the basic Mercurial commands and understand what they do, it will be easier for you to understand and use the TortoiseHg interface.

### hg – the Mercurial command

When working from the command prompt, all Mercurial commands are invoked by typing hg followed by the name of the command. This runs the program hg.exe, which was installed in C:\Program Files\TortoiseHg and added to your path. For example, type hg version  to see which version of Mercurial is installed.

```
C:\>hg version
Mercurial Distributed SCM (version 1.9.2)
(see http://mercurial.selenic.com for more information)

Copyright (C) 2005-2011 Matt Mackall and others
```

```
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

The examples that follow are drawn from the experiences of Bob and Carol, two developers who work for a small software development company called Megasoft located in Bluemond, Washington. Bob and Carol are collaborating on a project called myProject, which will be written in Visual FoxPro.

The project is just getting started. The Megasoft marketing department has promised customers that this will become the absolutely best app in the industry. However, the details conveyed to the developers at this point are sketchy, other than being told it must be awesome. With this in mind, Bob writes the first line of code.

```
? “Fox rocks!”
```

This pleases the marketing department, although customers who are shown the early beta are left to wonder what the app might actually do.

## Creating a local repository

Knowing that the project will go through some changes and most likely end up being more than one line of code, Bob decides to put it under version control. He has heard of Mercurial and decides to try it. After installing and learning a little about it he bravely sets out to create the local repository for myProject.

Bob first saves his work as fox.prg in the C:\Bob\myProject folder on his machine. He then opens a command window, CD’s to the C:\Bob\myProject folder, and issues the `hg init` command to create the local repository.

```
C:\Bob\myProject>hg init

C:\Bob\myProject>
```

Hmm. There were no error messages, so maybe this succeeded. Bob does a `dir` command to see if anything happened.

```
C:\Bob\myProject>dir
 Volume in drive C has no label.
 Volume Serial Number is 8869-060D

 Directory of C:\Bob\myProject

09/17/2011  04:16 PM    <DIR>          .
09/17/2011  04:16 PM    <DIR>          ..
09/17/2011  04:14 PM    <DIR>          .hg
09/17/2011  04:16 PM                16 fox.prg
               1 File(s)             16 bytes
               3 Dir(s)  24,673,058,816 bytes free
```

Bob sees that the .hg folder has been created, as expected. Although he knows he's not supposed to ever have to actually work directly with any of the files in the .hg folder, he wonders what's in there so he does a dir on that folder.[2]

```
C:\Bob\myProject\.hg>dir
 Volume in drive C has no label.
 Volume Serial Number is 8869-060D

 Directory of C:\Bob\myProject\.hg

09/17/2011  04:14 PM    <DIR>          .
09/17/2011  04:14 PM    <DIR>          ..
09/17/2011  04:14 PM                57 00changelog.i
09/17/2011  04:14 PM                33 requires
09/17/2011  04:14 PM    <DIR>          store
               2 File(s)             90 bytes
               3 Dir(s)  24,673,058,816 bytes free
```

He sees that the .hg folder contains a couple of files and a subfolder named `store`. It looks like everything worked.

## Doing the initial commit

Bob decides this is a good time to store a record of his work in the local repository. He knows this is what the commit command is for, so he types hg commit.

```
C:\Bob\myProject>hg commit
nothing changed
```

What's this? Nothing changed? Why not??

Bob scratches his head for a minute, then remembers that Mercurial tracks only those files that it's been told to track. He hasn't told it what files to track yet, so it didn't know what to do. Bob remembers this is what the add command is for, so he uses that command to tell Mercurial to add the files in his working directory to those being tracked for this project.

```
C:\Bob\myProject>hg add
adding fox.prg
```

Looking good. Bob now tries to commit again.

```
C:\Bob\myProject>hg commit
abort: no username supplied (see "hg help config")
```

[2] One exception to this rule is the .hg\hgrc configuration file, which you may want or need to edit manually when working from the command prompt.

Starting to get frustrated, Bob takes a short break and thinks back over what he learned about Mercurial. After a bit he remembers that in order to identify who is responsible for each changeset, every commit must be associated with a specific username. He looks at the Mercurial documentation and finds that project-level usernames are stored in the .hg\hgrc configuration file. That file doesn't exist yet, so he creates it and enters a username for himself using the standard first name, last name, email address format.[3]

```
C:\Bob\myProject>copy con .hg\hgrc
[ui]
username = Bob Beta <bob@megasoft.com>
^Z
        1 file(s) copied.
```

Bob uses the copy con[sole] command to create the hgrc file because he knows that if he used Notepad he would have ended up with hgrc.txt, which Mercurial would not recognize. In your own work you will most likely be using TortoiseHg, which provides a GUI interface to the global mercurial.ini and the project-specific hgrc configuration files. I've used the manual method here for purposes of illustration, but in actual use it's unlikely you'll need to create or edit the configuration files manually.

Bob is finally ready to do the initial commit. He remembers that every commit must be accompanied by a message, which can be included with the commit command by adding the –m parameter.[4]

```
C:\Bob\myProject>hg commit -m "Initial commit"

C:\Bob\myProject>
```

There are no error messages, just a new command prompt, so it must have worked. Bob checks the log to see what happened.

```
C:\Bob\myProject>hg log
changeset:   0:7792cec862ab
tag:         tip
user:        Bob Beta <bob@megasoft.com>
date:        Sat Sep 17 16:45:55 2011 -0500
summary:     Initial commit
```

Cool. The Mercurial log shows Bob that the local repository now contains a changeset identified as revision 0. It has a unique changeset ID and is tagged as the tip revision. The

[3] The username field can contain whatever you want it to, but the first name, last name, email address format is the accepted standard.

[4] If you don't specify a comment with the –m parameter, Mercurial will open the default text editor, usually notepad.exe, and prompt you to enter a comment.

changeset is associated with the username Bob entered in the project's configuration file, it was committed on the date and time shown, and it is described as the "Initial commit".

## Making the first change

Early customer feedback shows that the product needs more features, but the marketing department is still vague on exactly what to add. Bob decides that if saying it once is good, saying it three times is three times as good, so he makes the following change to fox.prg.

```
for i = 1 to 3
  ? "Fox rocks!"
endfor
```

After testing the change to be sure it works, Bob is ready to commit the change to the repository. He first checks the status of the working directory to see what's going to be committed.

```
C:\Bob\myProject>hg st
M fox.prg
```

Sure enough, Mercurial has detected the change to fox.prg and shows it with status M, meaning "modified". Bob commits the change to the repository with an appropriate comment.

```
C:\Bob\myProject>hg commit -m "changed to 3 times"
```

He then checks the log to see what's happened.

```
C:\Bob\myProject>hg log
changeset:   1:e8f6bd8efd36
tag:         tip
user:        Bob Beta <bob@megasoft.com>
date:        Sun Sep 18 12:57:13 2011 -0500
summary:     changed to 3 times

changeset:   0:7792cec862ab
user:        Bob Beta <bob@megasoft.com>
date:        Sat Sep 17 16:45:55 2011 -0500
summary:     Initial commit
```

The log shows there are now two changesets. By default, they're listed in reverse chronological order, with the most recent on top. Bob can see that the commit he just did has been assigned revision number 1 and has its own unique changeset ID.

**If you remember nothing else from this session, remember the three basic commands: init, add, and commit.**

## Exploring differences

One thing developers commonly need to do when working with a version control system is to compare two versions of a file to see what changed. Mercurial provides the diff command for this purpose. The diff command take a –r parameter to tell Mercurial which two revisions to compare. Revisions can be specified either by their changeset ID or by their revision number.

Still new to Mercurial and not entirely trusting it yet, Bob decides to check and see if Mercurial correctly recognizes the change he made to fox.prg. From the log, Bob knows that the first commit was revision 0 and the most recent was revision 1, so he passes those numbers to the diff command to find out how Mercurial has tracked the change.

```
C:\Bob\myProject>hg diff -r0 -r1 fox.prg
diff -r 7792cec862ab -r e8f6bd8efd36 fox.prg
--- a/fox.prg   Sat Sep 17 16:45:55 2011 -0500
+++ b/fox.prg   Sun Sep 18 12:57:13 2011 -0500
@@ -1,1 +1,3 @@
-? "Fox rocks!"
+for i = 1 to 3
+  ? "Fox rocks!"
+endfor
```

Whoa. What the heck is that funky syntax all about?

Bob takes a look at the Mercurial documentation and finds out what's he's looking at is called a universal diff format. After a bit of inspection he figures out that the line marked with a minus sign was deleted and the three lines marked with a plus sign were added. He also observes that the diff output shows the changeset IDs of the two revisions, along with the date and time they were committed.

Having read ahead in the material, Bob knows that later on he'll be able to use a visual diff tool to better see the differences between two revisions, but he's glad to know about the universal diff format and how to read it.

## Updating and backdating

Bob's confidence in Mercurial is growing, but he want to see if it can truly handle updating the working directory to an earlier revision and back again. He knows that the update command is used for this purpose, and that it can take a revision number as a parameter.

To check this out, he decides to update the working directory back to his original version of fox.prg, which is stored as revision 0 in the repository.

```
C:\Bob\myProject>hg update -r0
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

He then uses the Windows type command as a quick way to view the contents of fox.prg.

```
C:\Bob\myProject>type fox.prg
? "Fox rocks!"
```

And voilà! Bob sees that Mercurial has updated fox.prg in the working directory back to the way it was in revision 0.

To complete the test, he issues the update command with no parameter to tell Mercurial to update the working directory to the most current revision.

```
C:\Bob\myProject>hg update
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

Displaying the contents of fox.prg, he sees it once again includes the change he made and committed in revision 1.

