

MERCURIAL

HG YELP



Kenny Ballou
Document Version 0.0.9
September 14, 2012

Contents

1	Introduction	4
1.1	What is Mercurial?	4
1.2	Why?	4
1.2.1	SVN	4
1.2.2	CVS	4
1.2.3	Git	4
1.2.4	HG	5
1.3	About Notation	5
2	Configuration	5
2.1	Mercurial Configuration	5
2.1.1	Configuration Files	5
2.1.2	Username	6
2.1.3	CA Certificates	6
2.1.4	External Tools (Diff/Merge)	6
2.1.5	Proxy Settings	8
2.1.6	Editor	8
2.2	Project Layout/ Configuration	8
2.2.1	HGIgnore	8
2.2.2	README	9
3	Mercurial Basics	9
3.1	Help	9
3.2	Init and Clone	10
3.2.1	Init	10
3.2.2	Clone	10
3.2.3	Status	11
3.3	Modification	12
3.3.1	Add	12
3.3.2	Remove	12
3.3.3	Forget	13
3.3.4	Commit	13
3.3.5	Rename	14
3.4	Sharing Changes	14

3.4.1	Update	14
3.4.2	Pull	15
3.4.3	Push	15
4	Collaboration and Best Practices	16
4.1	Overview	16
4.2	Low Level	16
4.2.1	Commit Comments are REQUIRED	16
4.2.2	Commit and Commit Often	17
4.2.3	Pull Often	17
4.2.4	Pull BEFORE Pushing	17
4.2.5	Prefer Pulling Over Pushing	17
4.2.6	Handle Merges Promptly	18
4.2.7	Use <code>.hgignore</code> appropriately	18
4.2.8	Default or trunk is always stable	19
4.2.9	Fork by convention rather than exception	19
4.2.10	Features and patches live explicitly separate	20
5	Advanced Mercurial	20
5.1	Log	20
5.1.1	Example	20
5.2	Clone	22
5.2.1	Example	22
5.3	Merge and Resolve	23
5.3.1	Example	23
5.4	Update	24
5.5	Diff	25
5.5.1	Working Difference	25
5.5.2	Between Revisions	26
5.5.3	Changes	27
5.6	Incoming and Outgoing	28
5.6.1	Incoming	28
5.6.2	Outgoing	28
5.7	Reverting a Change	29
5.7.1	Example	29
5.8	Rolling Back a Transaction	29

5.8.1	Example	29
5.9	Pushing and Pulling: Advanced Uses	30
5.9.1	Situation	30
6	Converting VCS	31
6.1	Setting it Up	31
6.1.1	CVS	32
7	TortoiseHg – Mercurial Basics	32
7.1	Creating a Repository	32
7.1.1	Steps	32
7.1.2	Example	32
7.2	Cloning a Repository	36
7.2.1	Steps	36
7.2.2	Example	36
7.3	Repository Status	36
7.3.1	Steps	36
7.3.2	Example	36
7.4	Adding Files	36
7.4.1	Steps	40
7.4.2	Example	40
7.5	Removing Files	40
7.5.1	Steps	40
7.5.2	Example	44
7.6	Committing Changes	44
7.6.1	Steps	45
7.6.2	Example	45
7.7	Pulling Changes	45
7.7.1	Steps	50
7.7.2	Example	50
7.8	Pushing Changes	55
7.8.1	Steps	55
7.8.2	Example	55

1 Introduction

I'm not going to introduce revision control or why it's useful. I'm assuming, since you're reading this, that you have a basic understanding of some sort of revision control, you do see the benefits, but you may not be completely sold on distributed revision control. I'll say this again later: I don't intend this document to be an all encompassing introduction to the world of revision control.

1.1 What is Mercurial?

Mercurial is a distributed revision control tool. By definition, this means it can keep track of changes to files and record all the history and progression of a grouping of files. By definition of being distributed, it is highly dependable in the way it handles concurrent development, something that is highly desired when most, if not all development is concurrent.

1.2 Why?

Why use Mercurial? Why not Git or SVN? [HG Book Chapter 1](#)

Well, there are several reasons. It is distributed, meaning all parties have copies of the **entire** repository; it is tremendously scalable; it has immutable commit history; and it is super easy to learn!

1.2.1 SVN

Subversion is a centralized version control system. That is, to use Subversion, you are required to have an SVN server to which your code/tracked files will live. Requiring such a server imposes restrictions against the developer and it is not very scalable. If you are working on a simple document, putting that document into its own project repository under SVN seems like a lot of work for the relative benefit of being able to have versions of the file. Furthermore, lets say your subversion server is hosted internally to your company and is only accessible through your company's internal network. If you wish to commit while on the road, you must VPN into your network just to commit (which may not always be available. . .).

1.2.2 CVS

CVS is very similar to SVN but is, well, older. And because of its age has a lack of a few features, that today, should be standard. Further to the issues pointed out above, it is a scary tool to work with. Some commands do more than you expect them to. This is dangerous. I know I certainly do not like typing commands that can potentially destroy my entire module(what we will later call repositories) that I have been hacking away at. Not to mention merging (between named branches, equivalent to all merging in HG) in CVS isn't exactly the most straight forward process. You have to know a bit about it before you can really, easily pull it off.

1.2.3 Git

“The infomation manager from hell” is not a bad tool. In fact, it is very comparable with Mercurial. Linus wrote Git because none of the SCM's at the time lived up the performance requirement to maintain the Linux kernel (among other reasons); so Git is not only fast, but is also very reliable, not to even mention the real beauty of distributed tools: concurrency. However, it requires manual maintenance of the repository to ensure efficiency. Another downside about Git is that it allows mutable revision history, which means you have some scary power of the past of your project that can boarder on the same level of scary as CVS

commands. Although, some might argue mutable revision history is good. That's a personal choice I leave to you. My argument against it, and this isn't truly my argument, is that your revision control tool should display the revision history of how changes *actually* happened and *not* how they should have.

1.2.4 HG

Mercurial is like Git in that it is distributed, fast, and highly redundant. As mentioned before, it also holds the benefit of having immutable revision history. You can use Mercurial for local one off code katas or for huge projects that are more than several million lines of code and can expect the same performance. It is vastly simpler in terms of learning; so simple, in fact, it will tell you what to do. When it comes to merging, Mercurial is very easy. It's nice to you too; after all, there is nothing like having to deal with a merge and have a bad revision tool to make it worse. And let's face it, "if you're still using a centralized revision control system, you're doing it **WRONG**".

1.3 About Notation

I will be using a specific notation throughout out this guide. Namely, commands will look as follow:

```
hg {command} {required_parameter} [optional_parameter]
```

For now, most commands will be from a Linux environment. Easily portable to Windows, however, `hg` will probably not be in your `PATH` and thus you will have to do something like: `C:\myPath\toHG\hg [command] [options] [arguments]` or you may prefer to put it in your path. (I prefer the latter.) If you use `PowerShell`, `hg` is already in your path and works as expected for all examples.

2 Configuration

Here we will talk a bit about Mercurial configuration options (briefly!) and a little about project layout and configuration.

2.1 Mercurial Configuration

There are a few settings that we need to setup initially. Namely, `username` is an absolute requirement if you wish to commit anything (without having to specify it every time). Linux machines will need to set up certificates. Also, use of merge tools will require some extra configuration. But once this is done, it should no longer require any change.

2.1.1 Configuration Files

We can change either the global settings file (if you have write access to it) or change our personal settings file, or even change settings per repository; just depends on what we wish to accomplish. Regardless of where we want to change the settings, it all has virtually the same effect.

All Users If you want to change the configuration for a specific repository only, there is a file for that: `.hg/hgrc`. That is, the `.hg` folder under the repository you're working in.

Linux Users Typically the global file is at `/etc/mercurial/hgrc`

Your user configuration file will be at (better, this is where Mercurial will look): `~/.hgrc`

Windows Users Windows users have several options: you can use the Unix style file name or the Windows style. For example, your user settings file could be either `.hgrc` or `Mercurial.ini`.

System wide configuration location: `<install-directory>\Mercurial.ini` Mine, however, on my Win 7 64 bit machine was: `C:\Program Files\Mercurial\hgrc.d\Mercurial.rc`

User Settings: `%HOME%\hgrc` or `%HOME%\Mercurial.ini`

2.1.2 Username

I might figure this to be obvious. But if it's not: to set your username, you simply provide a configuration key in the `[ui]` section of, preferably, your personal `hgrc` file.

```
[ui]
username = kballou <kballou@onyx.boisestate.edu>
```

This username can be any username you choose. But if you're using a tool like [Bitbucket](#) or [RhodeCode](#), you will want to use the same username and email used for those services.

2.1.3 CA Certificates

One of the first things that will become a real pain, if you're on Linux, is the certificate errors. We need to tell Mercurial where to look to compare certificates. This is easy, we just need to add a configuration section to our `hgrc` file of choice and be done. Just add the following::

```
[web]
cacerts = /etc/sll/certs/ca-certificates.crt
```

This may be done for you, if you're on Windows or if you're using a system that has already had Mercurial set up. Also, the exact location of the `ca_certificates.crt` file might be different per machine. [Mercurial Wiki: CA Certificates](#)

2.1.4 External Tools (Diff/Merge)

What if we don't like the internal tool for merging? Change it. It's not like Mercurial wasn't built to be extendable.

External Diff [External Diff](#)

```
[extensions]
hgext.extdiff = myDiffTool
```

External Merge Tool [External Merge](#)

```
[ui]
merge = myTool
[merge-tools]
myTool.args $base $local $output $other +close +close
```

Vimdiff

```
[ui]
merge = vimdiff
[merge-tools]
vimdiff.executable = vim
vimdiff.args = -d $base $local $output $other +close +close
```

KDiff3 This has been tested under Linux and Windows.

```
[ui]
merge = kdiff3
[merge-tools]
kdiff3.args $base $local $other -o $output
```

Internal Tool Maybe you prefer conflict characters...:

```
[ui]
merge = internal:merge
```

Add Diff Command There are cases where you do not want to change the `hg diff` but still want to also view the diff with more context. To do so, we can enable an extension and then define our external diff tool with a separate command and all.

First, you will need to enable the external diff extension (`hgext.extdiff`):

```
[extensions]
hgext.extdiff=
```

Next you will define a new section, `extdiff` and define the command and the parameters for the command (the following shows how to setup `gvim` to be your external diff command. Notice the parameters are specific to `gvim`):

```
[extdiff]
cmd.gdiff = gvim
opts.gdiff = -d -f
```

After which, you should now be able to execute `hg gdiff` and a `gvim` window will appear and display the difference between the current file and the last committed version.

For more information, visit the [wiki](#).

2.1.5 Proxy Settings

If you are working behind a proxy, you may run into a few frustrations where you are unable to do any remote actions. (and you're probably on a Windows machine. Linux seems to listen to the `http_proxy` environment variable). To resolve this issue, you can add the `[http_proxy]` section to any one of your `hgrc` files and Mercurial will now properly use the proxy server.

For more information, you may checkout the [wiki](#).

```
[http_proxy]
host={ip|hostname}[:port]
```

2.1.6 Editor

If you would like to change your editor (like you might if using Windows), you may specify the `editor` option in your `./hgrc` file; or if you're using Linux, you may also set the `EDITOR` environment variable.

```
[ui]
editor = {whatever}
```

Special Note for GVim If you, like me, want to set your default editor to GVIM, you will need to also supply the `-f` option for GVIM:

```
[ui]
editor = gvim -f
```

Otherwise, GVIM will immediately release the shell and HG will abort because of empty commit message. The `-f` tells GVIM to remain in the foreground and not release the shell.

2.2 Project Layout/ Configuration

How should I layout my project? Well, to be perfectly honest, that's entirely up to you (or your company's standard). Some build tools require you to have your project setup into specific sections, but Mercurial doesn't care. The point is, you may have a set of common files that you include with every project and sometimes it's a little unclear where they should live depending on the circumstance.

2.2.1 HGIgnore

Probably one of the most included files, ever? This file will help you keep files you don't want creeping into your repository, out. One of the things I like about this file, compared to another dreaded VCS, is that the ignore file can sit in the root of the repository and help Mercurial ignore files anywhere, arbitrarily deep, in the repository. Here, I give you a sample ignore file that you may use as an example or a base for you next repository:

```
syntax:glob

bin
**.class
```

```
** .jar
** .classpath
** .project
** .gbk
** .orig
** .bin
** .tar.gz
```

Virtually anything can be put into this file (files, folders, patterns), one exclusion per line. The first line tells Mercurial how to interpret the file and the following lines are exclusions. I will leave the rest to the wiki. [.hgignore](#)

2.2.2 README

A README file helps others get a quick understanding of the project, or it's your entire documentation. Either way, if you use other tools (like [Bitbucket](#)), you can have your README compiled to HTML using some light weight markup language, like reStructured text or markdown. This file is usually completely repository dependent and is typically included in the root of the repository. If you are using another tool, although this is dependent of the tool, typically your README *must* be in the root of the repository.