```
C:\Bob\myProject>type fox.prg
for i = 1 to 3
  ? "Fox rocks!"
Endfor
```

At this point, Bob's confidence in Mercurial is running high and he's feeling comfortable with the basic process of adding, committing, and updating. He decides it's time to investigate TortoiseHg, the Windows shell for Mercurial.

## *Introducing TortoiseHg*

The best way to learn how Mercurial works is to invoke the individual commands from the command prompt, as the previous examples have shown. While you can continue to do everything you need to do from the command prompt, developers who are accustomed to working in a GUI environment will most likely prefer to use the TortoiseHg shell.

The TortoiseHg shell is tightly integrated with Windows explorer. Most of the features and functions you'll use on a regular basis are available from a right-click context menu. The primary TortoiseHg interface, called the TortoiseHg Workbench, can be launched either from the context menu or directly from the Windows Start menu.

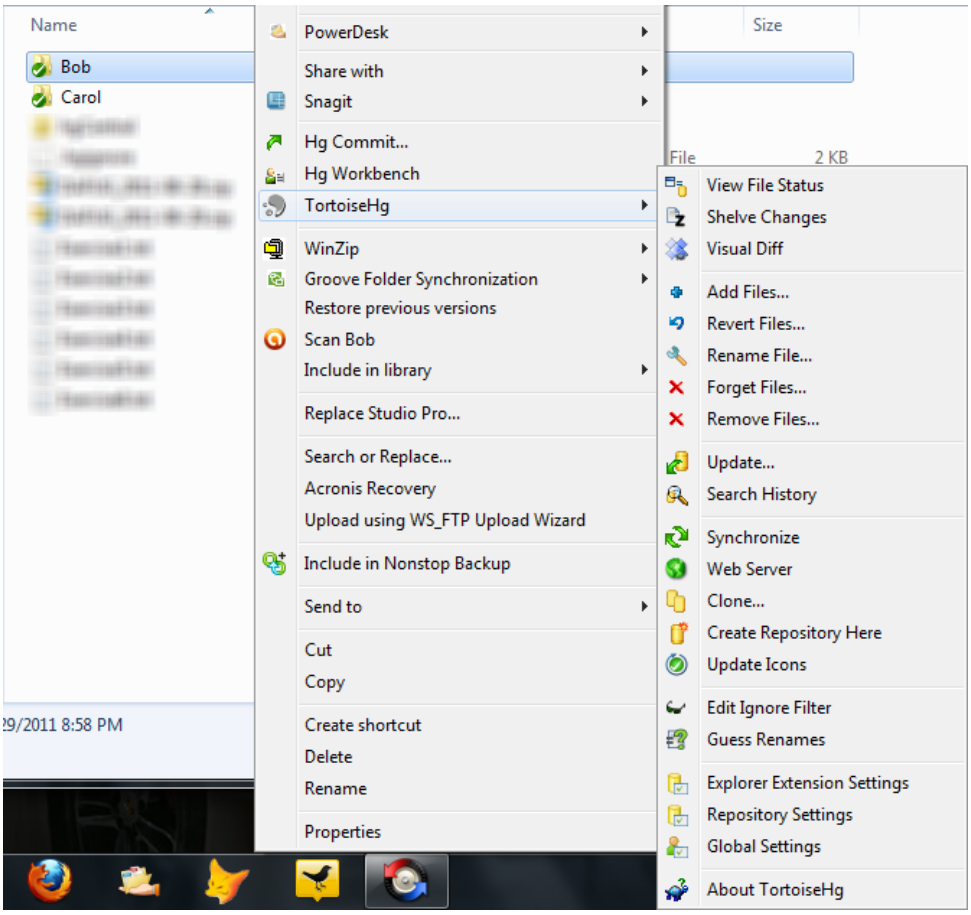


Figure 6: The TortoiseHg context menu provides access to the features and functions you'll most commonly use when working with Mercurial.

Bob decides to use TortoiseHg to take a look at the work he's already done. He launches the TortoiseHg Workbench by right-clicking on the C:\>Bob\myProject folder in Windows Explorer and choosing Hg Workbench from the context menu. He then selects revision 0 in the list and begins to explore the TortoiseHg interface as shown in Figure 7.

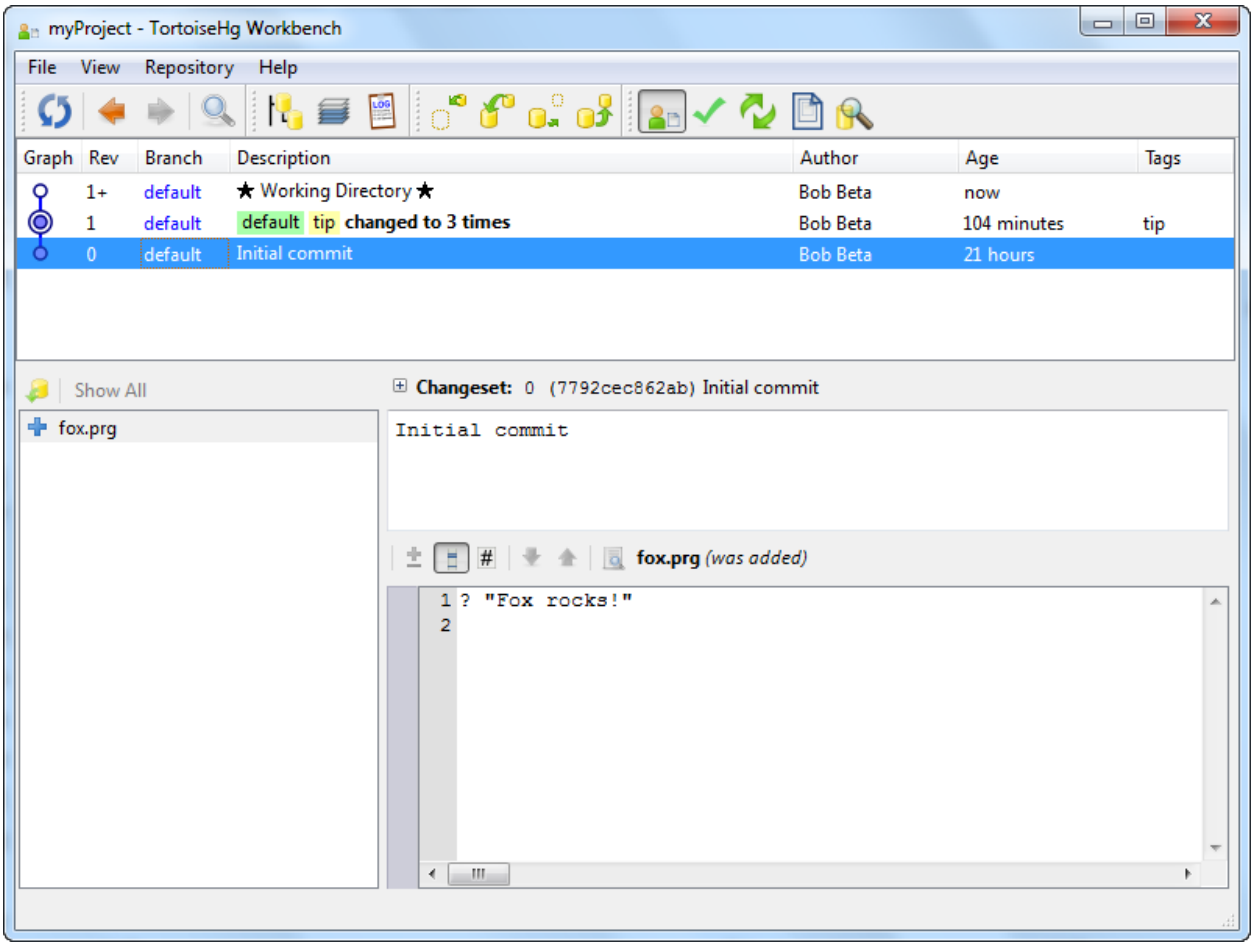


Figure 7: The TortoiseHg interface provides a visual, multi-pane interface to Mercurial.

Bob can see that the top-most pane of the TortoiseHg window shows the revision history in reverse chronological order. He realizes this is the same information he got from the log command when he was working from the command prompt, although in a somewhat different format.

The lower panes display detailed information about the selected revision. With revision 0 selected, Bob can see that the fox.prg file was added, what its content was at that time, the changeset ID that was assigned, and what commit message was associated with it.

Continuing to explore TortoiseHg, Bob selects revision 1 in the list. The lower panes now display information about that revision, as shown in Figure 8.

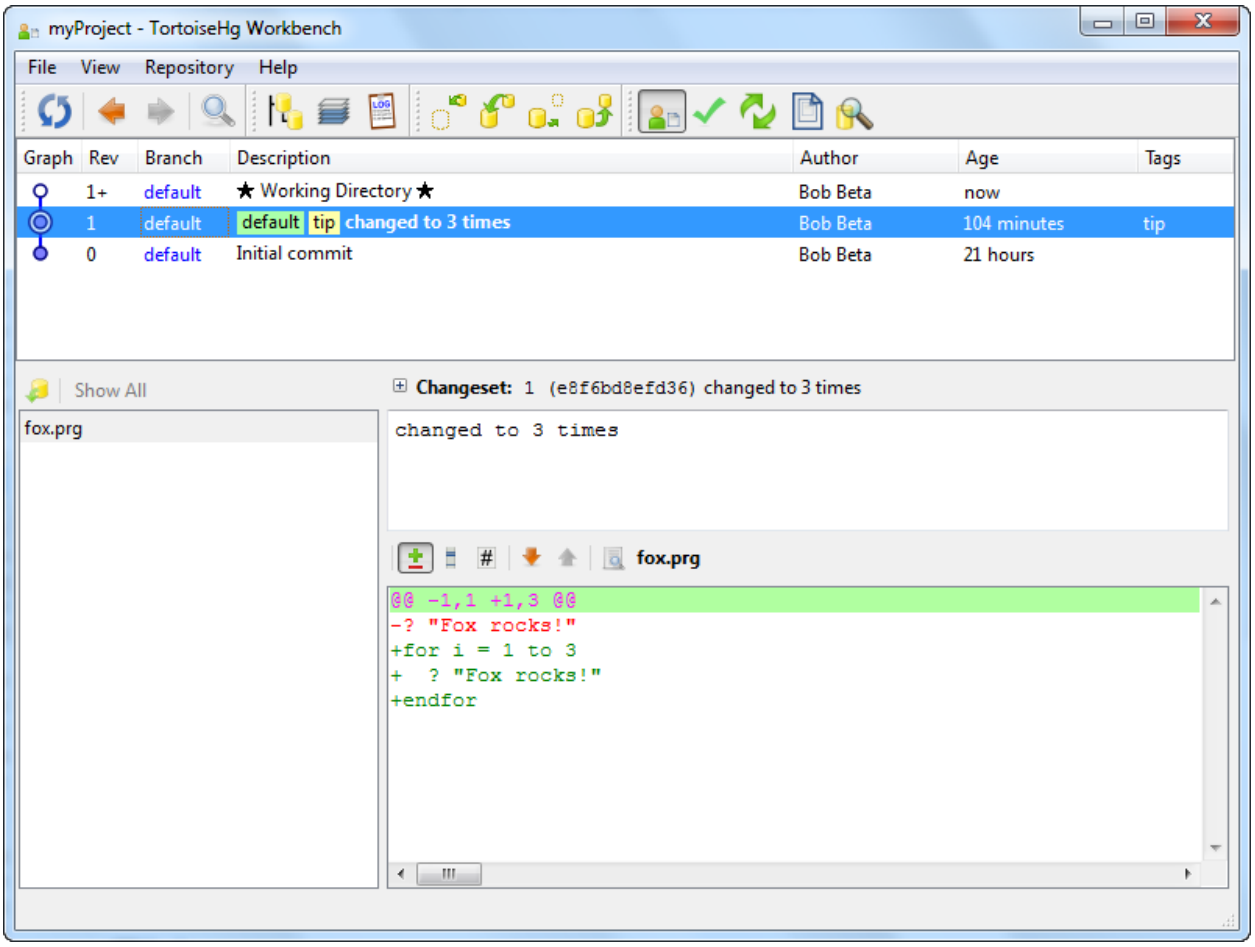


Figure 8: Details of the revision selected in the upper pane in the TortoiseHg Workbench are displayed in the lower panes.

So far Bob has used TortoiseHg only to review changes he already committed from the command line, but he knows that later on he'll most likely stop using the command line and begin using TortoiseHg exclusively for his commits and other interactions with Mercurial.

## *Branching and merging within a repository*

One of the fundamental requirements for any version control system is the ability to create and manage branches. Developers create branches whenever it's necessary or desirable to separate one line of development from another. Mercurial enables branching, but in a way that may be somewhat different from what you're used to if you've used other version control systems.

Mercurial provides two closely related commands for working with branches. The branches command (plural) displays the branches currently in the local repository, while the branch command is used to create a new branch.

Every repository contains a branch named default. If you don't create any other branches you'll most likely never be aware of the default branch's existence, but it's there.

Bob decides to embark on a risky change to his project. Rather than displaying "Fox rocks!" three times, he's going to push the envelope and go for five times. Not wanting to jeopardize the stable version that's already in production, he decides to create a test branch for this change. He first runs the branches command to see which branches are currently in the repository.

```
C:\Bob\myProject>hg branches
default                        1:e8f6bd8efd36
```

Mercurial shows that the default branch is the only branch in the repository, and that its tip is revision 1. Bob then runs the branch command to create a branch named test.

```
C:\Bob\myProject>hg branch test
marked working directory as branch test
```

This is where Mercurial differs from the way some other version control systems handle branching. Where at this point other version control systems might create an entire copy of the working directory, sometimes called a snapshot, Mercurial does not do this. Instead, it simply sets things up so the next changeset that's committed to the repository will be marked as belonging to a new branch called test.

Bob makes the change to fox.prg, and the copy in the working directory now looks like this:

```
for i = 1 to 5
  ? "Fox rocks!"
endfor
```

He now does a commit to the local repository, along with an appropriate comment.

```
C:\Bob\myProject>hg commit -m "changed to 5 times"
```

Curious what the repository history now looks like, Bob runs the log command. He finds that there is now a third changeset, which as expected is revision number 2. Bob also notes that this changeset is marked as belonging to the test branch.

```
C:\Bob\myProject>hg log
changeset:   2:3dbeadce0d3c
branch:      test
tag:         tip
user:        Bob Beta <bob@megasoft.com>
date:        Sun Sep 18 18:07:05 2011 -0500
summary:     changed to 5 times

changeset:   1:e8f6bd8efd36
user:        Bob Beta <bob@megasoft.com>
```

```
date:        Sun Sep 18 12:57:13 2011 -0500
summary:     changed to 3 times

changeset:   0:7792cec862ab
user:        Bob Beta <bob@megasoft.com>
date:        Sat Sep 17 16:45:55 2011 -0500
summary:     Initial commit
```

After extensive testing, Bob determines the change to five times works and is safe to bring back into the main line of development. To do this, he needs to perform a merge.

In a development environment where branches are being used, one branch is usually considered to be the "main" branch; other version control systems may refer to this as the trunk. Typically it's the branch where the source code supporting the stable, release version of the software resides. When merging changes from two branches, you generally want the main branch to be the one on the receiving end of the merge. This is because you want the merged changes to end up in the main line of development. In other words, you want to merge the changes from the test branch into the main branch.

Bob wants to merge that changes from his test branch back into his default branch. Having had some experience with other version control systems, he figures he should use the branch command to switch back to the default branch. He tries this but receives an unpleasant surprise.

```
C:\Bob\myProject>hg branch default
abort: a branch of the same name already exists
(use 'hg update' to switch to it)
```

Why was the operation aborted? As in most cases, Mercurial's message are pretty much self-explanatory. Bob has forgotten that the branch command tells Mercurial to create a new branch with the specified name. Mercurial is saying it can't create a branch named default because a branch with that name already exists. It goes on to suggest the Bob needs to do an update to switch back to the default branch.

Now, Bob knows very well that the default branch contains the version of fox.prg as it existed before he made the change he just committed to the test branch. He thinks it seems counter-intuitive to update the working directory to the older version from the default branch before doing a merge with the newer version from the test branch. After all, updating to the default branch would overwrite fox.prg in the working directory and wipe out the changes he just made, right?

Right. But in fact, that's exactly what he needs to do, and it's OK because the merge will bring the more recent changes back in. Bob issues the update command and specifies he wants to update to the default branch.

```
C:\Bob\myProject>hg update default
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

He then looks at fox.prg in the working directory and sees that, indeed, it's gone back to the older version of three times.

```
C:\Bob\myProject>type fox.prg
for i = 1 to 3
  ? "Fox rocks!"
endfor
```

Now that the working directory is again based on the default branch, Bob completes the merge process by issuing the merge command and specifying the test branch.

```
C:\Bob\myProject>hg merge test
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
(branch merge, don't forget to commit)
```

He again checks the contents of fox.prg in the working directory to be sure it's as expected.

```
C:\Bob\myProject>type fox.prg
for i = 1 to 5
  ? "Fox rocks!"
endfor
```

The last step is to commit the merged file to the repository, as suggested by the message Mercurial displayed after the merge command was run.

```
C:\Bob\myProject>hg commit -m "merged changes from test branch"
```

After doing the commit, the log shows that another changeset has been created, reflecting the merged version of fox.prg.

```
C:\Bob\myProject>hg log -r3:1
changeset:   3:970dcbd576c8
tag:         tip
parent:      1:e8f6bd8efd36
parent:      2:3dbeadce0d3c
user:        Bob Beta <bob@megasoft.com>
date:        Sun Sep 18 21:05:09 2011 -0500
summary:     merged changes from test branch

changeset:   2:3dbeadce0d3c
branch:      test
user:        Bob Beta <bob@megasoft.com>
date:        Sun Sep 18 18:07:05 2011 -0500
summary:     changed to 5 times

changeset:   1:e8f6bd8efd36
user:        Bob Beta <bob@megasoft.com>
date:        Sun Sep 18 12:57:13 2011 -0500
summary:     changed to 3 times
```

Notice that the tip revision (revision 3) has two parents. This is because revision 3 was created as the result of a merge between revision 1 and revision 2.

Running the branches command again at this point shows that the default branch is again the active one.

```
C:\Bob\myProject>hg branches
default                        3:970dcbd576c8
test                           2:3dbeadce0d3c (inactive)
```

This is a good place to take another look at TortoiseHg. Everything Bob just did from the command line – creating a branch in the local repository and then merging changes from it back in to the default branch – he could also have done using TortoiseHg.

Looking at the TortoiseHg Workbench, Bob can see that the changes he made are now included in the revision history in the top-most pane. In addition, he finds that TortoiseHg uses a graphical representation to nicely illustrate the branch and merge operations he just completed.

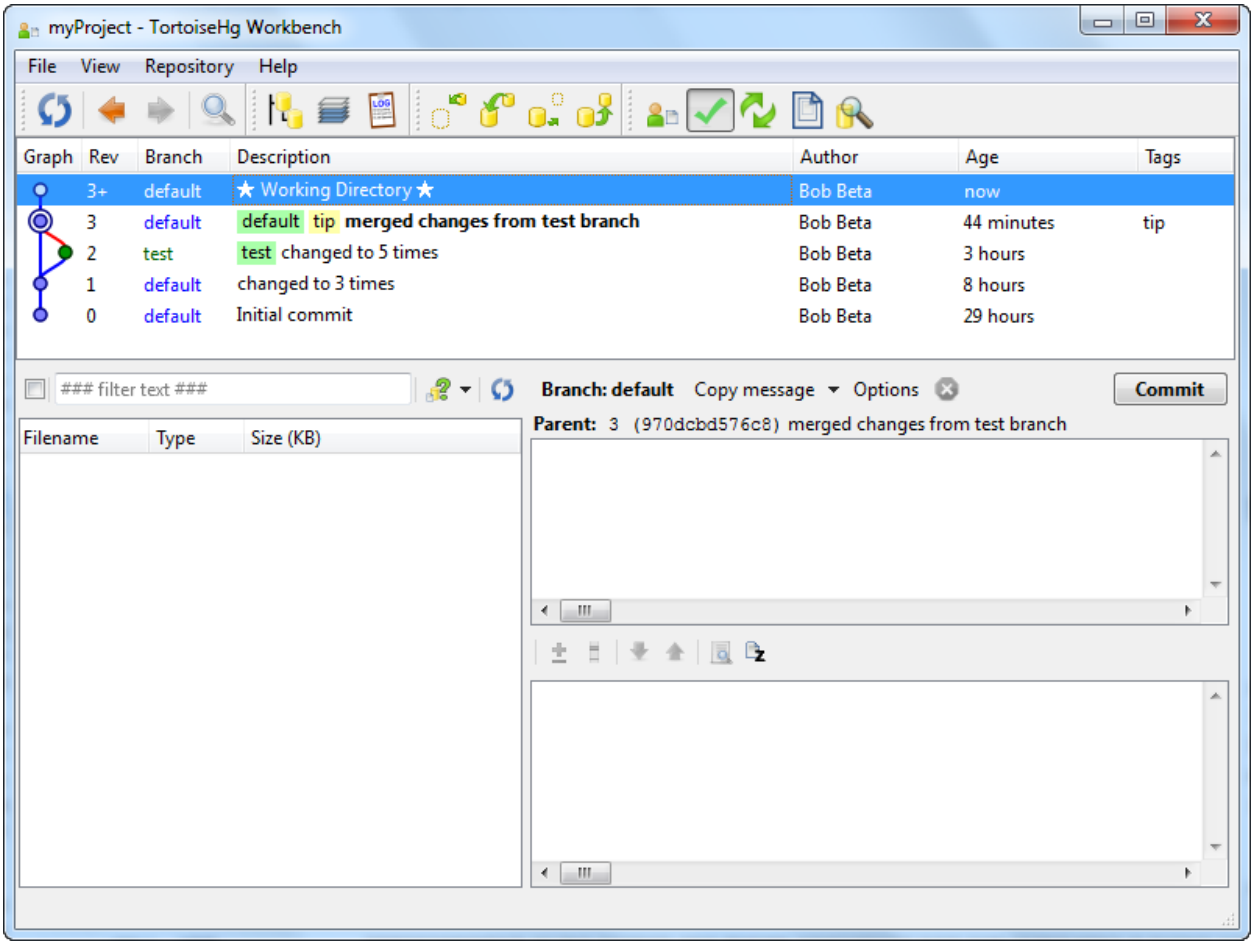

Figure 9: The TortoiseHg Workbench reflects the changes Bob made to his repository. The creation of the test branch at revision 2 and the merge back into the default branch at revision 3 can clearly been seen in the graphical representation of the repository's history.