Use with SCM Hosting solutions The two tools I have used, [Bitbucket](#) and [Rhodecode](#), both use a fairly consistent convention for determining how to parse and compile the README file. Namely, if you name your README file with the extensions: `.rst`, `.md`, or `.markdown` the tool should choose reStructured text or markdown respectively. Further, [Bitbucket](#) has a neat deal where it decides to treat your README as reStructured text if you pick Python as your language for the repository.

Notice, however, these tools may have particularities about them when it comes to the engine or parsing tool it uses to render your markup.

3 Mercurial Basics

Let's start a basic tour of the commands you will use while using Mercurial. This will serve as a very basic introduction to Mercurial's commands. It is in no way a definitive guide to Mercurial. If that's what you want, go read the *actual* [Definitive Guide](#)

I will be starting with command line usage for now. There are graphical tools for most of the commands provided, but I have no real experience with them so far.¹

3.1 Help

By far the most important `hg` command out there. This command takes as an argument something you want more information about. i.e. `hg help help`

`hg help {command|help topics}`: List the help/ wiki documentation for the command or help topic.

¹I have found TortoiseHg to be, no disrespect, but fairly counter intuitive to what I'm trying to accomplish. Certainly, this may be only an issue of familiarity. However, comparatively: TortoiseHg is far more confusing than the CLI.

3.2 Init and Clone

Before any Mercurial commands are even valid, we need a repository to work against. This is where `init` and `clone` come in.

3.2.1 Init

To get started with a new repository, you will need to tell Mercurial to start a new repository with `init`. `init` creates a new repository in the current directory as a new folder specified when you run the command. (it may even, if setup properly, create remote repositories. But I'm not going to get into that.)

```
hg init [name] "create a new repository in the given directory"
```

Example

```
[kballou@onyx ~]$ mkdir Temp
[kballou@onyx ~]$ cd Temp
[kballou@onyx Temp]$ pwd
/home/kballou/Temp
[kballou@onyx Temp]$ hg init Test
[kballou@onyx Temp]$ ls
Test
[kballou@onyx Temp]$ cd Test
[kballou@onyx Test]$ ls -a
.  ..  .hg
[kballou@onyx Test]$ pwd
/home/kballou/Temp/Test
```

3.2.2 Clone

To begin working on an existing repository that may be either hosted or located locally, you will need to `clone` the repository. `clone` downloads, (or copies, if local) the repository into the current directory.

```
hg clone {URI_TO_REPO_TO_CLONE} [name] "make a copy of an existing repository"
```

You can also name the repository locally with the optional `name` parameter. If you don't want to call your clone of HGYelp to be called `hgyelp`, you can provide a name for it and it will be cloned into a new directory called `[name]`.

Example

```
[kballou@onyx ~] cd Temp
[kballou@onyx Temp]$ pwd
/home/kballou/Temp
[kballou@onyx Temp]$ hg clone Test Test2
destination directory: Test2
requesting all changes
adding changesets
adding manifests
adding file changes
```

```

added 3 changesets with 4 changes to 3 files
updating to branch default
2 files updated, 0 files merged, 0 files removed, 0 files unresolved
[kballou@onyx Temp]$ ls
Test  Test2
[kballou@onyx Temp]$ cd Test2
[kballou@onyx Test2]$ ls
testFile  testFile2

```

3.2.3 Status

If we want to know what has changed to our working directory compared to the last checkin of the repository, we will need to print a list of files of interest. Status will do just that. Further, instead of being required to suppress noisy output, Mercurial will default to only giving you a list of “interesting” files. That is, files that are different (in any way) from the last checkin.

`hg status [file|folder list]` “show changed files in the working directory”

To get a full list of codes, use `hg help status`.

Here is a table that lists common ones:

Code	Description
M	Modified
A	Added
R	Removed
?	Not tracked
!	Missing

Example Ignore some of the commands. We will go over them soon.

```

[kballou@onyx Test]$ pwd
/home/kballou/Temp/Test
[kballou@onyx Test]$ echo "This is a test file" > testFile
[kballou@onyx Test]$ hg status
? testFile
[kballou@onyx Test]$ hg add
adding testFile
[kballou@onyx Test]$ hg status
A testFile
[kballou@onyx Test]$ hg commit -m "added testFile"
[kballou@onyx Test]$ hg status
[kballou@onyx Test]$ echo "This is a new testFile" > testFile
[kballou@onyx Test]$ cat testFile
This is a new testFile
[kballou@onyx Test]$ hg status
M testFile
[kballou@onyx Test]$ hg commit -m "Modified testFile"
[kballou@onyx Test]$ hg remove testFile

```

```
[kballou@onyx Test]$ hg status
R testFile
```

3.3 Modification

3.3.1 Add

To begin tracking changes to a file, we will need to tell Mercurial we are interested in the file's history. For that, we **add** the file. **add** will add, if nothing is specified, all files that are not currently being tracked. Notice, you do not have to worry, **add** does listen to the `.hgignore` file.

`hg add [file|folder name]` add the specified files on the next commit

Example (Explicit Adding)

```
[kballou@onyx Test]$ pwd
/home/kballou/Temp/Test
[kballou@onyx Test]$ ls
[kballou@onyx Test]$ echo "This is a test file" > testFile
[kballou@onyx Test]$ ls
testFile
[kballou@onyx Test]$ hg add testFile
```

Example (Implicit Adding)

```
[kballou@onyx Test]$ pwd
/home/kballou/Temp/Test
[kballou@onyx Test]$ ls
testFile
[kballou@onyx Test]$ echo "This is another test file" > testFile2
[kballou@onyx Test]$ ls
testFile testFile2
[kballou@onyx Test]$ hg add
adding testFile2
[kballou@onyx Test]$
```

Note The first command, `hg add testFile`, should have no output. This is good.

3.3.2 Remove

Projects change, requirements change, everything changes: files will inevitably change, die off, and will need to be cleaned up. We can remove such dead files

`hg remove [files...]` Remove file(s) from the repository.

Example

```
[kballou@onyx Test]$ ls
trash
[kballou@onyx Test]$ hg remove trash
[kballou@onyx Test]$ hg commit -m "took out the trash"
[kballou@onyx Test]$ ls
```

Note The last `ls` displays no files, because there are no longer any files.

3.3.3 Forget

`Forget` is, well, special. It serves a dual purpose, depending on the file you're running it against and its history. Namely, it can remove a file, like `remove`, if a file is already committed. Or, if you mistakenly add a file you didn't actually want to track before your next commit, you can use `forget` to tell Mercurial to not actually track the file.

`Forget` is not more powerful than `remove`. It is simply an opposite action to `add`. Analogously: subtraction to addition. Further, `forget` will not remove the file from the project history nor will it remove the file from the current working directory.

```
hg forget {files...} forget the specified files on the next commit
```

Example

```
[kballou@onyx test]$ dd if=/dev/urandom of=forgetMe count=1024
1024+0 records in
1024+0 records out
524288 bytes (524 kB) copied, 0.039488 s, 13.3 MB/s
[kballou@onyx test]$ hg status
? forgetMe
[kballou@onyx test]$ hg add forgetMe
[kballou@onyx test]$ hg status
A forgetMe
[kballou@onyx test]$ hg forget forgetMe
[kballou@onyx test]$ hg status
? forgetMe
```

3.3.4 Commit

As with any VCS, adding our files, isn't enough. We must also tell Mercurial when to save and increment a file's revision. For that, we have the `commit` command. Commit will save all or the specified files with the provided commit message. If you do not include a commit message, Mercurial will drop you into your `$EDITOR`. That is, the editor defined by the environment variable `$EDITOR`. More research on the topic will be needed for Windows. As of this version, Windows is ignoring the `%EDITOR%` and just launching notepad.

```
hg commit [-m {message}] [files|folders] "commit the specified files or all outstanding changes"
```

I find, if I have a particularly long commit message, not supplying `[-m {message}]` is useful. However, most other cases, I will always just do it inline.

Example

```
[kballou@onyx Test]$ hg commit -m "Initial Commit of testFile"
[kballou@onyx Test]$ ls
testFile
```

Note Like `add` and most other Mercurial (and Linux/Unix for that matter) commands, if successful, has no output.

3.3.5 Rename

The development process is no stranger to the fact that you will inevitably need to rename files, why should your VCS tool be a stranger to it as well? Mercurial can rename and move files. Further, and equally as important, Mercurial has automatic rename detection during merging. That is, Mercurial will know that a change done to an older version of the file will follow to the new file. [Shall we demonstrate this? I think so. Soon.]

`hg rename {old file} {new file}`: “rename files; equivalent of copy + remove”

Example

```
[kballou@onyx Test]$ pwd
/home/kballou/Temp/Test
[kballou@onyx Test]$ echo "This is garbage" > garbage
[kballou@onyx Test]$ hg add garbage
[kballou@onyx Test]$ ls
garbage
[kballou@onyx Test]$ hg commit -m "adding garbage"
[kballou@onyx Test]$ hg rename garbage trash
[kballou@onyx Test]$ ls
trash
[kballou@onyx Test]$ hg commit -m "renamed garbage"
[kballou@onyx Test]$ ls
trash
```

3.4 Sharing Changes

Another obvious fact of development is that you are hardly ever working alone. Even if you’re the only one contributing to a project. For this, we have the `remote`² operations `push` and `pull`.

3.4.1 Update

However, before we can go into `push` and `pull`, we must first, quickly, cover the `update` command.

`Update`, in its basic form, updates the working directory of the current repository. There are advanced uses of it, but we will cover those later. All that really needs to be known here is that you should update your repository after doing a remote operations. We will demonstrate as such in the `pull` example.

²Notice, “remote” in this context is meaning merely beyond a current repository.

3.4.2 Pull

Part of sharing your changes is getting the changes of other developers. Pull provides us this functionality. Pull, as the name might imply, will get the changes from the SOURCE of the repository (in most cases, this is probably *upstream*).

If SOURCE is omitted, Mercurial will look into the repository's `hgrc` file and use the `default` under the `[path]` section.

```
hg pull [SOURCE] "pull changes from the specified source"
```

Something to keep in mind in both cases, if the repositories are not related, Mercurial will not let you pull (or push) the changes in. That is, if the repository you are pulling from does not have the same/ similar parent change set, or is a different repository all together, you may not pull the change in. The wiki specifies a way around this, however, like the Wiki, I do NOT recommend this.)

Example

```
[kballou@onyx hgyelp]$ pwd
/home/kballou/Temp/hgyelp
[kballou@onyx hgyelp]$ hg pull
pulling from ssh://kballou_bitbucket/kballou/hgyelp/
searching for changes
adding changesets
adding manifests
adding file changes
added 3 changesets with 4 changes to 2 files
(run 'hg update' to get a working copy)
[kballou@onyx hgyelp]$ hg update
2 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

3.4.3 Push

Following our sharing analogy, we will want to some how share our changes with everyone else. To accomplish this, Mercurial provides us push. Push will place the missing change sets in DEST. (This is not typically *upstream*.)

If DEST is not specified, Mercurial will use the repository's `hgrc` file to guess.

```
hg push [DEST] "push changes to the specified destination"
```

Example

```
[kballou@onyx hgyelp]$ hg commit -m "Updated Existing Verbs"
[kballou@onyx hgyelp]$ hg push
pushing to ssh://kballou_bitbucket/kballou/hgyelp
searching for changes
remote: adding changesets
remote: adding manifests
remote: adding file changes
remote: added 1 changesets with 2 changes to 2 files
remote: bb/acl: kballou is allowed. accepted payload.
```


4 Collaboration and Best Practices

Before I delve into some more complicated commands and advanced uses, I would first like to talk a little bit about collaborating and some best practices. It's only natural that your development will lead to, at some point or another, a place in which collaboration is a must.

Obviously, some of the suggestions made here will not apply or work for every team or repository/ project and because they are just that, suggestions, you needn't worry yourself about potentially ignoring a few of them. If it doesn't make sense, or your team is not ready for the complexity, ignore it. Adopt it later, when it makes sense.

4.1 Overview

Here is an overview list of the suggestions

- Commit comments are **REQUIRED**
- Commit and commit often
- Pull often
- Pull before pushing
- Prefer pulling over pushing
- Use `.hgignore` appropriately
 - No Compiled Binary Files
- Handle merges promptly
- *default* or *trunk* is always stable
- Fork by convention rather than exception
- Features and patches live explicitly separate

4.2 Low Level

4.2.1 Commit Comments are **REQUIRED**

All commits should require a comment. This is probably one of the only rules that should be completely and utterly firm. I/we/your team doesn't care how small of a change your commit carries, if there's not message or link to your bug tracker to why you made the change, it's not (or at least it shouldn't) be accepted, Period.

Further, if looking for a good set of guidelines around your commit messages, I recommend reading the commit message guidelines set forward by a few open-source projects. Namely, the Linux Kerenl and Git. If you want just a simple summary, you may like "[A Note About Git Commit Messages](#)".

4.2.2 Commit and Commit Often

It should be understood that all team members commit often. It's better to commit often smaller changes or logical groups of changes together instead of large commits. This reduces the size of commit messages and reduces headache when merging. For example, if you're working on an issue, your commit should only include changes to the code base that are related to that issue. Don't drop your work and pick up somewhere else and make some other unrelated change. Ideally, you're using an issue tracker to help you decided logical groupings for code changes.

4.2.3 Pull Often

Similarly to Committing Often, it is desired that you integrate fairly often. Maybe not as often as you commit; but certainly more often than the duration of a sprint. This will help reduce the amount of branching occurs during development, and reduce pain caused by merging.

I haven't tried this yet, but it might be good to pull as often as you finish features or bug fixes. The end of a feature sounds like a good spot to merge and have code come together. Also a good point to make sure previous change sets don't regress your fix, or break your feature (or theirs!)

Notice, this merging is done by you, not your team; you have yet to push your change into the group fork; where, ideally, you have an integration service testing your code.

4.2.4 Pull BEFORE Pushing

Do you like clobbering other people's work? The `-f` option is for you! No seriously, pull from the repository you are about to push to. If you're seeing this, it's because either you didn't pull, or less likely because someone snuck a push in there before you (after you pulled and got the go ahead):

```
[kballou@onyx Yelp]$ hg push
pushing to ssh://kballou_bitbucket/kballou/hgyelp
searching for changes
abort: push creates new remote head 9cce737c0dbe!
(you should pull and merge or use push -f to force)
```

Mercurial tells you what you should do, listen to it. (Just don't use `-f`, please.)

4.2.5 Prefer Pulling Over Pushing

There is something to be said about Git and it not allowing you to push into local repositories by default. Allowing such a push creates internal inconsistencies with the repository receiving changes. Mercurial might be better prepared for these inconsistencies; however, it is never bad to avoid the potential of destroying a perfectly good repository. Even if you don't destroy it, you may create some *interesting* merge situations later because of it.

What About Remote Repositories? I can't avoid pushing to those Yes, there are a few situations in which you are left with nothing else but to push. This is usually much safer because the repository you are pushing into is not "checked out", so to speak. That is, the remote repository is usually the same as your repository after running `hg update null`; it's not a working directory. Similarly, there is no one on the other end trying to commit changes to this remote repository; whereas a local one is not guaranteed to be in any sort of state and may be a working directory for something else.

To further illustrate, imagine working with a co-developer and pushing your changes into his/ her repository. Imagine the confusion generated by having some *mysterious* changes appear... (Not to mention the frustration...)

4.2.6 Handle Merges Promptly

Sometimes, you're going to have to deal with a merge, and sometimes they aren't fun. But don't let that be a discouragement from doing this because you would rather not deal with it. I guarantee it's infinitely better to do while the source tree hasn't branched much than it is when you have even just two developers doing the merge toward the "end" of the project time line.

Further, when merging, do not make other changes unrelated to the merge. You do not want to mix up an actual new change while you're merging two pieces of code together. It's a very easy way for some code to have features and bug fixes vanish. When merging, or better, before merging, you should check the status of the your working copy to see if you're in an ideal state (no interesting files) to do a merge. If you're not, don't merge; Mercurial is okay with this, you should be too.

4.2.7 Use .hignore appropriately

Files that should not be checked in, should be ignored as to not have them sneak into your repository. Great candidates for ignoring are binary files and developer specific settings or files that are specific to a particular machine.

No Compiled Binary Files Your project can be large, why would you want to make it bigger with generated code that is just going to be recompiled on the next developers' machine? I want to say, to make my point very clear, "no binary files, **EVER!**" But that makes some project resources difficult. Especially for projects of the web application variety. So I will just leave it at: "No binary files, if you can avoid them."

But why should you avoid them? What makes them so bad? Does it really matter, your build tool is going to compile over them anyways, right? ... Right?!

Well, let's think about this for a second further. If you include a compiled binary file with your repository. You're not just including the binary file, but also its history. Now think, when you recompile said binary file, its `diff` is not just one line or two lines, usually the whole file is different now. Even with smart differentials, you are still left with a gigantic differential. Now think of this binary file 10 commits from now, 20, 30, 100, 200, or 1000 commits later. This **one** binary file, is not just costing your the size of the binary, but the size of each and every differential from the first commit to today's. Let's come back to this thought in just a second.

What's nice about not including them is you can ensure that your code can compile correctly with just source. There's no (at least less) "it works on my machine". One of the ideas behind build automation (another topic entirely) is that compiling and testing code is repeatable and predictable. If you include compiled binaries with your project, what sort of check do you have that ensures the build is working (or not working) because the compiled binary isn't actually being recompiled? By forcing them to be compiled you are actually saving yourself a lot of headache.

Further, even with Mercurial's smart approach to compression of files and what have you, a small project's repository can be about a megabyte. How big would a repository be with two million lines of code be? The Linux Kernel's source tree is about 529 MB (1.0 GB when fully checked out). Now remember that thought I said we would come back to: do you see now why this would be horrific?

Exceptions to the case There are a few cases where it is appropriate to include binary files. Images, library dependencies, etc. These are binaries that are not compiled but are required for the project to compile from scratch. Without them, (well except images...) the project would not compile.

Further, if you are wanting to take advantage of binary features of VB6, you will need an existing binary to compile with. That is, you need the binary interface to compile with. Without it, you are not able to use the binary features. (Or so I think...) Further, this is okay because, if you ever change the interface (and in practice this is usually less often) you will certainly need (or may want to) track such a difference. (Granted it's pointless to have a `diff` of such a file...)

4.2.8 Default or trunk is always stable

When ever someone checks out or clones the trunk of the repository, it should be guaranteed to be stable, working, and passing tests. As such, this is an always safe branch/ forking location in the project tree. Code that would otherwise break this rule, shall not be allowed to be pushed.

This is certainly a conventional suggestion that would have to be enforced by the repository owner.

4.2.9 Fork by convention rather than exception

As to have development agree with the first cannon, all code base changes, feature, patch, or otherwise will be forked from the trunk and will be developed independent of the authoritative repository. For really small projects (and arguably for small teams as well), this doesn't seem to have any benefit at all; although, in large projects that have bugs reported frequently and new features introduced semi-frequently, it's easy to see the benefit of being able to isolate all changes to the code base until the changes are ready for integration and release.

However, it also does benefit even small projects. Namely, the delayed integration serves to ease project release timing significantly. For example, like bigger projects, if there are a number of features and/ or bug fixes to be implemented, having each and everyone of those changes developed separate of each other dwarfs the pain of integration and release versus developing them all together.

Even in the simplest of work flows, Mercurial is inherently creating a fork for you when you clone, so even if your work flow still closely resembles that of a centralized VCS, you're already doing what is for your benefit. (Unless, you write some hook for commit to automatically push the change... I've lost hope for you...)

Avoid Named Branching Here is where I would say Git has a substantial advantage over Mercurial: branching. Git's branching system is far superior to that of Mercurial's. Git's branching system allows you to create and jump around your branches instantaneously, delete them when you're finished and only push a certain branch while leaving the rest of them behind. So a developer could employ branching in many ways without a whole lot of restrictions. However, You can get and utilize all of this power by forking instead of using named branches.

The reason Git has a substantial gain when it comes to branching is because Mercurial's branches behave much more closely to that of a centralized system. Meaning that, if you create a named branch, everyone will see that branch the next time your changes are shared, which in my opinion is not only counter to the distributed model but bad for all contributor's understanding of the project.

Counter Argument It might make a lot of sense to start utilizing branches when your source tree grows beyond several hundred MB. It might become unfeasible to do even local cloning; not to mention remote forks and pulling each of those... At which point, I would probably suggest your project move to Git. Linus is very serious about performance. (Why do you think he didn't choose Mercurial for the Linux Kernel?)

Remote Forks versus Local Clones Although this suggestion may be good, it certainly leaves you in the dark about where you should or at what level code should be forked. Let's fix that. (or attempt to...)

Remote Forks When trying to decide where to fork, you should ask yourself a few questions. Namely, is the bug or feature, now referred to as change, big? Risky? Prone to be dropped, abandoned, or otherwise orphaned? Will *you* be, if dropped or abandoned, the person who finishes the change if it is later adopted? If you have answered "yes" to any of those, I recommend that the change be done with a remote fork in mind. If all questions were answered "no", you may safely do a local clone instead.

Local Clones Now let's consider these questions in their opposite forms. Does the change present low risk, is minor in code change, is less prone to being orphaned? If all questions are answered with a "yes" then you may just use a local clone. Otherwise, you may rather use a remote fork.

Special Cases There may be a few special cases to consider. If you're using [Bitbucket](#) and do not own or have write access to a repository you wish to change you will be forced to using a remote fork for all changes, regardless. But these suggestions may be used recursively.

4.2.10 Features and patches live explicitly separate

Following the branch by convention, all changes shall be in there own fork. Okay, maybe a logical grouping can be used. I won't say this is a hard rule but rather a way to make using something like `bisect`, `blame`(sorry, I don't have the usage for these yet), and general development easier. You have some discretion has to how separated you want your development to be, though. [I think I can make this evident with a demo?]

5 Advanced Mercurial

Following is some more advanced Mercurial commands. These will help you in controlling your source code and better manage your repository with team members. It's one thing to work by yourself on your code. You typically don't have worry about other people clobbering your code or ripping out your changes or other crazy things that other developers can do.

5.1 Log

This command will list all of the `changesets` that are in your local copy of the repository.

`hg log [options] [files]` Print the revision history of the specified files or the entire project. There's quite a bit of options, so I will leave those to your own curiosity and a `hg help log`.

5.1.1 Example

```
#We need to artificially inflate the history...
```

```
[kballou@onyx Yelp]$ mkdir temp
[kballou@onyx Yelp]$ cd temp
```

```

[kballou@onyx temp]$ hg init test
[kballou@onyx temp]$ cd test
[kballou@onyx test]$ echo "test file1" > testFile1
[kballou@onyx test]$ hg add testFile1
[kballou@onyx test]$ hg commit -m "Added testFile1"
[kballou@onyx test]$ echo "test file 2" > testFile2
[kballou@onyx test]$ hg add testFile2
[kballou@onyx test]$ hg commit -m "Added testFile2"
[kballou@onyx test]$ echo "test file 3" > testFile3
[kballou@onyx test]$ hg add testFile3
[kballou@onyx test]$ hg commit -m "Added testFile3"
[kballou@onyx test]$ hg log
changeset:  2:2ff4bf12a254
tag:        tip
user:       kballou <kballou@onyx.boisestate.edu>
date:       Tue Apr 03 00:07:57 2012 -0600
summary:    Added testFile3

changeset:  1:01a7ba2472eb
user:       kballou <kballou@onyx.boisestate.edu>
date:       Tue Apr 03 00:07:29 2012 -0600
summary:    Added testFile2

changeset:  0:42d193d34445
user:       kballou <kballou@onyx.boisestate.edu>
date:       Tue Apr 03 00:07:08 2012 -0600
summary:    Added testFile1

```

Useful Options If you would like to know a little bit more or display all of a commit message (instead of only the first line) you can supply the verbose option: `-v`

Example

```

#Continuing from the previous example:
[kballou@onyx test]$ hg log -v
changeset:  2:0d409f2fecf8
tag:        tip
user:       kballou <kballou@onyx.boisestate.edu>
date:       Mon Apr 23 21:06:10 2012 -0600
files:      testFile3
description:
Added testFile3

changeset:  1:56c362588188
user:       kballou <kballou@onyx.boisestate.edu>
date:       Mon Apr 23 21:05:02 2012 -0600
files:      testFile2
description:
Added testFile2

```

```
changeset: 0:761646b81712
user:      kballou <kballou@onyx.boisestate.edu>
date:      Mon Apr 23 21:04:39 2012 -0600
files:     testFile1
description:
Added testFile1
```

5.2 Clone

Now that we have used `clone` to clone repositories. Let's use it for something a little more complicated: check out specific revisions. This is easy.