## *Working with remote repositories*

Much of the power of Mercurial and other distributed version control systems comes from their ability to work with remote repositories. This enables team members to easily share changes with one another while still retaining all the speed and other advantages of using their own local repository.

In the simplest terms, a remote repository is any repository other than your local one. Aside from how it's accessed, a remote repository is no different than a local repository. The fact that it's considered "remote" is simply a matter of the developer's perspective: my local repository could be your remote one, and vice versa, or we could set up a third repository that would be considered remote for both of us.

Mercurial provides the clone, pull, and push commands for interacting with remote repositories.

Megasoft decides to bring Carol in to work with Bob on the development of myProject. Carol will be working on her own machine but will share changes back and forth with Bob.

To get started, she uses Mercurial's clone command to create a local copy of the project for herself. The clone command takes a two parameters, a source and a destination. The destination is optional, and if omitted the current directory is used. Carol types the following command:[5]

```
C:\Carol>hg clone ..\Bob\myProject
destination directory: myProject
updating to branch default
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

Because Carol was working from her C:\Carol folder and did not specify a destination, Mercurial cloned the project into the same subfolder relative to her current directory that it came from on Bob's machine, namely myProject. Carol does a dir command to verify that she does indeed have a copy of the myProject folder and all its contents, including a copy of the local repository with all of the change history to date.

```
C:\Carol>cd myProject

C:\Carol\myProject>dir
 Volume in drive C is OS
```

[5] Because I'm doing all of this on a single machine, I'm using C:\Bob to simulate Bob's machine and C:\Carol to simulate Carol's. In a real world scenario each of them would be using their own individual machines and would be able to access the other's machines over a local area network, an intranet, or the Internet.

```
 Volume Serial Number is 6EDB-0B0A

 Directory of C:\Carol\myProject

09/18/2011  10:50 PM    <DIR>          .
09/18/2011  10:50 PM    <DIR>          ..
09/18/2011  10:50 PM    <DIR>          .hg
09/18/2011  10:50 PM                42 fox.prg
               1 File(s)             42 bytes
               3 Dir(s)  152,899,506,176 bytes free
```

Working on her local machine, Carol makes a couple of changes to fox.prg to bring it into compliance with corporate standards. This includes declaring local variables and using the modified Hungarian naming convention.

```
local lni
for lni = 1 to 5
  ? "Fox rocks!"
endfor
```

When she cloned Bob's repository, Mercurial created an hgrc configuration file in Carol's .hg folder, but it's not identical to the one in Bob's repository. The file Carol gets as a result of the clone command contains the path to the location from which she cloned Bob's repository (think of this as a back-link), but it does not include a username. Before she can commit changes to her local repository she needs to edit her hgrc file and add her own username. In the listing below, Mercurial has supplied the [paths] section and Carol adds the [ui] section.

```
[paths]
default = C:\Bob\myProject
[ui]
username = Carol Coder <carol@megasoft.com>
```

With that done, Carol can commit her changes to her local repository.

```
C:\Carol\myProject>hg commit –m "comply with corporate standards"
```

The log for her local repository now shows the change she committed, along with those from Bob's earlier work.[6]

```
C:\Carol\myProject>hg log -r4:3
changeset:   4:77ef582612ec
tag:         tip
user:        Carol Coder <carol@megasoft.com>
date:        Sun Sep 18 23:35:54 2011 -0500
summary:     comply with corporate standards
```

[6] The –r4:3 parameter tells Mercurial to show only revision 4 (the one Carol just committed) and revision 3.

```
changeset:   3:970dcbd576c8
parent:      1:e8f6bd8efd36
parent:      2:3dbeadce0d3c
user:        Bob Beta <bob@megasoft.com>
date:        Sun Sep 18 21:05:09 2011 -0500
summary:     merged changes from test branch
```

Meanwhile, back in Bob's office, the project manager walks in and tells Bob he needs to bring in the changes Carol just made. Bob knows that Mercurial provides the pull command to get changes from a remote repository. Carol's local repository is a remote repository from Bob's point of view, so he runs a pull command specifying the path to Carol's myProject folder as the source.

```
C:\Bob\myProject>hg pull ..\..\Carol\myProject
pulling from ..\..\Carol\myProject
searching for changes
adding changesets
adding manifests
adding file changes
added 1 changesets with 1 changes to 1 files
(run 'hg update' to get a working copy)
```

At this point, Bob's local repository has been updated with Carol's revision, but his working directory has not been affected. It still contains the last version Bob saved, which does not include Carol's changes. In order to bring his working directory up to date, he needs to run the update command.

```
C:\Bob\myProject>hg update
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

Looking at the copy of fox.prg in his working directory, Bob sees that he now has Carol's changes.

```
C:\Bob\myProject>type fox.prg
local lni
for lni = 1 to 5
  ? "Fox rocks!"
endfor
```

As before, Bob and Carol could have made these changes using TortoiseHg instead of the command line. The TortoiseHg Workbench shows the new history in Bob's repository.

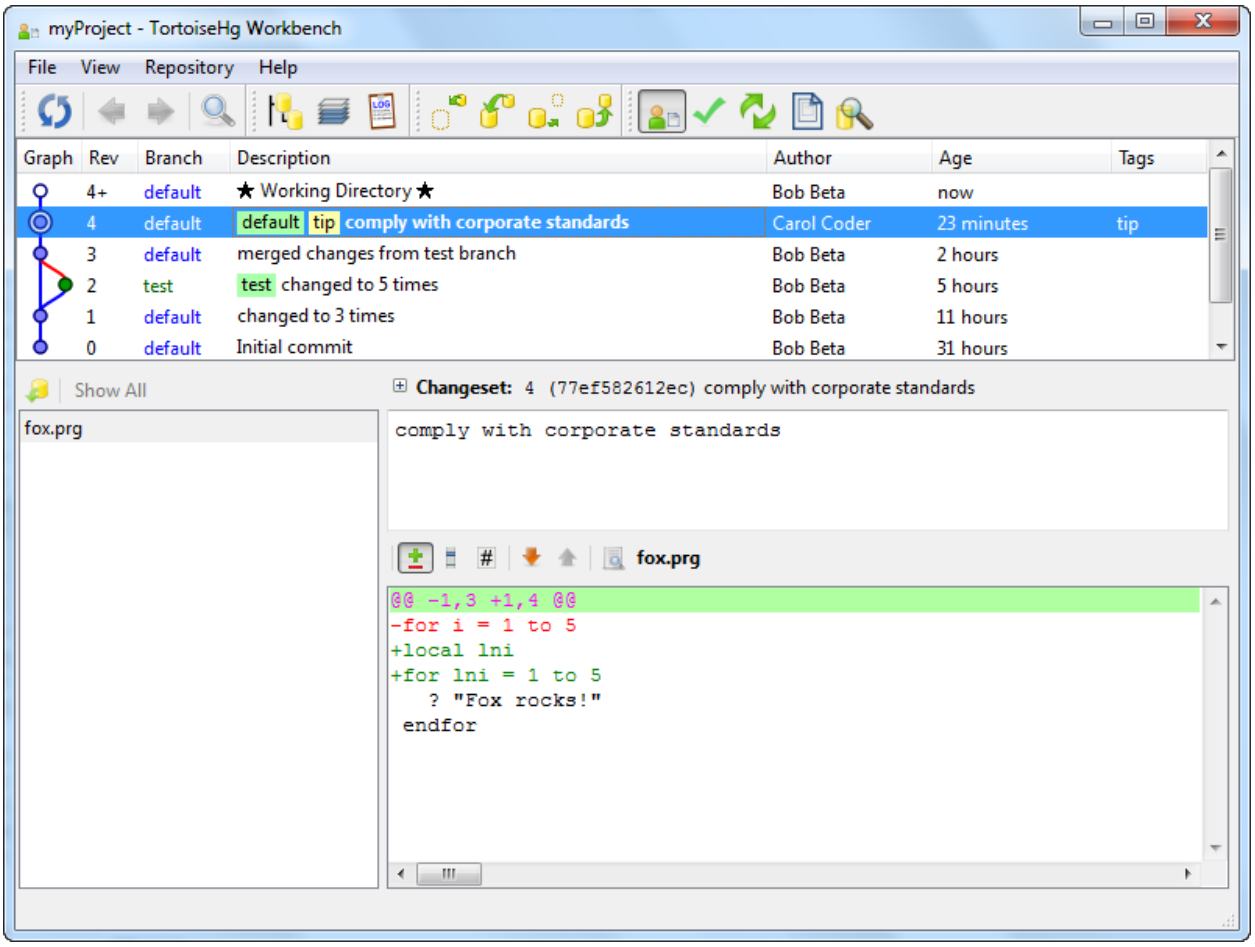


Figure 10: TortoiseHg shows the changes to Bob's repository after pulling Carol's changes.

## *Handling merge conflicts*

In the previous example, Mercurial had no difficulty merging Carol's changes to fox.prg with the version in Bob's repository. However, consider what might happen if Bob makes some changes to his version of fox.prg while Carol makes some other changes to her version. In this case it's possible each of them could make a different change to the same line of code. If this happens, Mercurial has no way of knowing which change to keep when the two are merged together. This results in what's called a merge conflict.

At the end of the previous exercise, Bob had just pulled Carol's changes into his repository and had updated his working directory. At that point, Bob and Carol's both have the same version of fox.prg.

```
local lni
for lni = 1 to 5
  ? "Fox rocks!"
endfor
```

During a high-level, closed-door management meeting, Megasoft's marketing department convinces senior management that the company really needs to ramp up the feature set in myProject in order to increase sales. There is some confusion about exactly what needs to be added, but the meeting breaks up when the donuts run out even though no clear decision has been made. The VP of Marketing walks away excited, while the Project Manager walks away knowing that something needs to be done but not sure exactly what.

On his way back from the meeting, the Marketing VP passes by Bob's office. Not one to waste any time, he sticks his head in the door and tells Bob to increase the "Fox rocks!" display from five times to ten times. Customers will love it! His work now being done for the day, the VP leaves for the golf course, naturally without telling the project manager that he's talked to Bob.

The project manager, on the other hand, considers the situation carefully and decides Carol should be the one to add the new features. He's a little more conservative that the Marketing VP so he tells Carol to increase the display to seven times.

Bob makes the requested change to his copy of fox.prg and commits it to his local repository. Carol makes the changes she was told to make and commits it to her local repository. Their code now looks like this:

```
* Bob's version of fox.prg
local lni
for lni = 1 to 10
  ? "Fox rocks!"
endfor
```

```
* Carol's version of fox.prg
local lni
for lni = 1 to 7
  ? "Fox rocks!"
endfor
```

Bob is in charge of doing the release builds, and company policy says he needs to be sure he pulls changes from the other developers on the team before doing a build in order to insure that nothing gets left out. Bob pulls the latest from Carol's repository.

```
C:\Bob\myProject>hg pull ..\..\Carol\myProject
pulling from ..\..\Carol\myProject
searching for changes
adding changesets
adding manifests
adding file changes
added 1 changesets with 1 changes to 1 files (+1 heads)
(run 'hg heads' to see heads, 'hg merge' to merge)
```

Hmm. Mercurial is saying something about "+1 heads" and displays a message about running hg heads and hg merge. But the last time Bob pulled changes from Carol's machine, he simply had to do an update command. He decides to try that again this time.

```
C:\Bob\myProject>hg update
abort: crosses branches (merge branches or update --check to force update)
```

Well, that didn't work! Something's clearly different this time. Bob decides to run the hg heads command as Mercurial suggested.

```
C:\Bob\myProject>hg heads
changeset:   6:927666924516
tag:         tip
parent:      4:77ef582612ec
user:        Carol Coder <carol@megasoft.com>
date:        Mon Sep 19 19:57:25 2011 -0500
summary:     changed to 7 times

changeset:   5:1ba4106239cc
user:        Bob Beta <bob@megasoft.com>
date:        Mon Sep 19 19:56:49 2011 -0500
summary:     changed to 10 times

changeset:   2:3dbeadce0d3c
branch:      test
user:        Bob Beta <bob@megasoft.com>
date:        Sun Sep 18 18:07:05 2011 -0500
summary:     changed to 5 times
```

Reading up from the bottom of the list, Bob recognizes head revision 2 as the one he created earlier in his test branch. It's realizes that it's still part of his repository history, but understands that it's of no concern here. He focuses on revisions 5 and 6, which are the two heads in question. He can see that head revision 5 is the one he just committed; it carries his username and his commit message noting that he changed to 10 times. Finally, he can tell that head revision 6 at the top of the list is the result of his pulling changes from Carol, because it carries her username and a different summary.

Bob is beginning to get the picture. The default branch of his local repository now contains two heads representing two different sets of changes. Using TortoiseHg, Bob can see this condition in the graphical display.

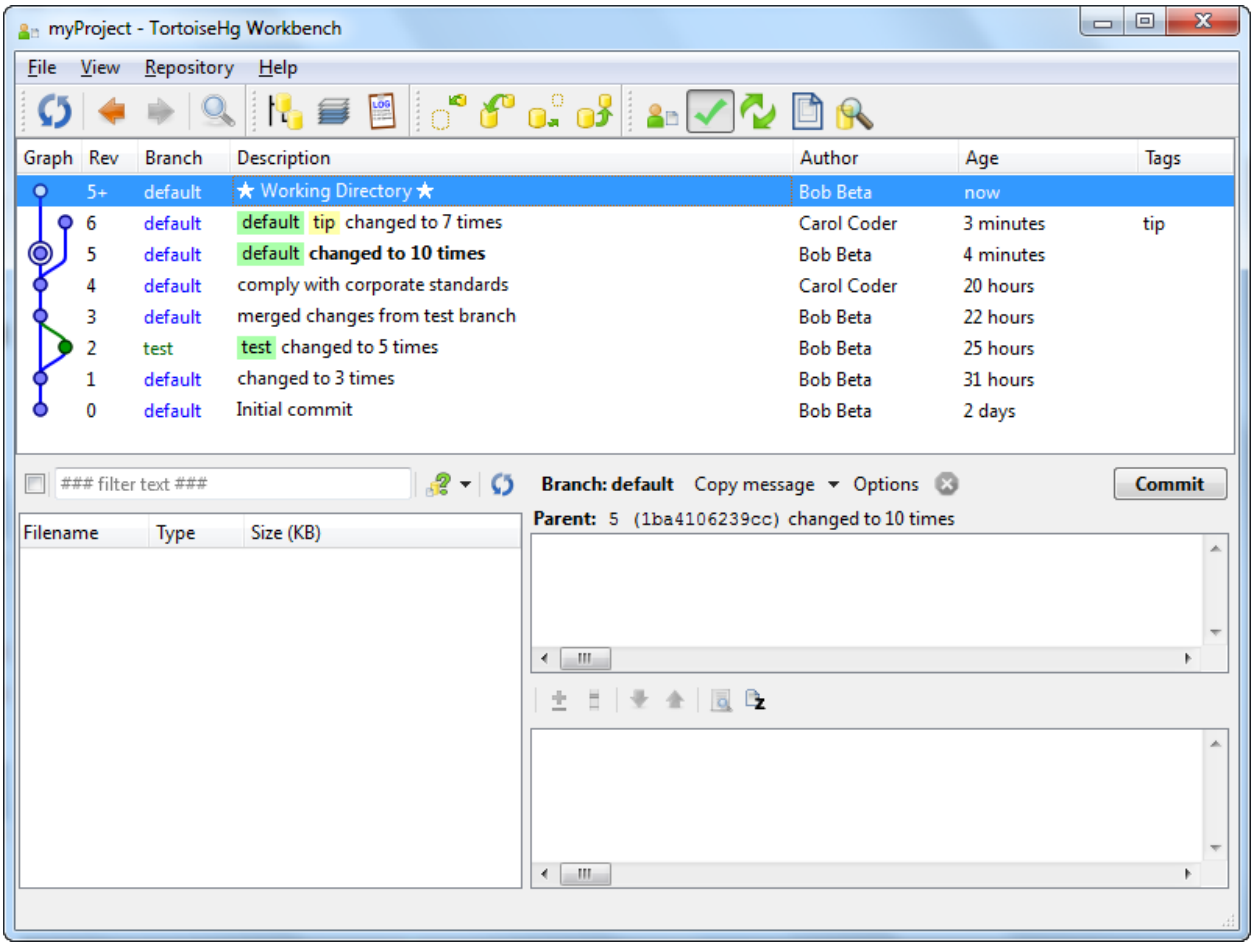


Figure 11: The graphical display in TortoiseHg shows that Bob's local repository has two heads, one that he created (rev 5) and the other (rev 6) created by Carol, which was added to Bob's repository as a result of the pull.

In addition to suggesting that he run the heads command, Mercurial also suggested that Bob needed to do a merge. So that's what he does.

```
C:\Bob\myProject>hg merge
merging fox.prg
```

What Mercurial does at this point depends on which tool, if any, Bob has set up to handle merge conflicts. Bob has been a long time fan of the excellent Beyond Compare utility from Scooter Software, and has configured Mercurial to use it as his default merge conflict resolution program. Because there is in fact a merge conflict in this example, Mercurial opens Beyond Compare's three-way merge tool so Bob can see and resolve the conflict.

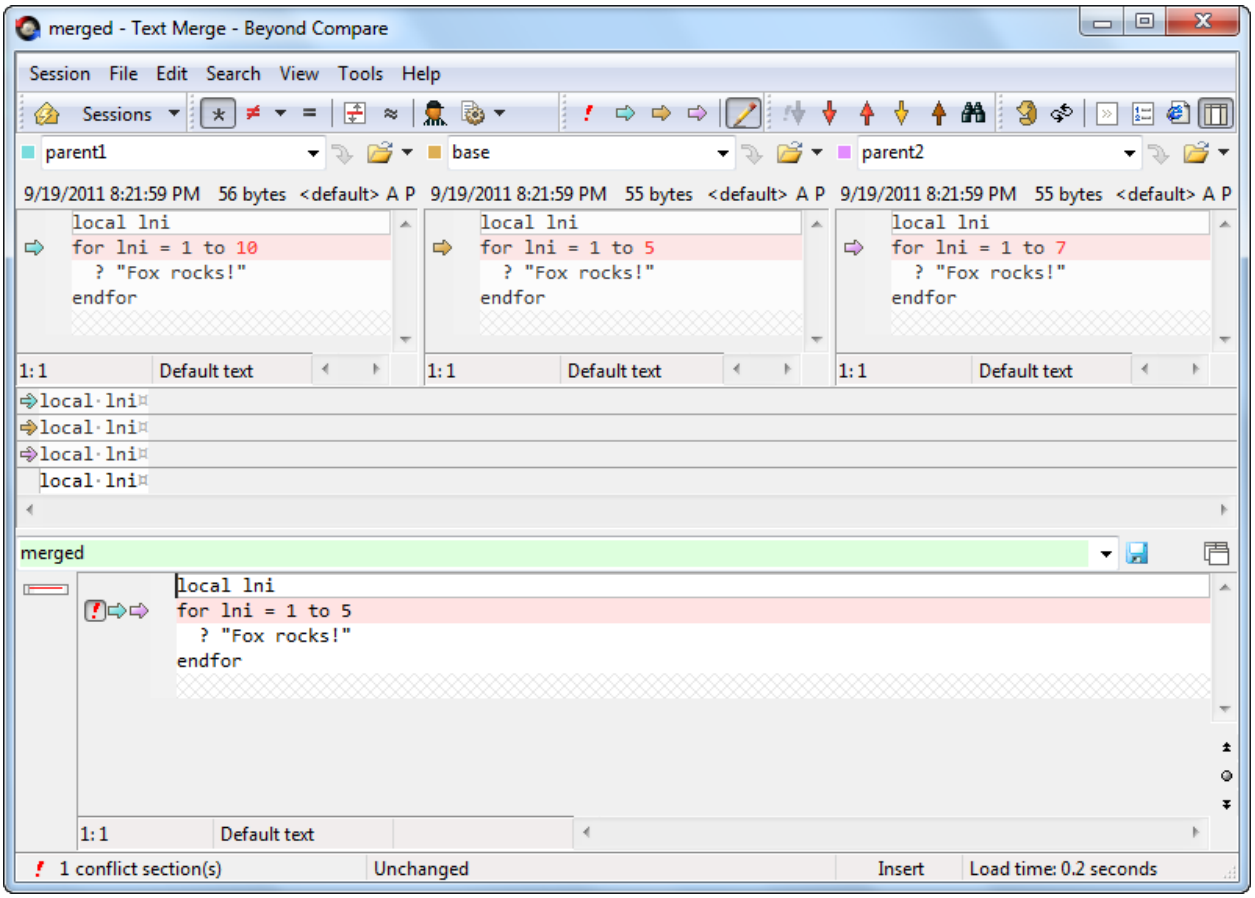


Figure 12: Mercurial can be configured to use Beyond Compare's three-way merge tool to resolve merge conflicts.

Whatever merge tool you use, the concept is the same: there is a base version of the line or lines in conflict (top center, in Figure 12), there are the two competing changed versions (top left and top right), and there is some way (the edit panel, bottom) for you to edit the file and resolve the conflict.

In this case, Bob makes a command decision: he decides to compromise between the two changes and display the message 8 times. Carol won't care, since she did what she was told, and after a few beers on the golf course the Marketing VP won't remember what he told Bob anyway. Bob makes the change to the merged version in the bottom panel and saves the edited file, as shown in Figure 13.

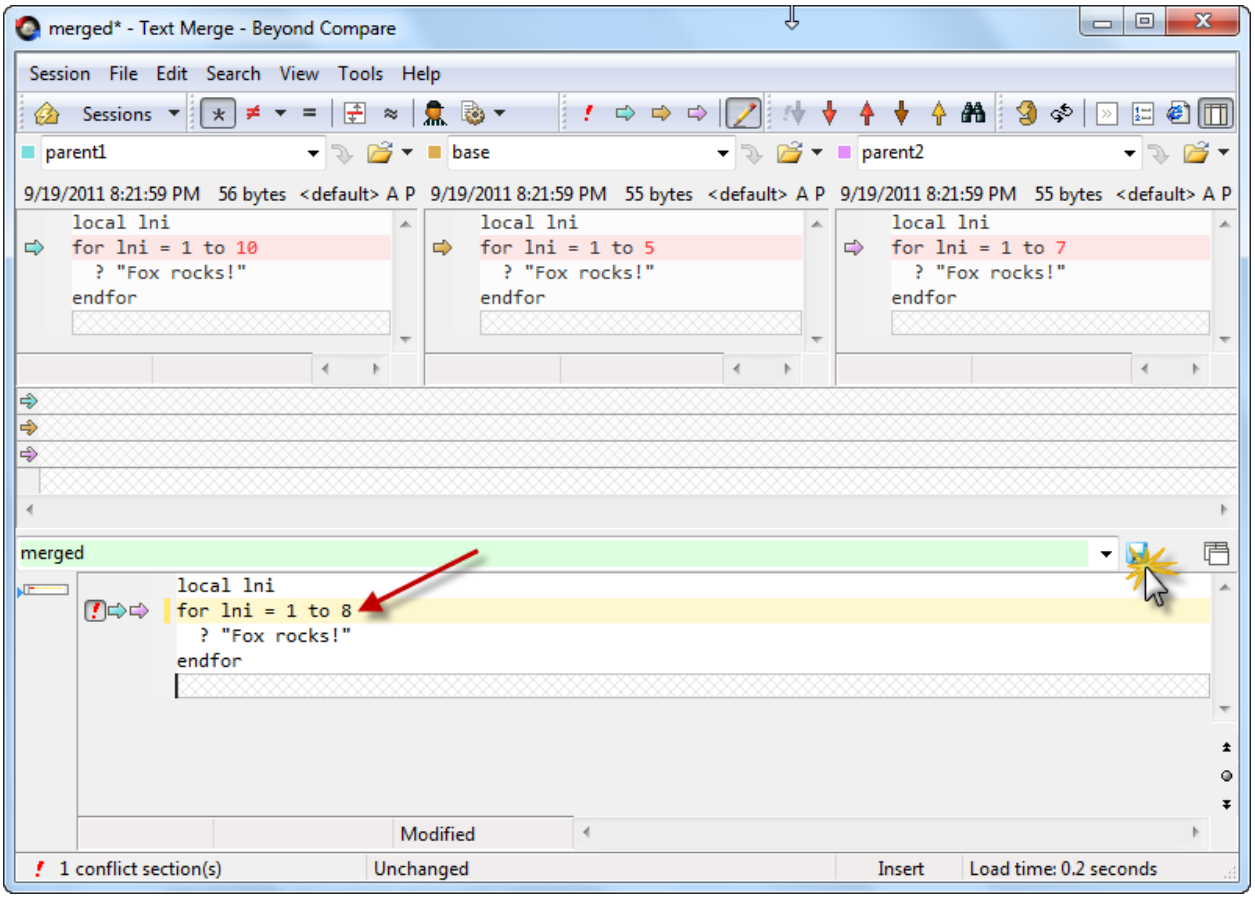


Figure 13: When a merge conflict occurs, the developer needs to make the change(s) necessary to resolve the conflict and then save the merged file.

While Bob has been working in Beyond Compare, Mercurial has been is a suspended state waiting on a resolution to the merge conflict. When Bob saves the merged file and closes Beyond Compare, Mercurial detects that the conflict has been resolved and completes its merge operation.[7]

```
C:\Bob\myProject>hg merge
merging fox.prg
0 files updated, 1 files merged, 0 files removed, 0 files unresolved
(branch merge, don't forget to commit)
```

The version of fox.prg that Bob just saved is now stored in his working directory, but his local repository doesn't have a record of that change yet. That's why Mercurial prompted Bob to do a commit, which he does.

[7] In this example, Mercurial knew the conflict had been resolved because it was set up to be integrated with Beyond Compare. If Bob had not been using an integrated tool and had simply edited fox.prg manually, Mercurial would have had no way of knowing the conflict was resolved. Mercurial provides a resolve command to handle this situation. Bob would have run hg resolve fox.prg to let Mercurial know the conflict had been resolved.

```
C:\Bob\myProject>hg commit -m "resolved merge conflict, changed to 8 times"
```

Bob could now run the log command to see a record of the most recent changes. He can also view the history in the TortoiseHg graphical interface, as shown in Figure 14. Either way, he sees that there is a new revision 7 representing the resolution of the merge conflict.

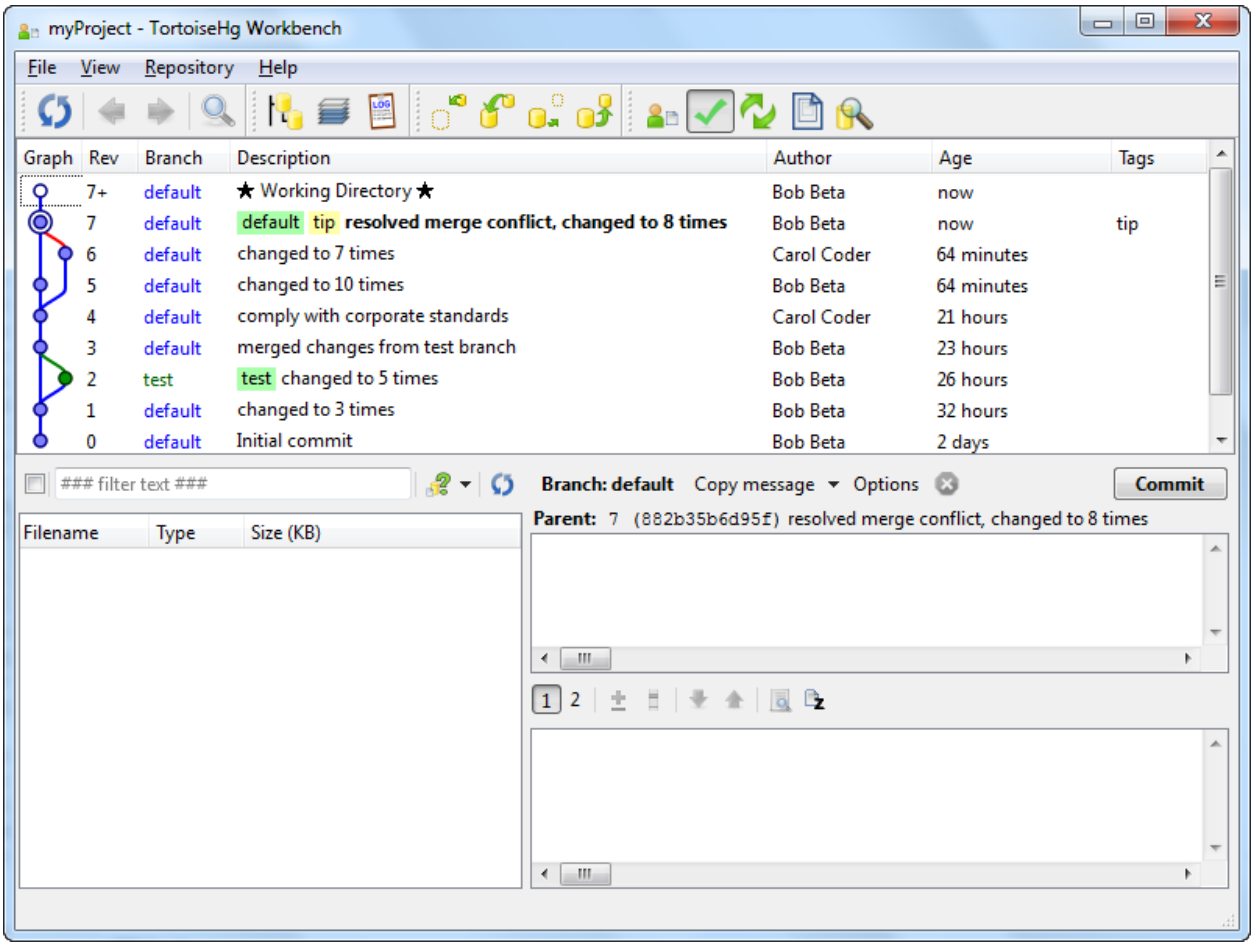


Figure 14: The TortoiseHg graphical display illustrates the merge conflict at revisions 5 and 6, and its resolution at revision 7.

Let me stress again that everything that's been done from the command prompt in these examples could also have been done using TortoiseHg. Don't get the impression that the TortoiseHg Workbench is only for viewing – it's a completely interactive tool for working with Mercurial. I have chosen to introduce Mercurial's behavior using its commands from the command prompt because I believe that's a better way to learn the fundamentals. Having now learned about the individual commands, you'll have a better understanding of what TortoiseHg is doing behind the scenes.

# Special Considerations for VFP

## What to include in the repository

One of the first considerations when starting to use a version control system is which files to include in the repository. When you read about version control systems, the general rule of thumb is to include only files that you can edit directly, such as program source code files. There's no point in including compiled object code files, executable files, and other files of that nature since you can't edit them directly and they're derived from the source code files anyway.

On the other hand, if you want to enable other developers to create a clone and work on your project, you need to include everything necessary for them to edit the source code and build the project. For Visual FoxPro developers this means including forms, reports, menus, visual class libraries, etc. The issue when it comes to tracking changes to these types of files in a version control system is that they are binary files, not text files, so they require special handling. I'll come back to the issue of binary files a bit later on. For now, let me just recommend that you include them in your local repository.

You will also want to include any header files (.h,), configuration files (.ini), and so on that are subject to change over time and are necessary for building and running your project. In addition, you may want to include other files you might not initially think about, such as your Inno Setup scripts, readme.txt file, and so on.