```
hg clone [-r {rev}] {URI} [name]. Clone specific revision, {rev}, from {URI} and put it in a folder [name]
```

Notice, the revision you specify for the clone will be the `tip` for the clone. That is, it will not pull any changes that are rooted by `-r {rev}`.

5.2.1 Example

```
[kballou@onyx temp]$ pwd
/home/kballou/Dev/Yelp/temp
[kballou@onyx temp]$ ls
test
[kballou@onyx temp]$ hg clone -r 2dfa1733e551 test test2
adding changesets
adding manifests
adding file changes
added 3 changesets with 3 changes to 3 files
updating to branch default
3 files updated, 0 files merged, 0 files removed, 0 files unresolved
[kballou@onyx temp]$ cd test2
[kballou@onyx test2]$ hg log
changeset: 2:2dfa1733e551
tag:       tip
user:      kballou <kballou@onyx.boisestate.edu>
date:      Tue Apr 03 00:35:51 2012 -0600
summary:   Added testFile3

changeset: 1:5d9b355567d3
user:      kballou <kballou@onyx.boisestate.edu>
date:      Tue Apr 03 00:35:24 2012 -0600
summary:   Added testFile2

changeset: 0:f4d21bc583d1
user:      kballou <kballou@onyx.boisestate.edu>
date:      Tue Apr 03 00:34:57 2012 -0600
summary:   Added testFile1
```

5.3 Merge and Resolve

There are several situations when we will want to merge: Merge changes between developers working on the same file, more specifically the same lines of the same file. Or if we want to merge two different revisions, like we will do later.

`hg merge [options]` Merge working directory with another revision.

Also, because we are going over merging, we might also run into conflicts; conflicts that might not resolve so nicely.

5.3.1 Example

```
[kballou@onyx temp]$ ls
test test2
[kballou@onyx temp]$ cd test
[kballou@onyx test]$ ls
testFile1 testFile2 testFile3 testFile4
[kballou@onyx test]$ echo "new testFile1" > testFile1
[kballou@onyx test]$ hg commit -m "Update testFile1"
[kballou@onyx test]$ cd ../test2
[kballou@onyx test2]$ echo "test2:testFile1" > testFile1
[kballou@onyx test2]$ hg commit -m "test2:updated testFile1"
[kballou@onyx test2]$ hg pull
pulling from /home/kballou/Dev/Yelp/temp/test
searching for changes
adding changesets
adding manifests
adding file changes
added 3 changesets with 2 changes to 2 files (+1 heads)
(run 'hg heads' to see heads, 'hg merge' to merge)
[kballou@onyx test2]$ hg heads
changeset: 6:175db4128bd2
tag:       tip
user:      kballou <kballou@onyx.boisestate.edu>
date:      Tue Apr 03 01:18:47 2012 -0600
summary:   Update testFile1

changeset: 3:800c45a81aa4
user:      kballou <kballou@onyx.boisestate.edu>
date:      Tue Apr 03 01:20:24 2012 -0600
summary:   test2:updated testFile1

[kballou@onyx test2]$ hg merge
merging testFile1
warning: conflicts during merge.
merging testFile1 incomplete! (edit conflicts, then use 'hg resolve --mark')
1 files updated, 0 files merged, 0 files removed, 1 files unresolved
use 'hg resolve' to retry unresolved file merges or 'hg update -C .' to
abandon
[kballou@onyx test2]$ cat testFile1
<<<<<< local
```



```

test2:testFile1
=====
new testFile1
>>>>>> other
[kballou@onyx test2]$ echo "merged:testFile1" > testFile1
[kballou@onyx test2]$ hg resolve --mark
[kballou@onyx test2]$ hg commit -m "Merged Changes"
[kballou@onyx test2]$ hg status
? testFile1.orig

```

Notes Notice how Mercurial tells you what to do; don't worry, it's not trying to trick you.

5.4 Update

Straight from the `hg help update`: update working directory (or switch revisions)

We use this command after pulling or if we want to locally switch revisions.

For this section, we care more about the latter use. Let's say we have been slinging a bunch of code down, committing along the way. Then we notice we made a mistake not in the last commit, but several commits ago. We have several options: we could clone into another directory, specifying the revision we want to fix (then we could fix the issue, push up). Or we could update to a specific revision, fix and other wise move along. (Personally, I might actually prefer the former. But the latter works just as well.)

Example

```

[kballou@onyx test]$ pwd
/home/kballou/Dev/Yelp/temp/test
[kballou@onyx test]$ hg log
...
[Snipped for brevity]
...
[kballou@onyx test]$ hg update -r 1
0 files updated, 0 files merged, 1 files removed, 0 files unresolved
[kballou@onyx test]$ ls
testFile1 testFile2
[kballou@onyx test]$ echo "test file 4" > testFile4
[kballou@onyx test]$ hg add testFile4
[kballou@onyx test]$ hg commit -m "Added testFile4"
created new head
[kballou@onyx test]$ ls
testFile1 testFile2 testFile4
[kballou@onyx test]$ hg log
changeset: 3:059719334da1
tag:       tip
parent:    1:01a7ba2472eb
user:     kballou <kballou@onyx.boisestate.edu>
date:     Tue Apr 03 00:26:55 2012 -0600
summary:  Added testFile4

```

```
changeset: 2:2ff4bf12a254
user:      kballou <kballou@onyx.boisestate.edu>
date:      Tue Apr 03 00:07:57 2012 -0600
summary:   Added testFile3
```

```
changeset: 1:01a7ba2472eb
user:      kballou <kballou@onyx.boisestate.edu>
date:      Tue Apr 03 00:07:29 2012 -0600
summary:   Added testFile2
```

```
changeset: 0:42d193d34445
user:      kballou <kballou@onyx.boisestate.edu>
date:      Tue Apr 03 00:07:08 2012 -0600
summary:   Added testFile1
```

```
[kballou@onyx test]$ hg merge -r 2
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
(branch merge, don't forget to commit)
[kballou@onyx test]$ hg commit -m "Merged changes"
[kballou@onyx test]$ ls
testFile1 testFile2 testFile3 testFile4
```

Notes I used the revision number not the hash. But, I strongly recommend you use the hash revision tag; especially if you're working with more than, say, just yourself.

5.5 Diff

There are a few different ways to get diff's of your work. I will cover a few cases here: difference between default/ tip and modified work, difference between working directory and revision or change between revision and next revision.

```
hg diff [options] [files...] Diff repository (or selected files)
```

5.5.1 Working Difference

We checked out some code; we made some changes; then we forget what we changed! Oh No! Well we can get a differential of all the changed files or specify one or more files.

```
hg diff [files...]
```

Example

```
[kballou@onyx Test]$ pwd
/home/kballou/Dev/Yelp/Temp/Test
[kballou@onyx Test]$ echo "Change 1" > testFile1
[kballou@onyx Test]$ hg add testFile1
[kballou@onyx Test]$ hg commit -m "Start Temp Project"
[kballou@onyx Test]$ echo "Change 2" > testFile1
[kballou@onyx Test]$ hg diff
diff -r 7a584cce4319 testFile1
```

```

--- a/testFile1 Mon Apr 09 14:11:55 2012 -0600
+++ b/testFile1 Mon Apr 09 14:12:08 2012 -0600
@@ -1,1 +1,1 @@
-Change 1
+Change 2
[kballou@onyx Test]$ echo "This is another file" > testFile2
[kballou@onyx Test]$ hg add testFile2
[kballou@onyx Test]$ hg diff
diff -r 7a584cce4319 testFile1
--- a/testFile1 Mon Apr 09 14:11:55 2012 -0600
+++ b/testFile1 Mon Apr 09 14:13:18 2012 -0600
@@ -1,1 +1,1 @@
-Change 1
+Change 2
diff -r 7a584cce4319 testFile2
--- /dev/null Thu Jan 01 00:00:00 1970 +0000
+++ b/testFile2 Mon Apr 09 14:13:18 2012 -0600
@@ -0,0 +1,1 @@
+This is another file: Change 1
[kballou@onyx Test]$ hg diff testFile1
diff -r 7a584cce4319 testFile1
--- a/testFile1 Mon Apr 09 14:11:55 2012 -0600
+++ b/testFile1 Mon Apr 09 14:16:05 2012 -0600
@@ -1,1 +1,1 @@
-Change 1
+Change 2

```

5.5.2 Between Revisions

Well the previous example is good if we have made changes. What about when we want to know the difference between some past revision and the current revision or range between two different revisions (not necessarily the default/ tip).

```
hg diff [-r {rev}]
```

```
hg diff [-r {rev}] [-r {other_rev}]
```

Notice: we can specify the `-r` option multiple times.

Example

```

[kballou@onyx Test]$ pwd
/home/kballou/Dev/Yelp/Temp/Test
[kballou@onyx Test]$ hg diff -r 0
diff -r 7a584cce4319 testFile1
--- a/testFile1 Mon Apr 09 14:11:55 2012 -0600
+++ b/testFile1 Mon Apr 09 14:32:29 2012 -0600
@@ -1,1 +1,1 @@
-Change 1
+Change 2
diff -r 7a584cce4319 testFile2
--- /dev/null Thu Jan 01 00:00:00 1970 +0000

```

```

+++ b/testFile2 Mon Apr 09 14:32:29 2012 -0600
@@ -0,0 +1,1 @@
+This is another file: Change 1
diff -r 7a584cce4319 testFile3
--- /dev/null Thu Jan 01 00:00:00 1970 +0000
+++ b/testFile3 Mon Apr 09 14:32:29 2012 -0600
@@ -0,0 +1,1 @@
+Another new File
[kballou@onyx Test]$ hg diff -r 0 -r 1
diff -r 7a584cce4319 -r af56a4a59193 testFile1
--- a/testFile1 Mon Apr 09 14:11:55 2012 -0600
+++ b/testFile1 Mon Apr 09 14:29:41 2012 -0600
@@ -1,1 +1,1 @@
-Change 1
+Change 2
diff -r 7a584cce4319 -r af56a4a59193 testFile2
--- /dev/null Thu Jan 01 00:00:00 1970 +0000
+++ b/testFile2 Mon Apr 09 14:29:41 2012 -0600
@@ -0,0 +1,1 @@
+This is another file: Change 1

```

5.5.3 Changes

This is easy enough, let's say you want to know the change between a specified revision and the next revision immediately after it.

```
hg diff [-c {rev}]
```

Example

```

[kballou@onyx Test]$ pwd
/home/kballou/Dev/Yelp/Temp/Test
[kballou@onyx Test]$ hg diff -c 2
diff -r af56a4a59193 -r 710b88f0e9eb testFile3
--- /dev/null Thu Jan 01 00:00:00 1970 +0000
+++ b/testFile3 Mon Apr 09 14:30:07 2012 -0600
@@ -0,0 +1,1 @@
+Another new File
[kballou@onyx Test]$ hg diff -c 1
diff -r 7a584cce4319 -r af56a4a59193 testFile1
--- a/testFile1 Mon Apr 09 14:11:55 2012 -0600
+++ b/testFile1 Mon Apr 09 14:29:41 2012 -0600
@@ -1,1 +1,1 @@
-Change 1
+Change 2
diff -r 7a584cce4319 -r af56a4a59193 testFile2
--- /dev/null Thu Jan 01 00:00:00 1970 +0000
+++ b/testFile2 Mon Apr 09 14:29:41 2012 -0600
@@ -0,0 +1,1 @@
+This is another file: Change 1

```

5.6 Incoming and Outgoing

While using a hosted repository, (or any cloned repository for that matter) we may want to know what is going to change if we were to do a push or pull.

5.6.1 Incoming

Incoming will list the changes that are going to be brought into the repository if we were to do a pull.

`hg incoming` show new changesets found in source.

Notice: typically “source” means whatever `default` is in the repository’s `.hgrc` file. You may provide a named alternative if you would like. (See Advanced uses of Pushing and Pulling.)

Example

```
[kballou@onyx HgYelp]$ pwd
/home/kballou/Dev/HgYelp
[kballou@onyx HgYelp]$ hg incoming
comparing with ssh://kballou_bitbucket/kballou/hgyelp
searching for changes
changeset: 59:8e2d5e37fee9
tag:      tip
user:     kballou <kballou@onyx.boisestate.edu>
date:     Mon Apr 09 22:11:46 2012 -0600
summary:  incoming and outgoing
```

```
[kballou@onyx HgYelp]$
```

5.6.2 Outgoing

Outgoing will list the changes we will add to the remote repository.

`hg outgoing` show changes not found in the destination.

Notice: Like incoming, you may also provide a named alternative to the default (`default-push` in this case) destination.

Example

```
[kballou@onyx Yelp]$ pwd
/home/kballou/Dev/Yelp
[kballou@onyx Yelp]$ hg outgoing
comparing with ssh://kballou_bitbucket/kballou/hgyelp
searching for changes
changeset: 59:8e2d5e37fee9
tag:      tip
user:     kballou <kballou@onyx.boisestate.edu>
date:     Mon Apr 09 22:11:46 2012 -0600
summary:  incoming and outgoing
```

```
[kballou@onyx Yelp]$
```

5.7 Reverting a Change

In the middle of making changes, we discover we do not want to keep our current work (for whatever reason). We can revert our working copy to the previous commit. This will create a `[file].orig` so we don't have to worry about the chance of us actually wanting that work...

```
hg revert [option] [-r {rev}] {files...} restore files to their checkout state
```

You may also be interested in the `--all` flag, if you're crazy... This flag does exactly what you think it would do: revert **all** the files.

5.7.1 Example

```
[kballou@onyx test]$ pwd
/home/kballou/Dev/Yelp/temp/test
[kballou@onyx test]$ echo "change 1" > test
[kballou@onyx test]$ hg add
adding test
[kballou@onyx test]$ hg commit -m "Added test"
[kballou@onyx test]$ echo "another change" >> test
[kballou@onyx test]$ hg revert test
[kballou@onyx test]$ ls -a
. .. .hg test test.orig
[kballou@onyx test]$ cat test
change 1
[kballou@onyx test]$ cat test.orig
change 1
another change
[kballou@onyx test]$ hg status
? test.orig
```

5.8 Rolling Back a Transaction

In the case of committing a change, or some other transactional action to our repository, `revert` will not help us (Well, not in the manner of `.`. We need something a little bit more aggressive: `rollback`. With its aggressiveness, it should be noted and understood that rolling back is not necessarily safe nor is it guaranteed to succeed. You have been warned.

Something else to keep in mind when using something like `rollback`: if the change you are wanting to `rollback` is beyond only your local copy: there is nothing you can really do to guarantee it goes away and stays away. This is inherent with the distributed nature of Mercurial. Hardly a blemish in my book though.

```
hg rollback roll back the last transaction (dangerous)
```

5.8.1 Example

```
[kballou@onyx test]$ mv test.orig test
[kballou@onyx test]$ hg status
M test
[kballou@onyx test]$ hg commit -m "Changed test"
[kballou@onyx test]$ hg rollback
```

```

repository tip rolled back to revision 0 (undo commit)
working directory now based on revision 0
[kballou@onyx test]$ hg status
M test
[kballou@onyx test]$ cat test
change 1
another change

```

5.9 Pushing and Pulling: Advanced Uses

There are some advanced uses for pushing and pulling I would like to talk about and as the name might imply, it's a bit more than I would like to put into the basics section.

When faced with complicated repository forking, we may start to lose track of which repository is which, what URI to use and other what have yous.

We can add extra `paths` to our repository's `hgrc` file to give us a short cut to our commonly used URIs.

5.9.1 Situation

Let's further explain the situation:

You have only read access to the authoritative repository. You can fork this repository. You are probably not the only person working on this project and/ or you have to continuously change which fork of the project you are working on. (i.e. you're working on a new feature whilst bug fixes must be handled more promptly. Or any mix of that with more people. . .) As such, you want to easily pull from the stable to merge other work into your current work , you will want to be able to push up to your fork and NOT try to push against the master repository, you may even want to quickly be able to push/ pull to a shared in-between repository that your and your colleagues are tearing away at. In all cases you don't want to constantly type out the same or similar (prone to mistakes with potentially devastating consequences) URI's for each of the different repositories. Fortunate for you, there's a way to alias paths with `hgrc paths`.

Example 1 For example, you want to help me write this guide. You fork the repository and clone your repository. Now, I won't take your pull request unless you have merged my recent work with yours before asking me. You want to be able to pull from my repository often enough from within your fork. So instead of having to type:

```
hg pull https://[yourname]@bitbucket/kballou/hgyelp
```

You can simply type:

```
hg pull MAIN
```

Or whatever you may call it after enter the alias in your fork's `hgrc` file. That is, you will add a line similar to the following to your `.hg/hgrc`

```

[paths]
default = https://[yourname]@bitbucket.org/[yourname]/hgyelp
MAIN = https://[yourname]@bitbucket.org/kballou/hgyelp

```

So you can still pull from your fork and you can easily reference the main repository without having to repetitively type that URL.

Granted this is a simplistic example of what you can do with this, it should get you started. Sadly, because the `hgrc` as previously mentioned is specific to the repository, you will have to change this for each repository (OR setup templates for Mercurial to use each time it creates a repository. No, I don't know how to do that, but it *should* be possible... should...)

Example 2 Similar to our previous example, with the slight difference in that instead of doing named paths, we can specify where to pull and where we should be pushing.

We can clone from where we should be pulling everything. Then we will add a `default-push` value to this new checked out repository's `hgrc` file.

```
[paths]
default = URI_TO_MASTER
default-push URI_TO_PUSH
```

You should note, the `default-push` value will be the preferred URL for all `hg push` operations. (Except, of course, ones with an explicit named path.) This way you can still get stable updates from the production repository while pushing changes to your test repository. Your test repository would get to production through other means like the tools provided by the hosted server?

Reference [Mercurial URLs](#) if you want to learn more about this.

6 Converting VCS

If you were using some other tool to do your source control and you want to switch, you are in luck. I personally cannot say how well and perfectly the convert extension works with any of the other source control tools. I do know that my initial few times of doing it with a fairly large project went pretty swimmingly. (Well, actually, that's not necessarily true. See CVS section below.) That said, I'm only going to talk about the basic things needed to get it setup and running. The rest is up to you and the [wiki](#).

6.1 Setting it Up

To convert your previous repository you will need to setup that tool (which is mostly likely already setup...)

Next you will need to add a line to the `extensions` section of your `.hgrc` file; whichever one you prefer.

```
[extensions]
hgext.convert=
```

As per the wiki, you will need to checkout the project/ repository/ module you want to convert. After which, you will run the `hg convert {moduleName} [where]`

6.1.1 CVS