Last but not least, don't forget to include the VFP project files (.pjx and .pjt).

## What to ignore

When you run the hg add command without any parameters, Mercurial detects and offers to include in the repository every file it finds in your working directory. However, there are typically several types of files whose changes you'll never want to track. This includes things like backup files, temporary files, compiled files, zip archives, and so on.

Mercurial provides a mechanism whereby you can tell it which files it should always ignore. Not surprisingly, it's called the .hgignore file. This is simply a text file where you can list all of the individual files, groups of files, and/or folders you want Mercurial to ignore.

The download files for this session include a generic .hgignore file for Visual FoxPro applications, which you can use a starting point for your own. Note that the file names and file extensions in the .hgignore file are case sensitive, so most of them appear twice – once in lower case and once in upper case. This is the only place I've found where case matters.

It's a good idea to create the .hgignore file for a project before you add files to its repository for this first time. However, if you find you have left something out of the .hgignore file and end up with a lot more files in the repository than you wanted after the initial commit, you can always blow away the local repository and start over by simply erasing the .hg folder.

After the initial commit, you can still make changes to the .hgignore file and they will take effect on the next commit, but they won't affect history already in the repository from earlier commits.

Documentation on the syntax of the .hgignore file can be found at http://www.selenic.com/mercurial/hgignore.5.html.

## What to do about binary files

As every VFP developer well knows, Visual FoxPro stores the source code for forms, reports, menus, labels, and visual class libraries in tables. Tables are binary files, not plain text files. When it comes to version control, the issue with binary files is that you can't do a diff/merge on them. The ability to view a diff and do a merge between two sets of changes is a fundamental part of using a version control system, so this presents a problem for Visual FoxPro developers.

The standard solution to this problem is to use a tool that can create a plain text version of the source code from the table, and vice versa. After changes are made to source code stored in a table, the tool is used to create a plain text version and the plain text version is then committed to the repository.[8] When it's time to build the project, the same tool can be used in reverse to re-create the table file from the plain text file.

There are several such tools in use, most of which have been around for a long time. Using them with Mercurial is no different than using them with any other version control system. I've listed some of the popular tools below, along with a few brief comments. For further information, refer to the source for each of these tools. Many of them have also been covered in various VFP-related publications over the years, so there's no lack of information.

**SCCText** is the tool that ships with Visual FoxPro. It reads the source code from a VFP binary source code file (the table) and writing it to a plain text file. These plain text files are written out to disk with an "a" in place of the "x" in the file name extension – for example, .sca for .scx files, .vca for .vcx files, etc. For that reason they are commonly referred to simply as the "a" files.

The SCCText program is a file named scctext.prg in your C:\Program Files\Microsoft Visual FoxPro 9 folder. It's designed to work with a single file at a time. Since it's a VFP .prg file, you can open it and look at the source code to see how it works, and make modifications if you want to.

Because of some of the limitations of this tool, many VFP developers have looked for and developed alternatives over the years.

[8] You will probably also want to include the binary files in the repository, but that's a separate issue and a separate decision.

**Matt Slay** recently published a Visual FoxPro wrapper class for SCCText.prg to create the "a" files for all the binary files in a project in one step. Matt describes this project in his blog post at http://mattslay.com/foxpro-class-to-generate-scctext-for-all-files-in-a-project and has made the source code available for download from http://codepaste.net/9yy1gm.

As it stands, Matt's class creates the "a" files for all the forms, reports, and visual class libraries referenced in the project. This comprises not only the files in the project folder and its subfolders, but also files that exist outside of the project folder such as native VFP class libraries like the FFC classes and any framework classes you may be using in your project. As a result, you may end up with "a" files for things you don't want or need to store in your repository. While this isn't really a problem, it would be more efficient if the class had an option to exclude files that reside outside of the project folder.

**Alternate SCCText**, also known as **SCCTextX**, is a community based enhancement to the standard SCCText program. The project manager for SCCTextX is well-known VFP developer Jürgen "wOOdy" Wondzinski. SCCTextX is available for download from VFPX.

SCCTextX offers several advantages over SCCText. Quoting from the VFPX webpage:

> There are several improvements made to the original SCCText.prg included in Visual FoxPro:
>
> - Consistent and case insensitive sorting of methods, objects, and properties
> - Corrects several bugs in the original version shipped from Microsoft
> - Optimizations
> - German, Spanish and French localizations

Rick Schummer wrote an article about SCCTextX in the January 2010 issue of FoxRockX, which is a great source for more information about this tool.

**TwoFox** is a two-way conversion tool written by Christof Wollenhaupt. It differs from SCCText and SCCTextX in that it creates XML files instead of plain text files, but its purpose and use are essentially the same as the other tools. TwoFox is available for download from www.foxpert.com/downloads.htm.

Toni Feltman wrote about a set of tools she developed to help her work with Subversion, which she published along with her paper Introduction to Subversion and Tortoise SVN for Southwest Fox 2009. These utilities could be adapted to assist developers working with Mercurial, too. Toni has generously granted permission to distribute these tools in the downloads for this session. They're included as FeltmanT_Subversion_Tools.zip.

## The effect of recompiling all files

Version control tools typically use a file's date-time stamp to determine if the file has been changed. In Visual FoxPro, it's a best practice to do a Recompile All Files before releasing an update, if not more often during development. Marking the Recompile All Files check box when you build a project tells VFP to regenerate the object code stored in visual class libraries, forms, reports, and the other binary files. The updated files then get written to disk with an updated date-time stamp.

Because the date-time stamp has changed, version control systems detect these files as being newer even if the source code did not actually get changed. As a result, these files are included in the next commit to the repository even though they're effectively unchanged from the previous version. This behavior is not unique to Mercurial, but Mercurial is no exception.

There's no way around this that I'm aware of – it's just a fact of life when working with VFP projects. Fortunately it's not really a problem. The only downside I can see is that the changesets include things that really didn't get changed, and therefore the repository grows larger over time than it otherwise would. Just don't be surprised at the number of files that show up as modified after you build your project with Rebuild All Files marked.

## Case sensitivity

As VFP developers are well aware, the VFP project manager sometimes changes the case of file names. For example, a file you create and save as myClassLibrary.vcx becomes myclasslibrary.vcx. This would cause problems with case sensitive version control systems because such systems would treat them as two different files.

Mercurial respects the case sensitivity of the operating system on which it's running. The Windows operating system is case preserving but case insensitive. This means a file can be stored as myClassLibrary.vcx or as myclasslibrary.vcx and Windows knows it's the same file either way. Of course, on a Windows system both file names cannot exist in the same folder at the same time anyway.

Fortunately for us as VFP developers working on Windows, case sensitivity is not a problem with Mercurial. Quoting from p. 73 of Mercurial: The Definitive Guide (PDF):

> ### 7.7.1. Safe, portable repository storage
>
> Mercurial's repository storage mechanism is **case safe**. It translates file names so that they can be safely stored on both case sensitive and case insensitive file systems. This means that you can use normal file copying tools to transfer a Mercurial repository onto, for example, a USB thumb drive, and safely move that drive and repository back and forth between a Mac, a PC running Windows, and a Linux box.

## Integration with the VFP project manager

Like many modern version control systems, Mercurial does not support the interface necessary to integrate it with the Visual FoxPro project manager.

In my opinion, this is not a bad thing.

Although integrating your version control system with the VFP Project Manager may provide some measure of convenience, there is also a downside. First of all, that interface is designed around the centralized version control system idea of checking files in and out, concepts which don't apply to a distributed version control system. Second, there can be a performance penalty when working with large projects. And finally, if you're using a remote repository and the connection to that repository is unavailable, you're pretty much dead in the water because you can't check out any files to work on them. Sure, there are workarounds, but it gets messy.

My general rule of thumb, based on my own experience and from talking to others, is not to integrate your version control system with the VFP Project Manager even if your VCS does support it. Naturally your opinion may vary, but if you're using Mercurial the decision has already been made for you because you can't do it even it you wanted to.

## Integrating Mercurial into your daily development workflow

You've seen what Mercurial can do for you, so you decide to install it and begin using it in your daily Visual FoxPro development work. How do you get started?

### *Step 1 – Create a local repository*

Assuming you have an existing Visual FoxPro project you want to place under Mercurial version control, the first step is to create the local repository. This is easily done from the Windows Explorer context menu. The working directory will be the existing root folder of your project, for example C:\SWFox2011\Sessions\Mercurial\myVFPApp. Right-click on that folder in Windows Explorer and choose Create Repository Here from the TortoiseHg menu pad.

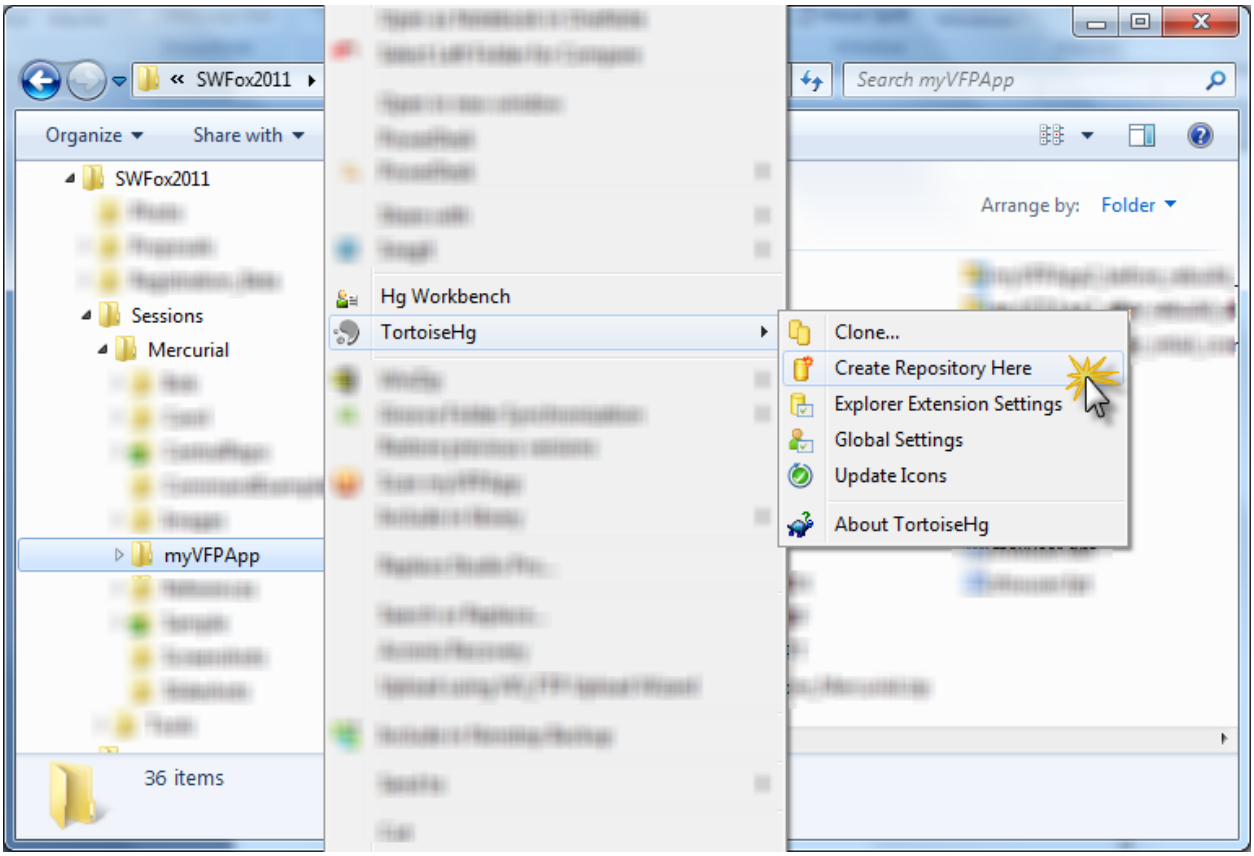


Figure 15: Much of what you'll want to do with Mercurial, such as creating a new repository, is available from the context menu in Windows Explorer.

A small dialog window opens and presents a couple of options – you can simply accept the defaults. When you click the Create button TortoiseHg issues the init command to Mercurial, which in turn creates the .hg local repository folder in your working directory.

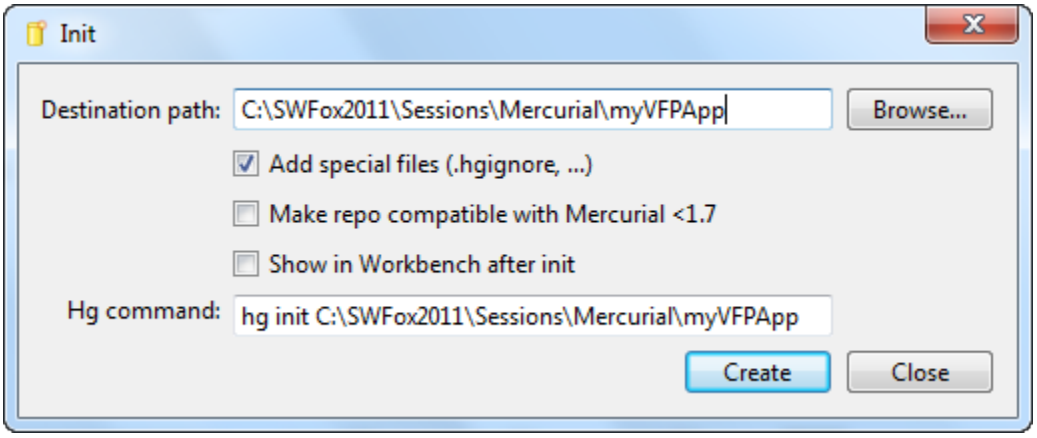


Figure 16: When you use TortoiseHg to create a new repository, it offers a couple of options and shows you the hg init command it will run.

## *Step 2 – Enter your configuration settings*

You need to specify a username before you can do any commits to the new local repository. Many other settings are available, but the username is the only one required at this point.

To set your username using TortoiseHg, right-click on your project's working directory in Windows Explorer and choose TortoiseHg Workbench from the context menu. Then choose Settings from the Workbench window's File menu. In the TortoiseHg Settings dialog (Figure 17) you can enter both your global settings, which act as the default for all repositories on your machine, and your local settings, which are specific to each individual project. The Settings File line indicates which file you're working with: global settings are stored in the mercurial.ini file under your user profile, while local settings are store in the hgrc file in the project's local repository.

Select the global settings tab in the TortoiseHg Settings dialog, then click on the Commit item in the list on the left. Enter your name and email address in the username field, as illustrated in Figure 17. You can enter anything you want here, but the common format is your first name and last name, followed by your email address enclosed in angle brackets.

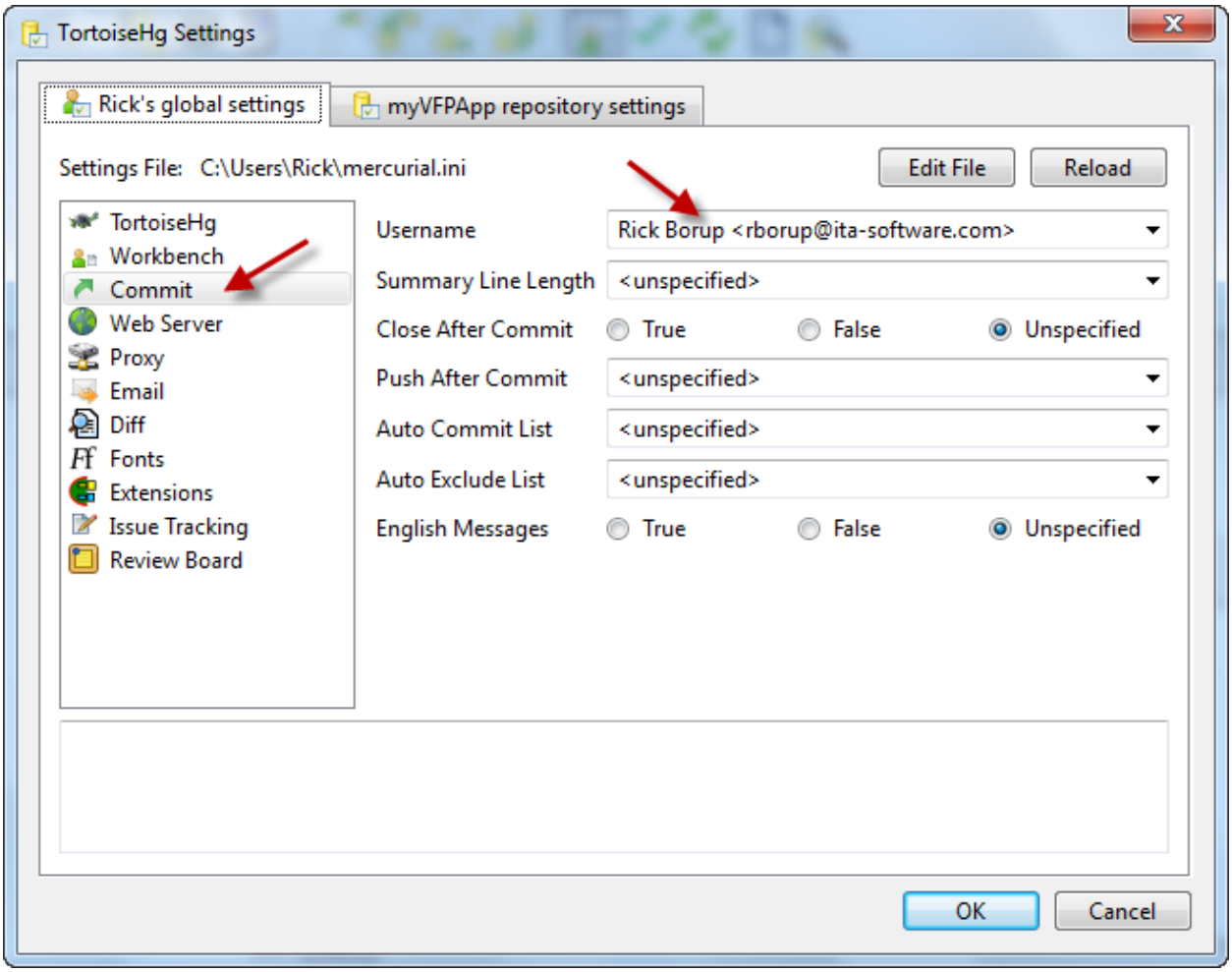


Figure 17: The TortoiseHg setting dialog enables you to set both global and project-specific options.

For now, that's all you need to do. Click OK to save this information.

## Step 3 – Set up your .hgignore file

I recommend you set up the .hgignore file for each project before doing the initial commit. That way Mercurial knows which files to exclude when you tell it to add files for the first time. If you left the default option checked when you created the repository in Step 1, TortoiseHg created an empty .hgignore file for you. Otherwise you can create one manually. Either way, it's just a text file so you can edit it using any text editor, including the file editor in VFP.[9]

I find it helpful to have a boilerplate .hgignore file for VFP projects stored somewhere on my machine so I can simply copy it into the working directory when I place a VFP project under Mercurial version control. I've included one in the session downloads, which you're free to use and to modify for your own purposes.

## Step 4 – Add files and do the initial commit

In TortoiseHg Workbench, select the * Working Directory * line in the list of revisions in the upper-most panel. Because this is a brand new repository to which nothing has yet been committed, the Working Directory line is the only entry in the list and is labeled as rev -1.

When you select the Working Directory in the revision list, the lower left panel displays all the files Mercurial has detected in your project's working directory. This list of files already excludes the ones Mercurial knows it should ignore, based what you put in your .hgignore file; that's why you wanted to set up the .hgignore file before getting to this point in the process.

All of the files that TortoiseHg lists are initially marked with a status of "?", meaning Mercurial considers them to be "unknown". This is because you haven't yet told Mercurial that you do in fact want to track these files. Doing so is part of the next step.

[9] Some text editors automatically add a .txt file name extension if an extension is not specified. Be sure not to let this happen. Mercurial will not recognize a file named .hgignore.txt – it must be named .hgignore.

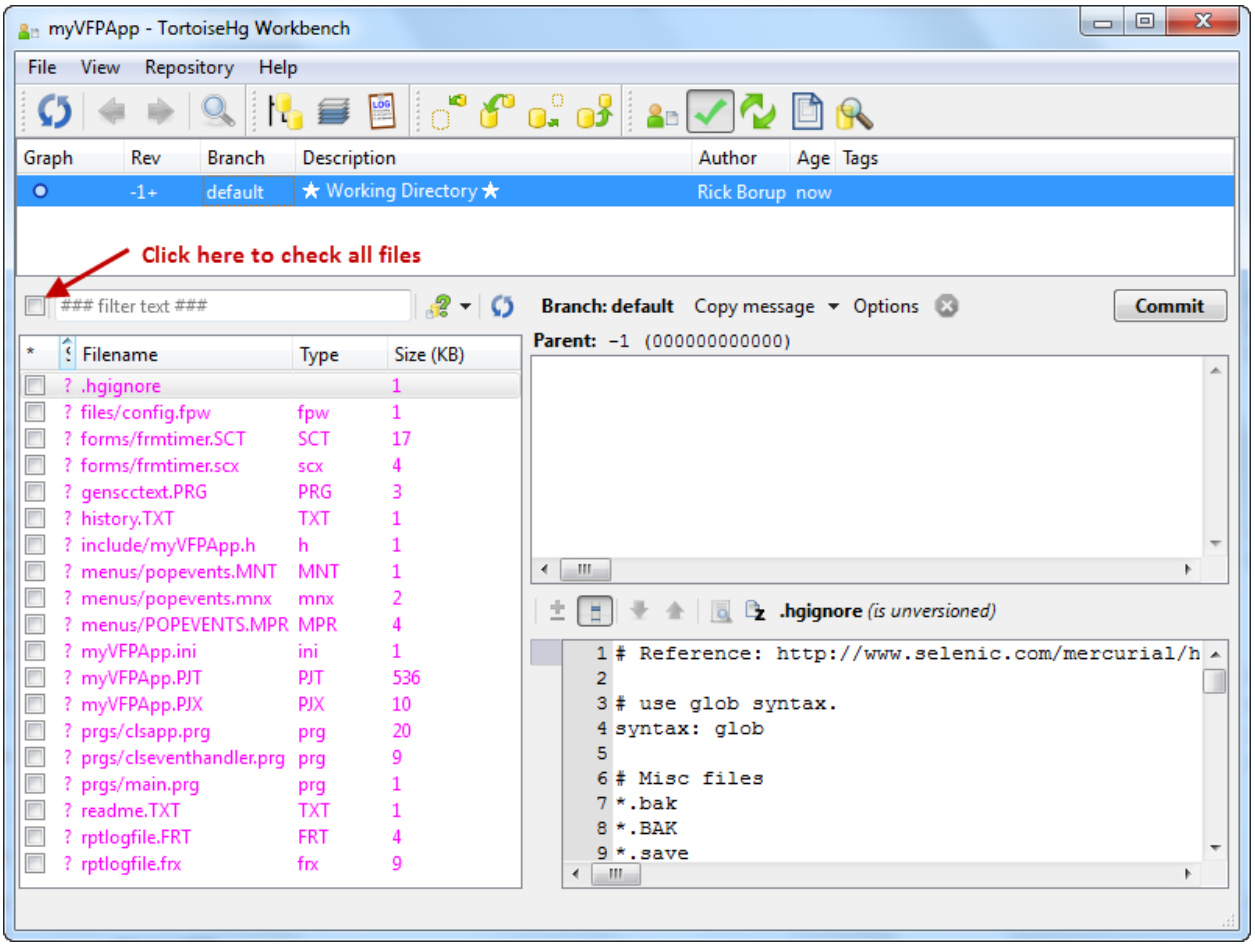


Figure 18: The lower left panel in the TortoiseHg Workbench shows the status of the files in the working directory.

TortoiseHg makes it easy to do both the add and the commit in one step. First, mark the check box for every file you want to add to the local repository. You can mark them individually, or you can simply mark the “check all files” check box above the list (see Figure 18) and TortoiseHg will mark them all for you.

Note that the .hgignore file itself is included in the list. It’s a good idea to mark the check box for this file so Mercurial will track the history of changes to it along with those to your project’s other files.

After reviewing the list of files to be sure it’s the way you want it, enter a commit message in the field to the right of the list and click the Commit button. A brief comment such as “Initial commit” is generally considered to be appropriate for the first commit to a new repository.

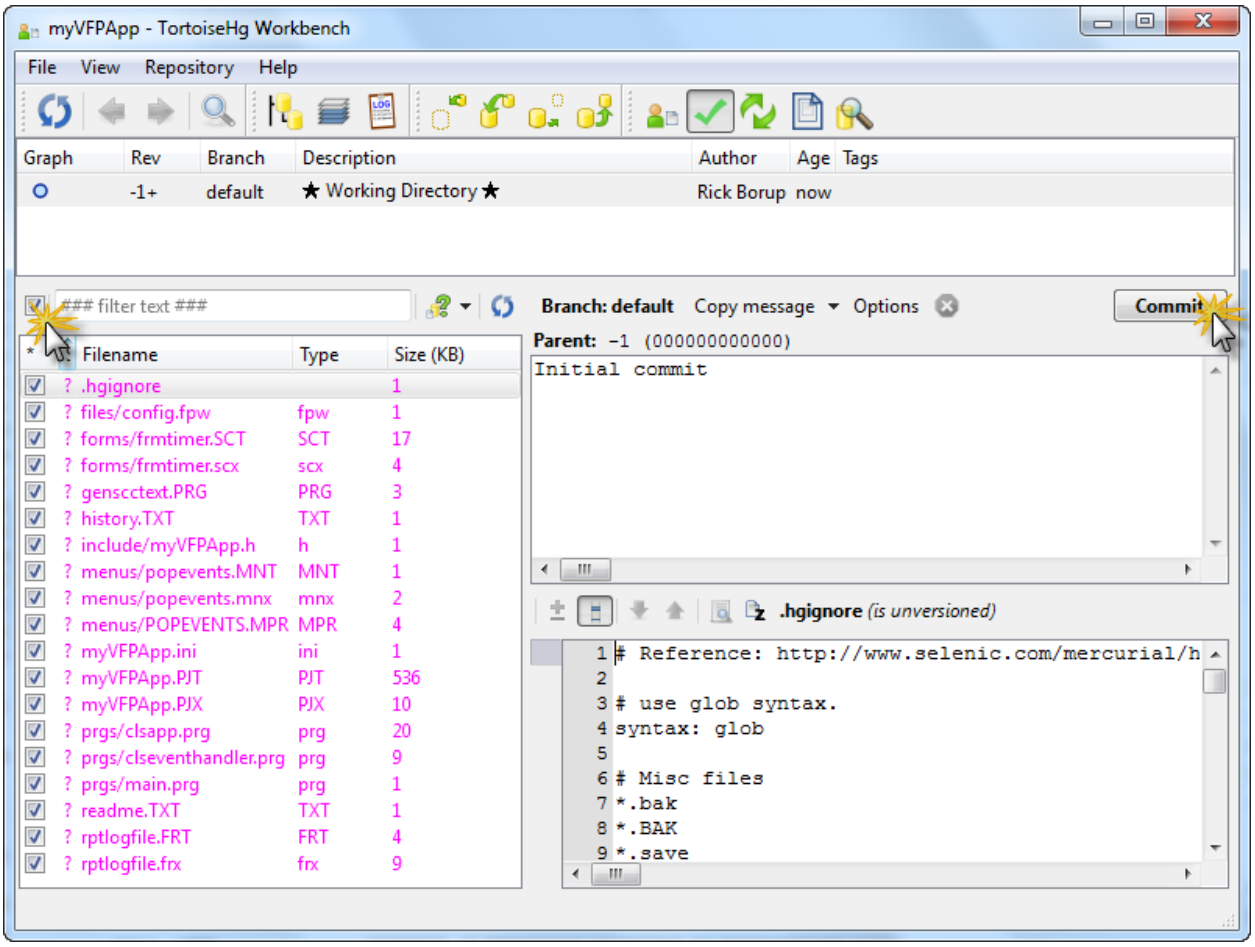


Figure 19: After marking all the files whose changes you want to track, enter a commit message and click the Commit button to add these files to your local repository.

TortoiseHg displays a dialog asking if you want to add the selected untracked files. This is how it does the add and commit in one step. Click the Add button to proceed.

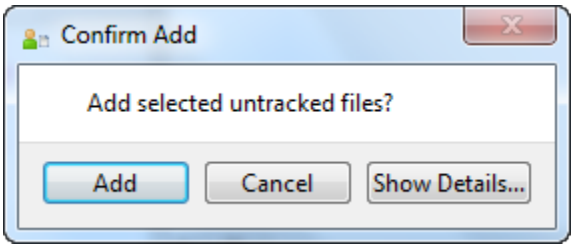


Figure 20: If you've marked untracked files to be included in the commit, TortoiseHg prompts you to add them.

And that's it! Your project is now under Mercurial version control, and the current versions of all tracked files are now recorded in the local repository. Looking at the upper-most panel in TortoiseHg, you can see that the local repository now contains changeset 0, which is marked as the tip revision and which carries your commit message of "Initial Commit".

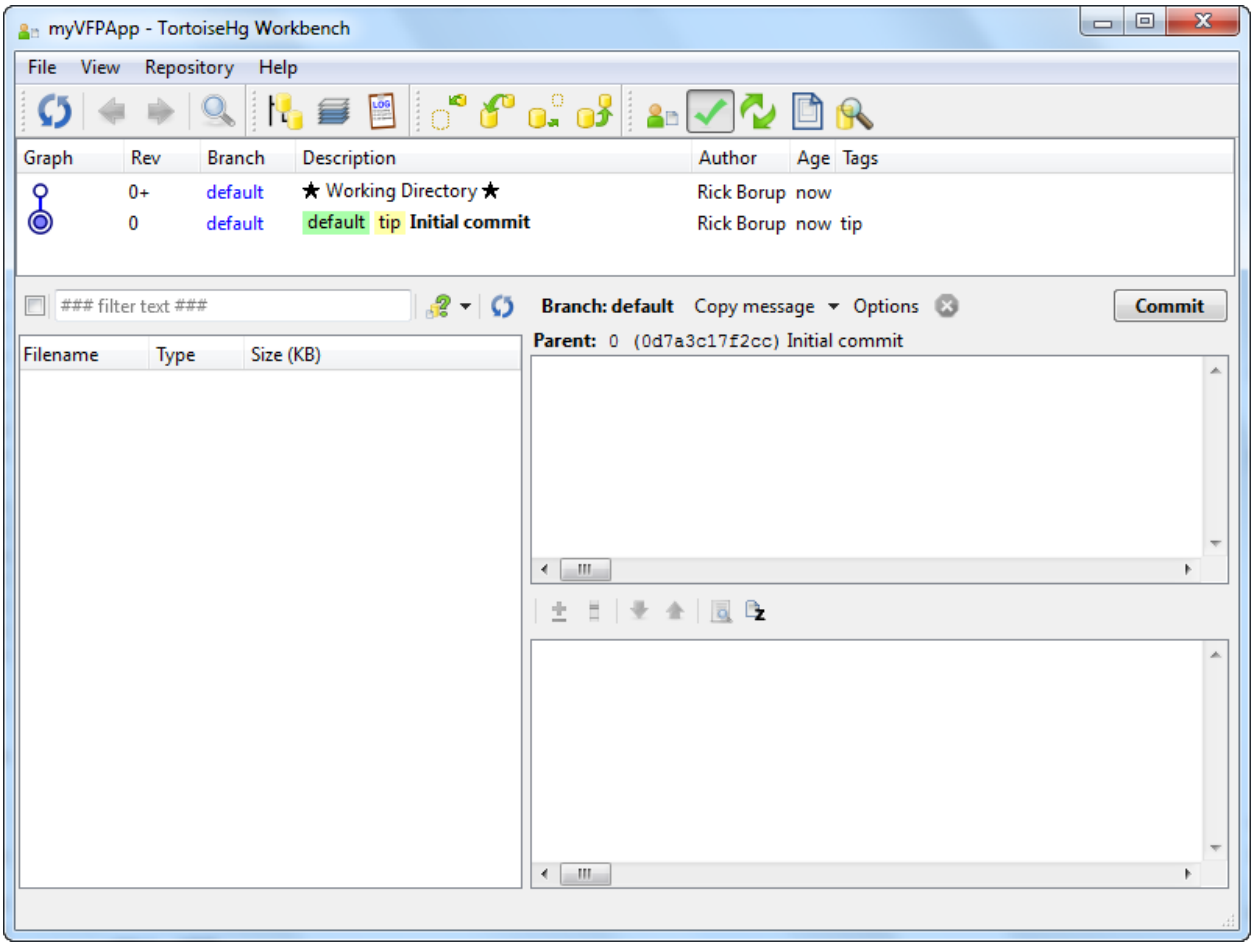


Figure 21: After the initial commit, the TortoiseHg Workbench shows that revision 0 has been recorded in the local repository.