```
#Check out the HEAD of the project
cvs co -d moduleName modulePath/moduleName
#Because I wanted to know how long it took
#convert the checked-out project
time hg convert moduleName
```

There are a lot of extra options that can be added to this command, like username mapping, branch mapping, file mapping, and a few others. But I will let the [Wiki](#), `hg help convert`, or the [HG Book](#) explain those.

Notice There are a few things to consider when converting CVS modules to Hg repositories. Namely, “Mercurial’s ConvertExtension uses a naive algorithm to understand CVS.” That is, if the CVS module to be converted doesn’t have tags or branches, the convert should be fine. However, if that’s not the case, you may want to consider looking into some other extensions for converting your CVS modules to Mercurial ([CVS Convert Wiki](#)).

Update: This might be out of date. As of (or maybe earlier) version 2.2.3 it seems Mercurial does a sane conversion of CVS modules to HG repositories. The one I keep converting seems to work correctly. But what do *I* know?

7 TortoiseHg – Mercurial Basics

Here we will do a very general tour of a GUI tool for Mercurial. Namely: [TortoiseHg](#). I will only be covering a way to do all the options listed in the Mercurial Basics Section.

7.1 Creating a Repository

To create a Mercurial repository with TortoiseHg, we do a very similar step as we did before: tell Mercurial where we want a repository. But with TortoiseHg, at least on Windows, we can just right click in the Windows Explorer and say “Create Repository Here”

7.1.1 Steps

1. Right-Click in Windows Explorer Directory
2. In the context menu, select TortoiseHg->Create Repository Here
3. Specify Destination Path and any other extra options desired
4. Click Create

7.1.2 Example

View Figures 1 through 5.