## *Step 5 – Rinse and repeat*

From here on out, all you need to do is to incorporate a commit step into your regular workflow whenever you feel it's appropriate to commit changes to the local repository. For some developers this may be several times a day, for others perhaps less often. You'll also want to add new files as they occur in the development process, but you don't have to do so explicitly. Unless they're excluded by an entry in the .hgignore file, any new files added to your working directory automatically show up in the list displayed by TortoiseHg when you go to do a commit, so you can mark and add them just like you did during the initial commit.

When using Mercurial there is really no downside to doing frequent commits, other than perhaps the continually increasing size of the local repository. Remember, however, that Mercurial stores changesets, so it doesn't create a complete copy of every revised file each time you do a commit. It only stores enough information to derive the difference between the current version of the file and the previous version. For text files the changeset can be quite small, although for binary files even a small change can result in a large delta.

# *Field notes from working with Mercurial*

## Backups

One great benefit of the local repository being a subfolder under the working directory is that when you make a backup of your working directory you automatically get a backup of the local repository.

Even though the use of a version control system such as Mercurial eliminates the need to create and store a series of complete, multi-generational (father – son – grandson) backup copies of your work over time, it's still important to make good backups. The advantage of using Mercurial is that every backup of the project folder includes the local repository. This means you can overwrite an older backup with a newer one without losing the ability to go back to an earlier version of a tracked file.

## Portable repositories

The local repository is file based, and the links it contains are relative to the working directory. This means you can copy or cut and paste projects from one location to another without affecting Mercurial. For example, if you begin working on a project in C:\Test\myProject and later decide you want to move it to F:\Work\myProject, you can do so without breaking anything as far as Mercurial is concerned, as long as both the working directory and the local repository get copied or moved as a unit.

If you work on two machines—for example, one at the office and one at home—you can synchronize your work between the two by copying the working directory and its local repository back and forth. I frequently use Dropbox to do this when I need to take work home in the evening and then bring whatever changes I've made at home back to the office in the morning. This is an admittedly unconventional way of synchronizing things, but it works as long as the changes you make on one machine don't diverge from the changes you make on the other.

The conventional way of sharing changes among machines would of course be to push changes from one machine up to a remote repository and to then pull those changes down to the other machine, or to treat both as remote (each from the other's point of view) and push and pull changes between them.

## Use descriptive messages with every commit

It's a best practice to use a descriptive message with every commit. This is for your own benefit as well as for the benefit of other developers with whom you may be collaborating. You don't have to write an essay, but it's helpful to include more than just a cryptic remark such as "Fixed a bug". A good commit message describes the changes that make this version different than the previous one. For example, a good commit message might be "Changed to say 'Fox rocks!' seven times. Declared local variables and used modified Hungarian notation for variable names."

If you're working from the command line, use the –m option to add a message. If you omit the –m option, Mercurial opens a file in the default text editor where you can enter the message. If you're using TortoiseHg, the Workbench provides a field large enough to enter a good commit message.

When doing a commit from the command line, you can use the –l or --logfile option to read the commit message from an external file. If you use a readme.txt file to document the changes for every release, for example, you could use that file as the source for the commit message. The syntax looks like this:

```
hg commit –l readme.txt
```

I haven't found a way to do this in TortoiseHg yet, but you can always copy and paste your comments from another file into the message field in the TortoiseHg Workbench before committing.

# Tips, tricks, and advanced techniques

## *Getting help*

For help with individual commands, simply type hg help <command> from the command prompt. I use this frequently and have found it to be the easiest way to quickly review what a command does, what parameters it takes, and what options it offers. For example, to get help on the commit command, type

```
C:\Bob\myProject>hg help commit
```

Mercurial responds with:

```
hg commit [OPTION]... [FILE]...
aliases: ci
commit the specified files or all outstanding changes
    Commit changes to the given files into the repository. Unlike a
    centralized SCM, this operation is a local operation. See "hg push" for a
    way to actively distribute your changes.
[more follows here]
```

Mercurial also comes with a set of text-based help files you can explore using any text editor. These files are installed in the C:\TortoiseHg\help folder.

For TortoiseHg, help is available from the Help menu, which links to an HTML Help file in the C:\Program Files\TortoiseHg\doc folder. That folder also contains a pair of PDF files; one is a copy of the book Mercurial: The Definitive Guide and the other is the TortoiseHg documentation. These resources and more can also be accessed from the TortoiseHg pad on the Windows Start menu.


Figure 22: The TortoiseHg pad on the Windows Start menu links to several sources for getting help for both TortoiseHg and Mercurial.

There are also a lot of fine references and tutorials available online. I mention a couple of these in the Resources section of this paper, but there are many, many more. As with pretty much everything else these days, remember GIYF.[10]

## *Fixing Mistakes*

During your development work there will likely be times when you decide, for whatever reason, that you want to abandon a set of changes you've been working on and go back to the way things were. This could happen for example if you start down the road with a certain idea of how to change something and then discover it won't work the way you thought it would, or if you get interrupted half way through a set of changes, lose your train of thought, and just want to start over. Mercurial provides a couple of different ways to help you out in these situations.

### Revert

If you've made some changes to one or more tracked files in your working directory and then decide you want to abandon those changes before you've committed them to the local repository, you can use the revert command to go back to the way things were.

Unless a different revision is specified, the revert command restores the files in the working directory to the state they were in as of the working directory's current parent revision. Normally this will be the version you most recently committed, so a revert basically gives you an easy way to say "just forget it and start over".

The revert command takes a parameter to specify the name(s) of the file(s) to be reverted, or use the --all parameter to revert all changes.

```
C:\Bob\myProject>hg revert --all
```

[10] Google is your friend.

```
reverting fox.prg
```

Files you've modified before reverting are saved in the working directory with a .orig file name extension.

```
C:\Bob\myProject>dir
 Volume in drive C has no label.
 Volume Serial Number is 8869-060D

 Directory of C:\Bob\myProject

09/23/2011  10:37 AM    <DIR>          .
09/23/2011  10:37 AM    <DIR>          ..
09/23/2011  10:37 AM    <DIR>          .hg
09/23/2011  10:37 AM                55 fox.prg
09/23/2011  10:36 AM                57 fox.prg.orig
               2 File(s)            112 bytes
               3 Dir(s)  23,461,097,472 bytes free
```

The revert command can take other options and parameters, which can be explored using hg help as noted above.

```
C:\Bob\myProject>hg help revert
hg revert [OPTION]... [-r REV] [NAME]...
[more help follows here]
```

## Rollback

If you've already committed changes to the repository and only then discover you've made a mistake, or if you've pulled a set of changes you discover you shouldn't have, Mercurial's rollback command can help you out. The rollback command performs a transaction rollback on the local repository in the same sense that a VFP rollback command performs a transaction rollback on a database.

Unlike the revert command, rollback does not alter the contents of the working directory. It simply provides a way to remove the history of an erroneous or undesired action from the local repository. You can only rollback one level.

The rollback command is considered somewhat "dangerous" and is labeled as such in the Mercurial documentation. First of all, there's no way to undo a rollback. Second, a rollback is useless once changes have been pushed to a remote repository. If Bob commits a set of changes to his local repository and then pushes them up to a shared repository, he can still do a rollback on his local repository but it has no effect on the shared repository. Anybody who pulls from the shared repository, including Bob himself, will still get the history Bob rolled back.

## Backout

Mercurial's backout command provides another way of dealing with undesired changes that have already been committed to a repository. The effects of the backout command can become quite complex, resulting in new changesets and sometimes creating the need for additional merges. If you find you need to use the backout command, take time to become familiar with it first.

Chapter 9 of Mercurial: The Definitive Guide is a good source of information for all things relating to finding and fixing mistakes in Mercurial.

## *Previewing actions*

A good way to prevent mistakes from happening in the first place is to preview what's going to happen before actually making it happen. Mercurial provides commands you can use to preview the effect of other commands, while other commands have an option you can use to preview what they will do.

For example, the add command has a –n option you can use to preview the list of files that will be added to the repository. Let's say Bob has intentionally added foo.prg to his project but has also been playing around with a bar.prg, which he doesn't want to save but has forgotten to remove. He uses the –n option to preview what Mercurial would add to his repository before actually adding anything.

```
C:\Bob\myProject>hg add -n
adding bar.prg
adding foo.prg
```

Seeing that Mercurial would add both foo.prg, which he wants, and bar.prg, which he doesn't want, he can delete bar.prg before doing the actual add.

Mercurial provides a pair of commands that let you preview actions involving a remote repository. The outgoing command previews the changes that would be pushed, while the incoming command previews the changes that would be pulled. These commands have several options, but in both cases at least one parameter is required in order to specify the remote repository.

For example, if Bob is ready to push his changes up to a remote repository located at \hgCentral\myProject (relative to his local repository) he could preview what the push command would do by running the outgoing command.

```
C:\Bob\myProject>hg outgoing \hgCentral\myProject
comparing with \hgCentral\myProject
searching for changes
changeset:   0:7792cec862ab
user:        Bob Beta <bob@megasoft.com>
date:        Sat Sep 17 16:45:55 2011 -0500
summary:     Initial commit
```

```
changeset:   1:e8f6bd8efd36
user:        Bob Beta <bob@megasoft.com>
date:        Sun Sep 18 12:57:13 2011 -0500
summary:     changed to 3 times

...

changeset:   7:882b35b6d95f
tag:         tip
parent:      5:1ba4106239cc
parent:      6:927666924516
user:        Bob Beta <bob@megasoft.com>
date:        Mon Sep 19 21:01:46 2011 -0500
summary:     resolved merge conflict, changed to 8 times
```

Because the repository at \hgCentral\myProject is currently empty, Mercurial shows Bob that all of the changesets in his local repository would get pushed to the remote repository, starting at revision 0 and going through the tip at revision 7.

As another example, Carol could use the incoming command to see if there are any changesets in the central repository that she doesn't have yet.

```
C:\Carol\myProject>hg incoming \hgCentral\myProject
comparing with \hgCentral\myProject
searching for changes
no changes found
```

Mercurial shows her that nothing has changed so nothing would be pulled. In this example, this is because nobody has pushed anything to the central repository for myProject yet, so it's still empty. However, this would also be the result if Carol's local repository was already in sync with the central repository.

Leaving the central repository out of the picture for the moment, Carol could also use the incoming command to see if Bob has any changes in his local repository that she doesn't have yet.

```
C:\Carol\myProject>hg incoming ..\..\Bob\myProject
comparing with ..\..\Bob\myProject
searching for changes
changeset:   5:1ba4106239cc
user:        Bob Beta <bob@megasoft.com>
date:        Mon Sep 19 19:56:49 2011 -0500
summary:     changed to 10 times

changeset:   7:882b35b6d95f
tag:         tip
parent:      5:1ba4106239cc
parent:      6:927666924516
user:        Bob Beta <bob@megasoft.com>
date:        Mon Sep 19 21:01:46 2011 -0500
```

```
summary:     resolved merge conflict, changed to 8 times
```

Aha! She sees that at his revision 7, Bob resolved a merge conflict between his version and hers. This is a change she needs but doesn't have yet, so she runs a pull command to add Bob's changes to her local repository, followed by an update to apply those changes to her working directory.

## *More about cloning*

Cloning is the process of making a complete copy of a repository from one location to another. This is a very common way for developers to share their work.

The syntax for cloning a repository is

```
hg clone source [ destination ]
```

where source and destination are URLs.[11] A source URL is required. The destination URL is optional; if not specified, it defaults to the base name of the source.

One example from the Visual FoxPro community is the GoFish4 project currently being worked on by Matt Slay. Matt placed this project under Mercurial version control and uploaded the repository to a publicly accessible location on bitbucket.org. You can clone his project from there.

To do so, first create or choose an empty folder on your local hard drive where you want to project to reside. Then clone Matt's repository into that folder from the public URL on bitbucket.org. The following example illustrates cloning the GoFish4 project into a new, empty folder under C:\Temp.

```
C:\>cd Temp
C:\Temp\>md MattSlay
C:\Temp\>cd MattSlay
C:\Temp\MattSlay>hg clone https://bitbucket.org/mattslay/gofish4
destination directory: gofish4
requesting all changes
adding changesets
adding manifests
adding file changes
added 192 changesets with 2727 changes to 259 files
updating to branch default
128 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

In this example the clone is created in C:\Temp\MattSlay\gofish4 on your machine. This is because Matt's project was uploaded to bitbucket.org from a gofish4 folder somewhere on

[11] Run hg help urls for more information about specifying URLs for Mercurial.

his machine. When you clone the repository, that folder name becomes is the base location for your clone.

Once you have created the clone, you can work with GoFish4 locally on your own machine as if it were your own project. For obvious reasons, Matt does not allow just anybody to push changes back up to his public repository, but you can explore and make changes to your own copy of the project as desired. You can even commit your changes to your own local repository without affecting Matt's public repository.

## *Are you being served?*

Mercurial comes with a small, stand-alone, bare bones Web server you can use to provide remote access to any local repository via http. While not suitable for enterprise level use, this server is useful for instructional purposes and can also be used to provide quick access to a repository that might otherwise be inaccessible to another developer.

The Mercurial server is launched simply by running the serve command from the working directory. By default, it listens on port 8000. Also by default, it is completely open and allows access by anyone who can reach the machine via the URL, although there are ways to restrict access, to implement secure access via https, and more. For additional information, start by running hg help serve and go from there.

As an example, assume that Bob's machine is not accessible to Carol as a mapped drive or shared resource on a local area network. However, Bob wants to temporarily enable Carol to access his repository over the corporate intranet using http. From his working directory for the project, Bob runs the serve command.