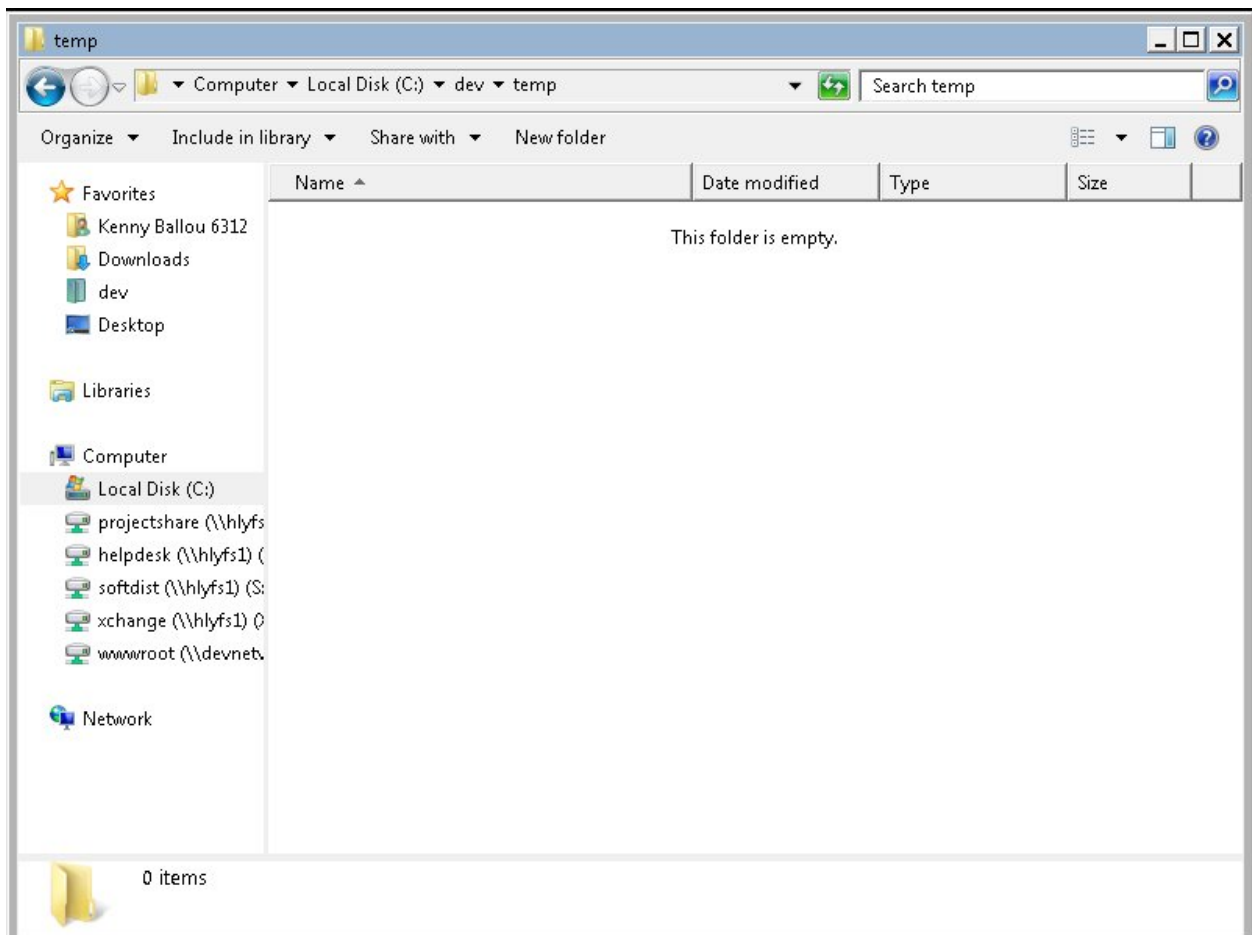


Figure 1: Empty Directory

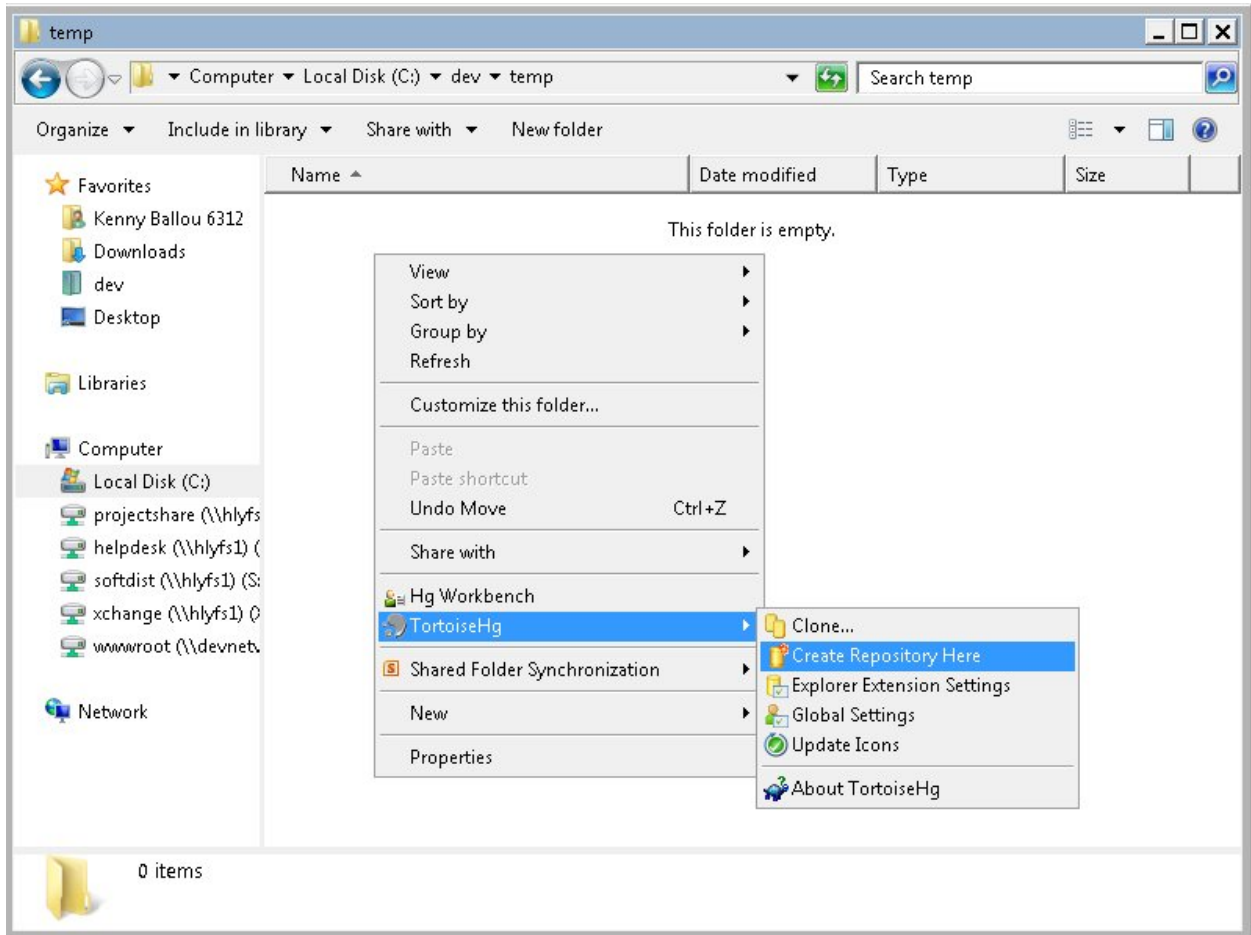


Figure 2: Context Create Repository

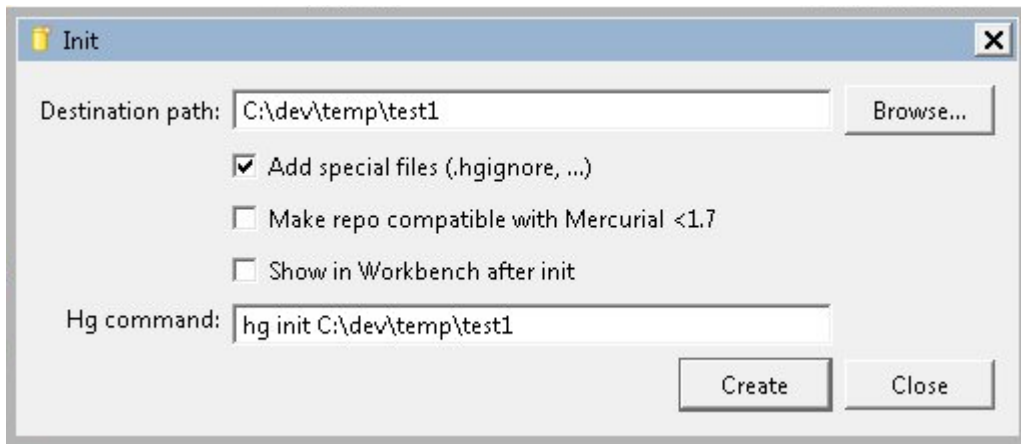


Figure 3: Create Repository Dialog

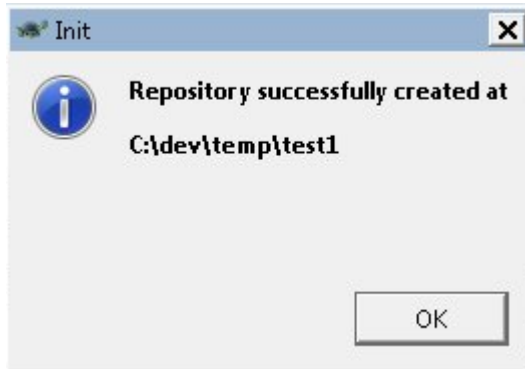


Figure 4: Success Message

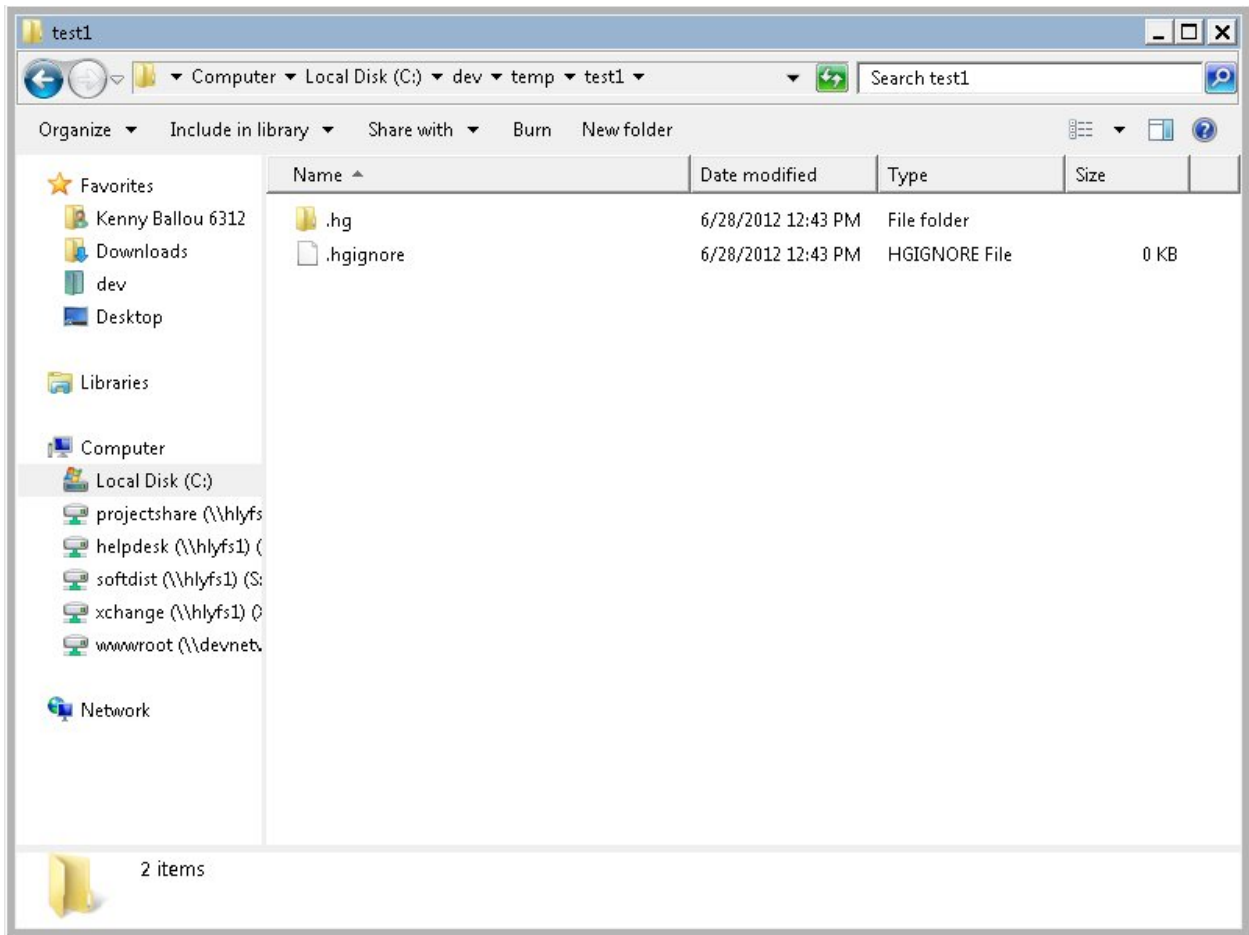


Figure 5: New Repository

7.2 Cloning a Repository

Like before, out steps to clone an existing repository are very simple and straight forward. Continually, I'm simply using the explorer context menus to demonstrate TortoiseHg. So, right click and select "Clone". You will need to provide host information and optionally give a location when cloning a remote repository. However, here I'm only doing a local clone.

7.2.1 Steps

1. Right-Click (a repository folder (for local clones)) in Windows Explorer Directory
2. In the context menu, select **TortoiseHg->Clone...**
3. Specify Source (and destination, if different). You may also want to specify a revision or whatever else. To do so, select the "Options" and provide the extra information you desire.
4. Click Create

7.2.2 Example

View Figures 6 through 8.

7.3 Repository Status

TortoiseHg provides a dialog that allows you to see the status of your repository. However, because of Windows Explorer Icons, this is almost not needed.

7.3.1 Steps

1. From within Windows Explorer, Right-Click the repository's root folder, empty space of the repository's folder, or any file selection under the repository
2. In the context menu, select **TortoiseHg->View File Status**

7.3.2 Example

View Figures 9 through 10.

7.4 Adding Files

To add files, we will need files to add. Create some files. Then, using the context menus, select add files and add the files you wish to add. We will go over committing the files in a second.