```
C:\Bob\myProject>hg serve
listening at http://ABCD1234:8000/ (bound to *:8000)
```

Mercurial responds by letting Bob know his repository is now online at the URL shown, where ABCD1234 is the name of his machine. Assuming Carol can access that URL, she now has access to Bob's repository.

Bob can also take advantage of the built-in Mercurial server to access his own repository from a Web browser via localhost. To check that the server is working, he goes to http://localhost:8000/ in his browser and sees the following page.

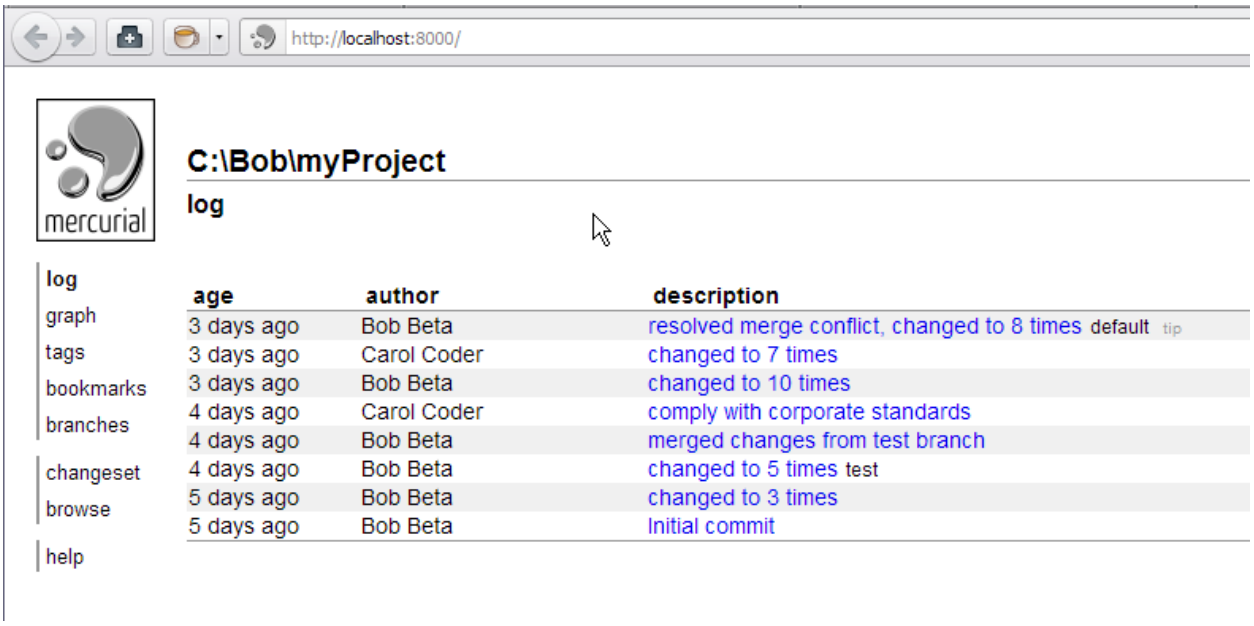


Figure 23: After running hg serve from his working directory, Bob's repository for myProject is available via http on port 8000. In this illustration, Bob is accessing it via localhost on his own machine.

The server initially shows the log for the project. The navigation links on the left support other functions. For example, clicking the browse link opens a page listing the files in the selected revision of the repository, which by default is the tip revision.

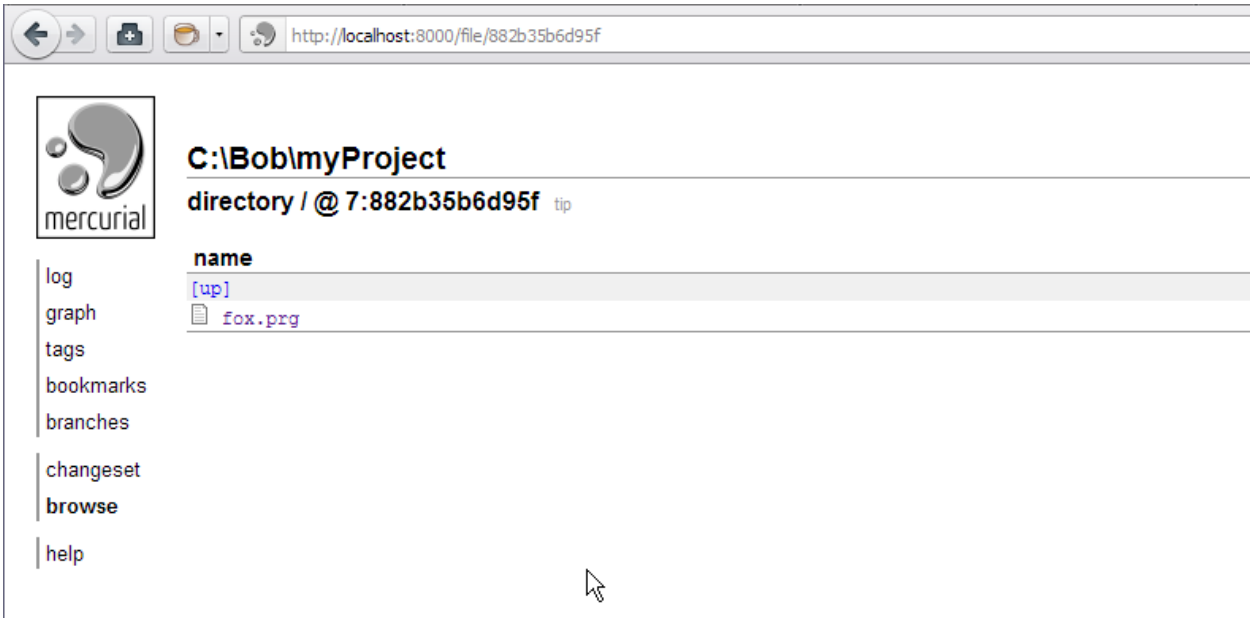


Figure 24: The browse function display a list of the files being tracked in the tip revision. The file name is a clickable link.

The filenames in the list are clickable links, enabling you to view information about the changeset along with the contents of the files.


Figure 25: The Mercurial web server enables you to view details about a changeset along with the actual content of the files in the repository.

Clicking on the diff link, Bob finds that this little Web server can even display the difference between the selected revision and the previous one, using the universal diff format.

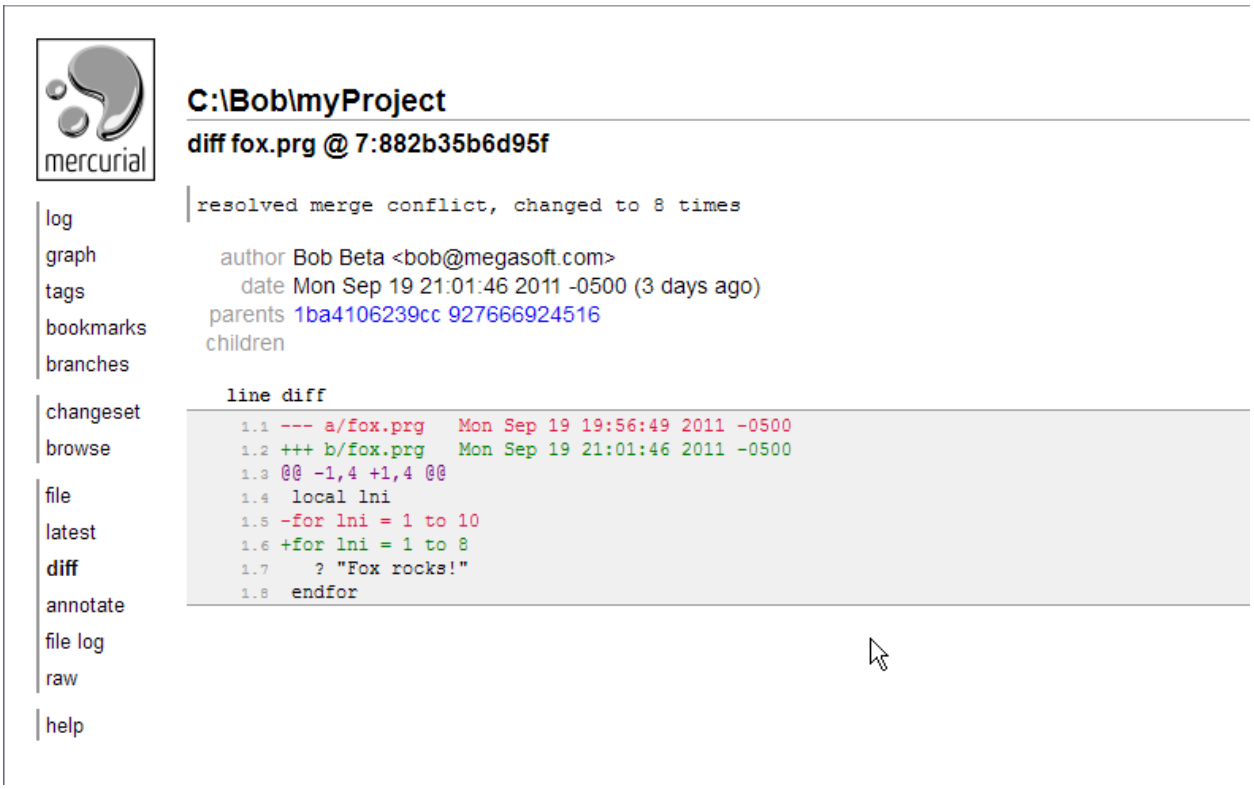


Figure 26: The built-in Web server can even display the diff between the selected revision of a file and the previous revision, using colors to help make it more readable.

If you're interested, you can explore the other functions offered by the Mercurial Web server on your own.

If Bob wanted to, he could enable Carol (or anybody else) to push changes to his repository. To do this, he first has to make a couple of entries in the project's hgrc configuration file. The first is to indicate that secure http (https) is not required, which it otherwise would be by default. The second is allow anybody to push changes without requiring authorization.[12]

Bob creates a [web] section and makes the following two entries in the project's hgrc file. If he was using TortoiseHg, he could also do this via the Settings dialog in TortoiseHg Workbench.

```
[web]
push_ssl = False
allow_push = *
```

If the Mercurial server is running when these changes are made, it needs to be restarted for the changes to take effect. Once that's been done, it's ready to go and Carol can push

[12] This obviously opens a huge security hole, but under controlled conditions you can do so if only for testing and experimentation. Remember to shut down the Mercurial server as soon as you're done using it, though!

changesets from her repository to Bob's. On the flip side, Carol could do the same thing on her machine in order to enable Bob to push changes to her repository over http.

## Hosting solutions

You have a few choices if you're looking for a hosted solution for your Mercurial repositories. The choice is principally between public hosting or private hosting.

The best list of public hosting solutions I've found is at http://mercurial.selenic.com/wiki/MercurialHosting. Of these, probably the most popular is Bitbucket, which offers several levels of plans and pricing. According to their website, "All plans include unlimited public and private repositories". A free plan is available for projects with up to five users. You can sign up for an account at https://bitbucket.org.

CodePlex has supported Mercurial since January 2010 – see the announcement at http://blogs.msdn.com/b/codeplex/archive/2010/01/22/codeplex-now-supporting-native-mercurial.aspx. Sourceforge.net and Google Code also support Mercurial.

The alternative to a public hosting solution is to set up your own server. One source of information on various ways to approach this can found at on the Mercurial wiki at http://mercurial.selenic.com/wiki/PublishingRepositories. Many of the solutions listed there involve using *nix servers and technologies. For Windows developers the question is, can Mercurial be hosted on a Windows machine running IIS? I haven't tried it, but the answer appears to be a qualified "Yes". The best reference I've found is at http://stackoverflow.com/questions/818571/how-to-setup-mercurial-and-hgwebdir-on-iis.

# *Tips*

This section is a somewhat random collection of information and tips about Mercurial commands that I've gathered while learning and working with Mercurial. Some of them come from my own experience, but many come directly from the mercurial.selenic.com/wiki/ website, which has a nice "tip of the day" feature.

Some of these things have been covered earlier in the paper, but others have not. I've grouped them into a couple of different categories for ease of reference, but each one pretty much stands on its own and they're not presented in any particular order.

## Basic commands

hg status – display the status of files in your working directory. Use *hg st* for short.
Use hg status --change n to see the status of changeset n.

hg summary – summarize the state of the working directory

hg log – display the history of changesets in the local repository
Use hg log -k <keyword> to search your history for a keyword
Use hg log --style xml to generate the log output in xml format
Use hg log --style compact to display the log output in a more compact format

Revsets are a powerful query language for log and other commands. See http://www.selenic.com/hg/help/revsets for more information.

See hg help revisions for the many ways to specify revisions.

**Visual FoxPro Tip: Browse the repository history in VFP**

From the command prompt, do hg log --style xml > log.xml to send the log output to a file named log.xml. Then, in VFP, do xmltocursor( "log.xml", "csrHgLog", 512) to create a cursor you can browse and work with in VFP.

hg id – identify the working copy or specified revision
Use hg id -i -r <rev> to find the changeset ID for a given revision or tag

hg tag – apply a name to the current or a specified revision
Note that adding a tag creates a new revision (i.e., a new changeset). You can use tags to search the log. For example, hg log –r "Release version 1.0.1" returns the revision tagged as "Release version 1.0.1".

hg rename – rename a tracked file. Mercurial handles this as a copy and delete.

hg manifest – see a list of the current or a specified revision of the project manifest (i.e., find out which files are under version control)

hg update – update the files in the working directory to a specific version from the local repository. Note that you can update to either an older or a newer version than the one which currently exists in the working directory. Updating to an older version amounts to a going back in time to an earlier state of the files in the working directory.

hg update --clean – discard uncommitted changes in the working directory, with no backup

hg branch – set or show the current branch name

hg branches – list the branches in the repository

hg merge – merge changes from one branch into another

hg bookmark – with no parameters, show the existing bookmarks; with one parameter, set a bookmark with the specified name. In Mercurial 1.8 and later, bookmarks can be pushed and pulled between repositories.

 

hg verify – verify the integrity of a repository

hg cat – output the current revision or a given revision of files

hg parent – show the parent(s) of the working directory or revision

## Working with a remote repository

hg serve – start the Mercurial server from a folder containing a repository[13,14]

hg clone – create a copy of a remote repository on the local machine

hg pull – fetch a set of changesets from a remote repository

hg merge – merge other's changes with the changesets in your local repository

hg push – push a set of changes from your local repository to a remote repository

hg outgoing – preview what hg push would do

hg incoming – preview what hg pull would do

## Mercurial extensions

Several extensions are available to apply custom configurations or behaviors to Mercurial. These are a few of them.

Enable the **color extension** to get colorized output in the command window. See http://mercurial.selenic.com/wiki/ColorExtension for more information.

Enable the **progress extension** to get a progress bar. See http://mercurial.selenic.com/wiki/ProgressExtension for more information.

Enable the **pager extension** to get paged output. See http://mercurial.selenic.com/wiki/PagerExtension for more information.

## FAQs

**Q:** How do I know which files are in my repository? In other words, how do I know which files are being tracked?

[13] If IIS is running, you may need to stop it in order to get to the Mercurial server. This seemed to be true on my Windows 7 machine but was not required on my XP machine.

[14] Once the server is running, you can access it from http://localhost:8000/ in a Web browser. In addition to showing the repository log (history of changes), there are also links to other functions including Help.

**A:** Look at the manifest. From the command line, do hg man. In TortoiseHg, click the manifest button on the toolbar for a convenient tree-view of the contents of the repository.

**Q:** How do I switch between branches?
**A:** Use the update command.
hg update default – change to the default (master) branch
hg update <branchName> – change to branch named <branchName>

**Q:** Which commands can potentially alter the contents of files in my working directory?
**A:** hg update, hg merge, hg revert, and hg rename

**Q:** How do I know if my working directory is current?
**A:** hg status and hg summary

**Q:** How do I know which revision(s) my working directory is based on ?
**A:** hg parents

# Resources

Here are a few sources of information and support to help you continue learning about Mercurial and TortoiseHg.

## *Downloads for Mercurial and TortoiseHg*

TortoiseHg and Mercurial (bundled installer for Windows):
http://tortoisehg.bitbucket.org

Mercurial stand-alone installers, various versions:
http://mercurial.selenic.com. The bundled installer with TortoiseHg is also available from this site.

Mercurial source control plug-in for Visual Studio:
http://visualhg.codeplex.com

## *References and support*

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/

Latest news and useful links:
http://mercurial.selenic.com/wiki

TortoiseHg documentation:
http://tortoisehg.bitbucket.org/manual/2.1

Tutorial by Joel Spolsky:
http://hginit.com

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

## Summary

Mercurial is an easy-to-install and easy-to-use distributed version control system. It's a great choice for Visual FoxPro developers and others working on Windows machines, especially when used in concert with the TortoiseHg shell. Mercurial offers significant benefits not only to teams but also to independent developers working solo.

If you're using a different version control system but are less than fully satisfied with it, Mercurial may be just what you're looking for. Mercurial can import version history from various other systems including Subversion and git, so migration may be easier than you think. If you're not using a version control system at all, what are you waiting for? Mercurial is a great way to get started.