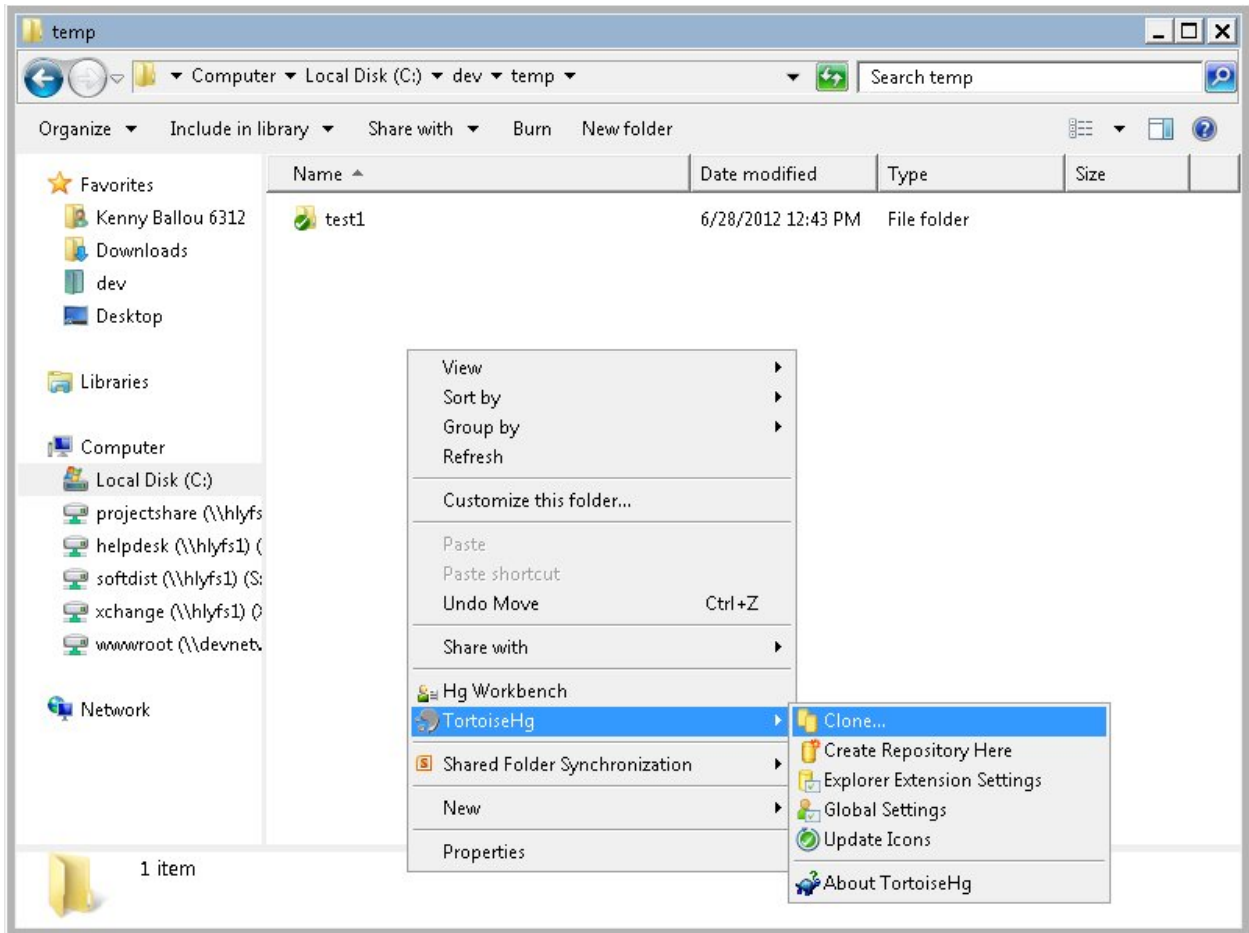


Figure 6: TortoiseHg -> Clone

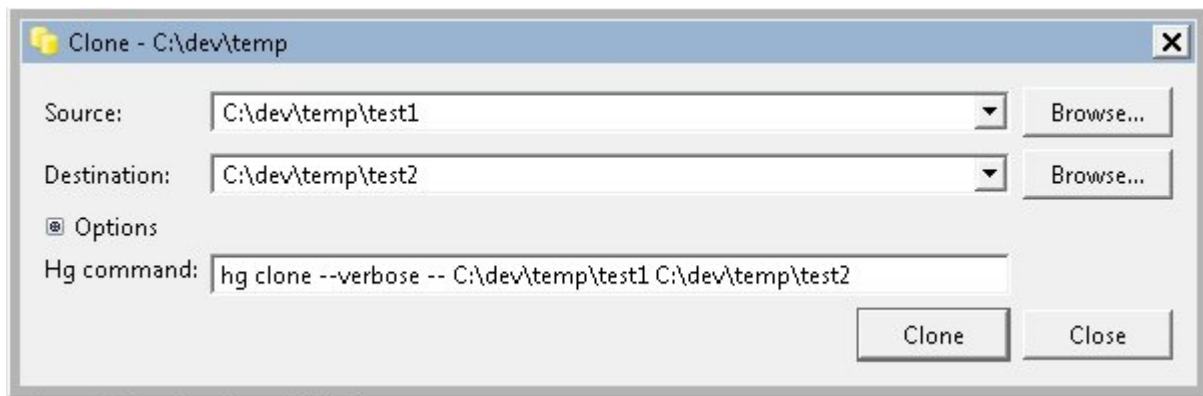


Figure 7: Clone Repository Dialog

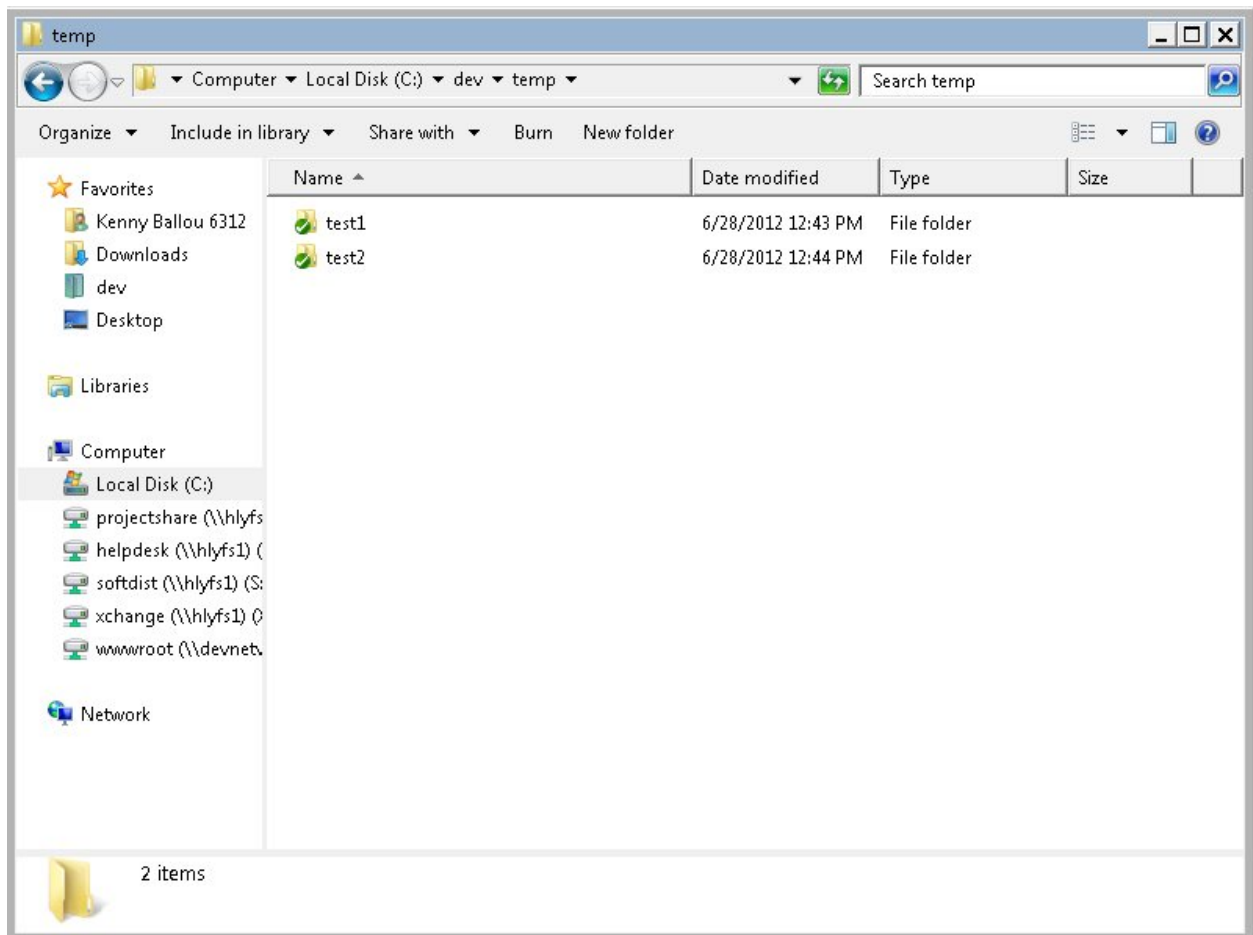


Figure 8: New Cloned Repository

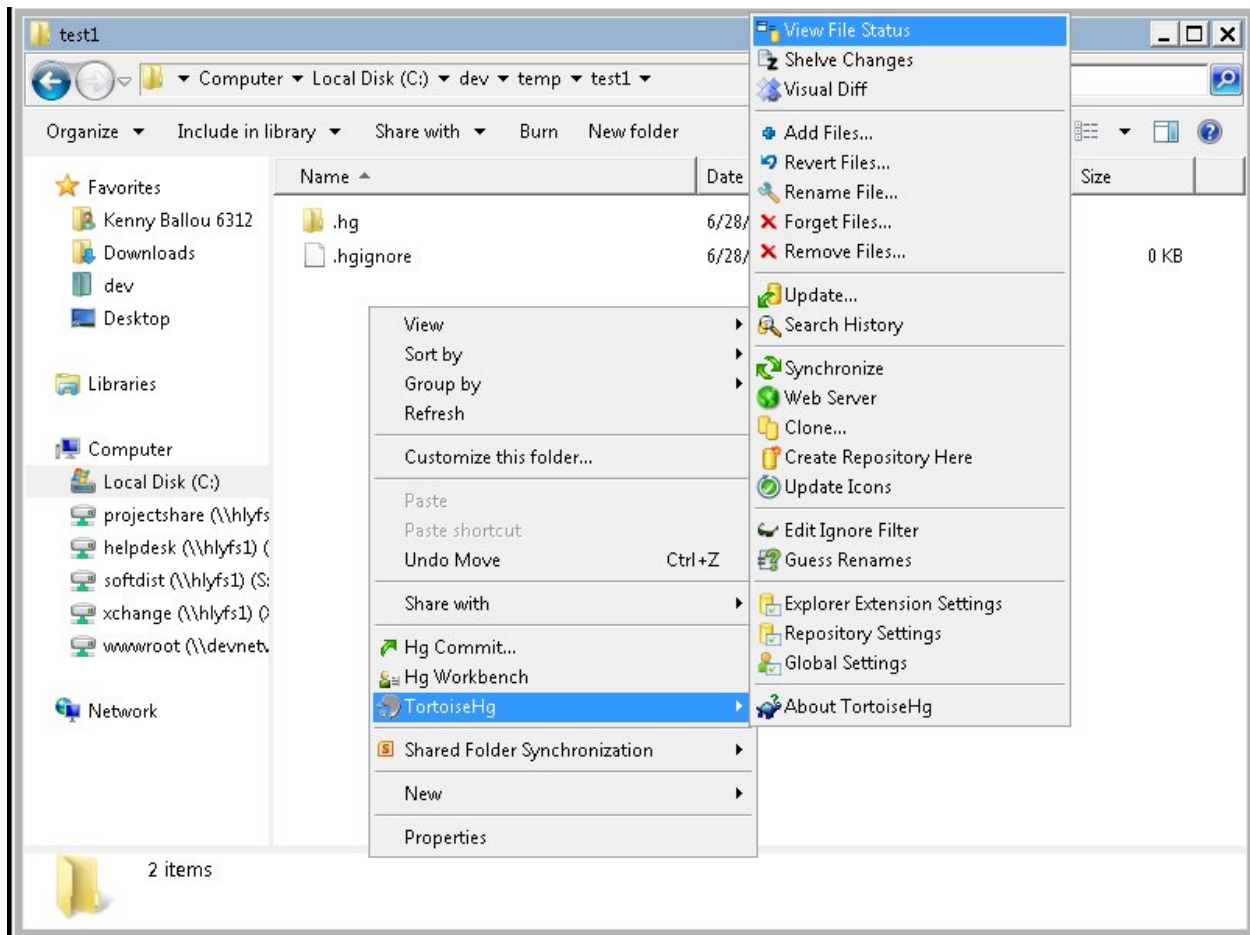


Figure 9: Context -> View File Status

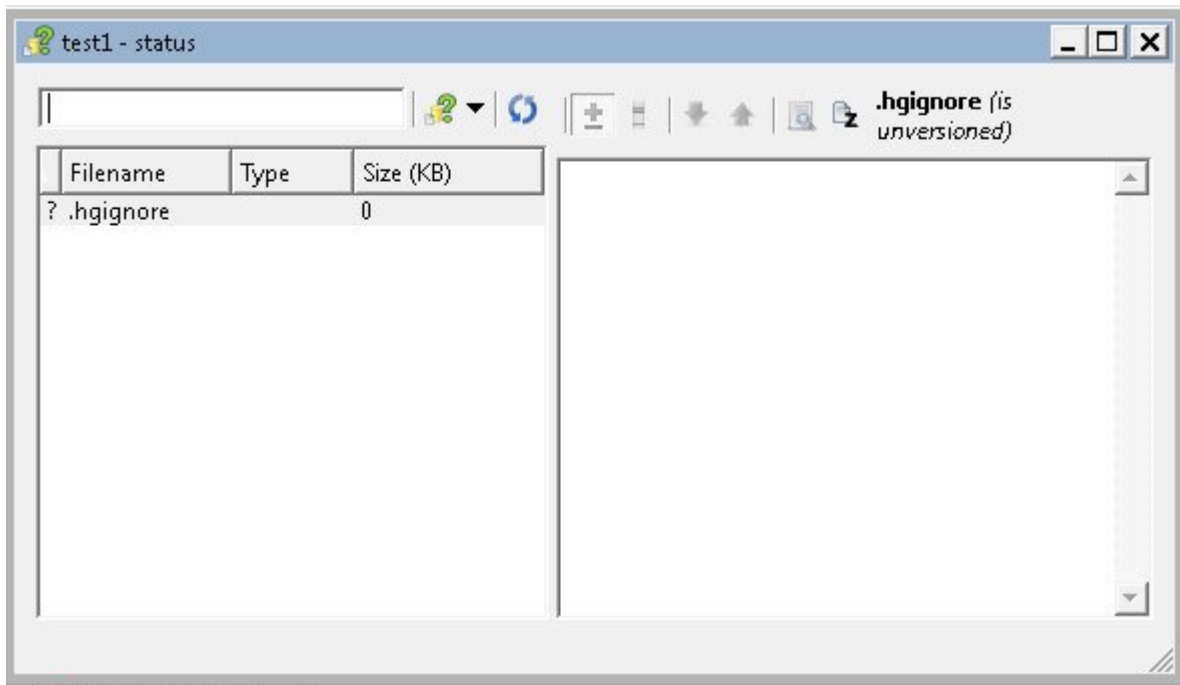


Figure 10: Status Dialog

7.4.1 Steps

1. Right-click the file(s)/folder(s) you wish to add
2. In the context menu, select TortoiseHg->Add Files...
3. In the Hg add file dialog, (de)select all the files to be added (if you need to edit your choice, that is)
4. Click “Add”
5. This action will need to be committed. (See committing.)

7.4.2 Example

View Figures 11 through 15.

7.5 Removing Files

Like adding, removing files is almost identical. Use the context menus to select the files you want to remove, and remove them. Notice, this will cause the file to be deleted. Check forget if you want to keep the file.

7.5.1 Steps

1. Right-click the file(s)/folder(s) you wish to remove
2. In the context menu, select TortoiseHg->Remove Files...

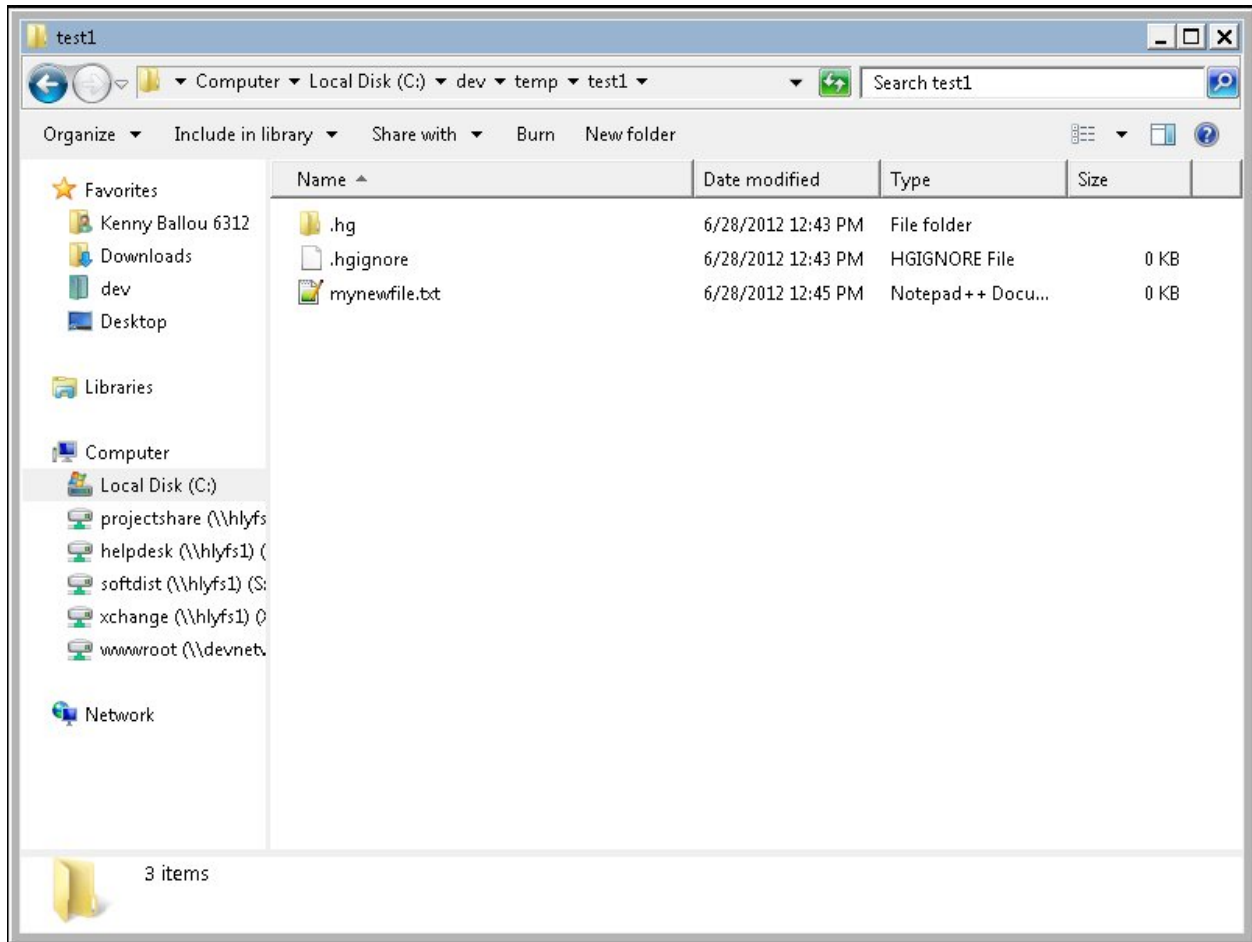


Figure 11: Creating Files

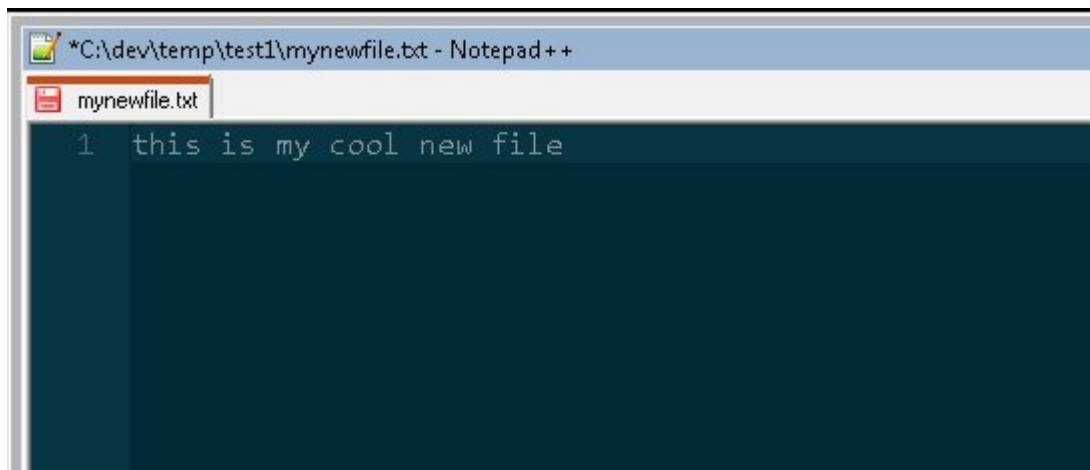


Figure 12: Adding contents to file

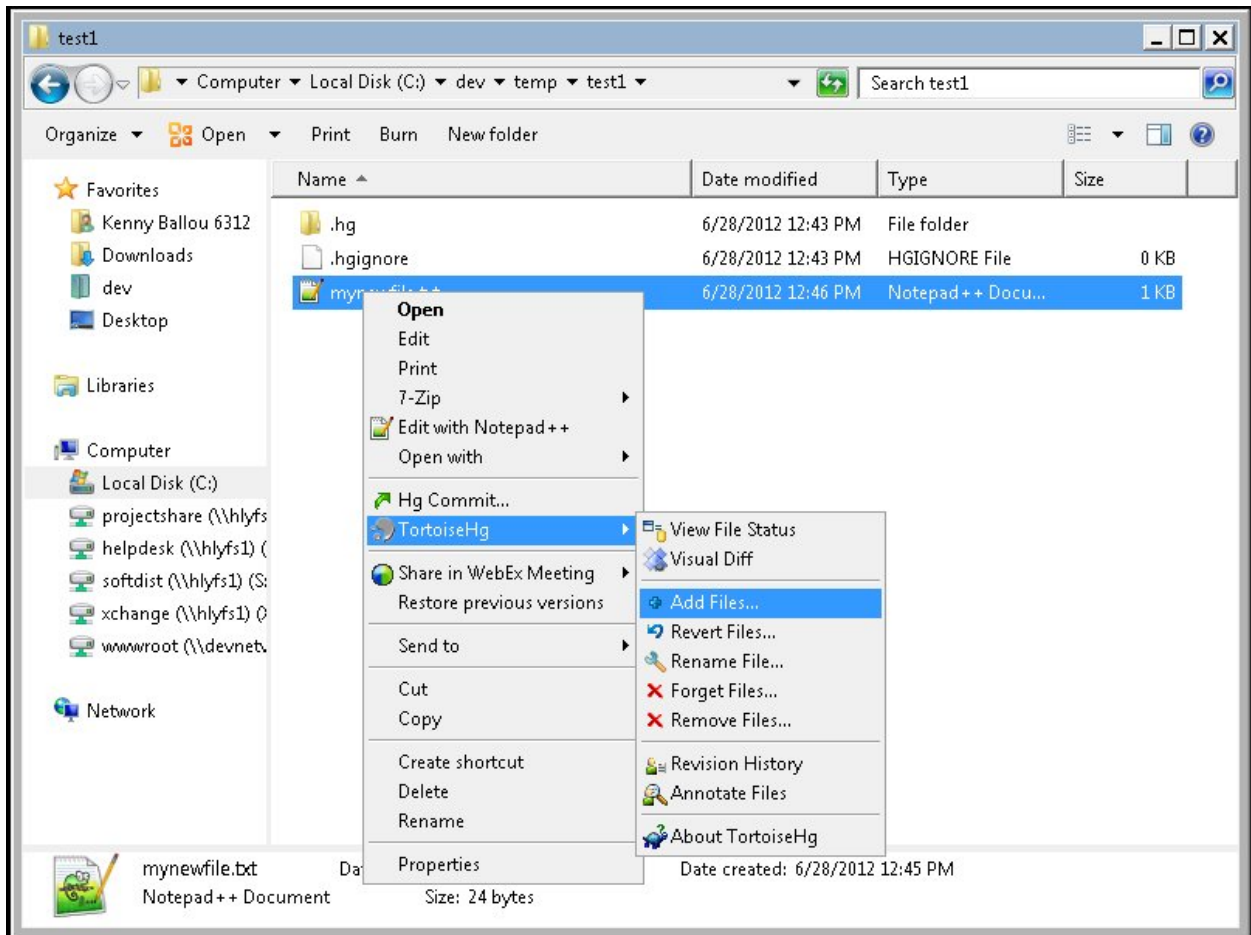


Figure 13: Adding files to repository

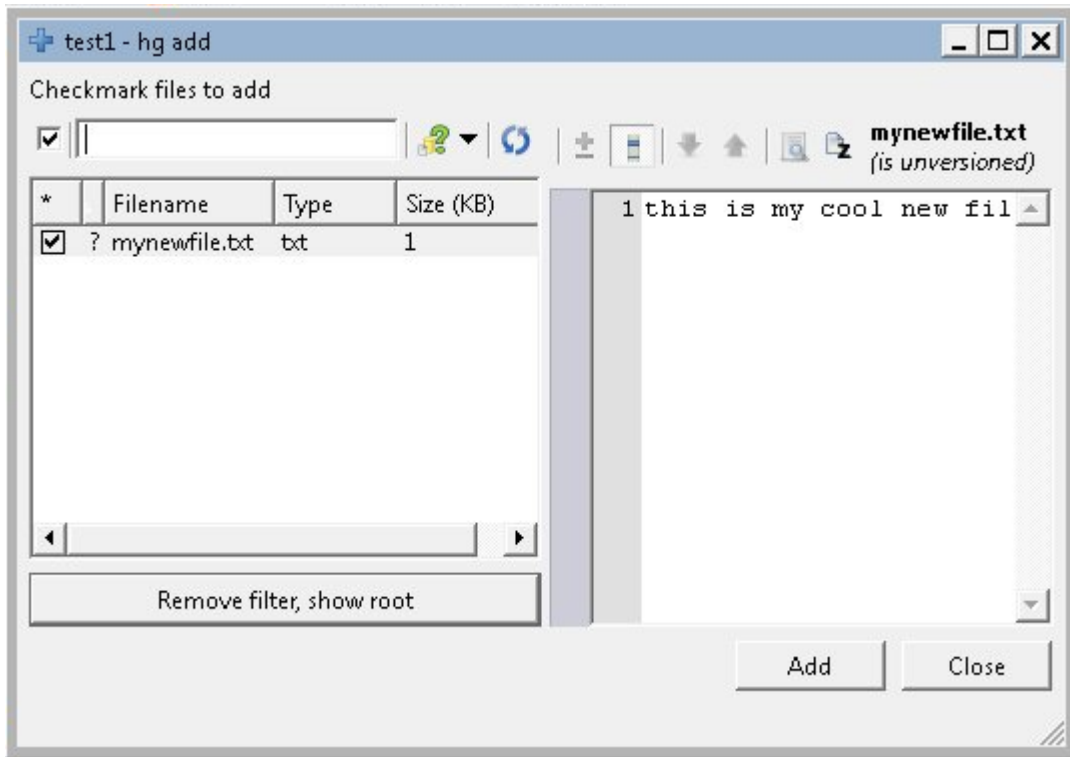


Figure 14: Add file dialog



 .hg	6/28/2012 12:47 PM	File folder	
 .hgignore	6/28/2012 12:43 PM	HGIGNORE File	0 KB
 mynewfile.txt	6/28/2012 12:46 PM	Notepad++ Docu...	1 KB

Figure 15: Examine new file status

3. In the Hg remove file dialog, (de)select all the files to be removed (if you need to edit your choice, that is)
4. Click “Remove”
5. This action will need to be committed. (See committing.)

7.5.2 Example

View Figures 16 through 18.

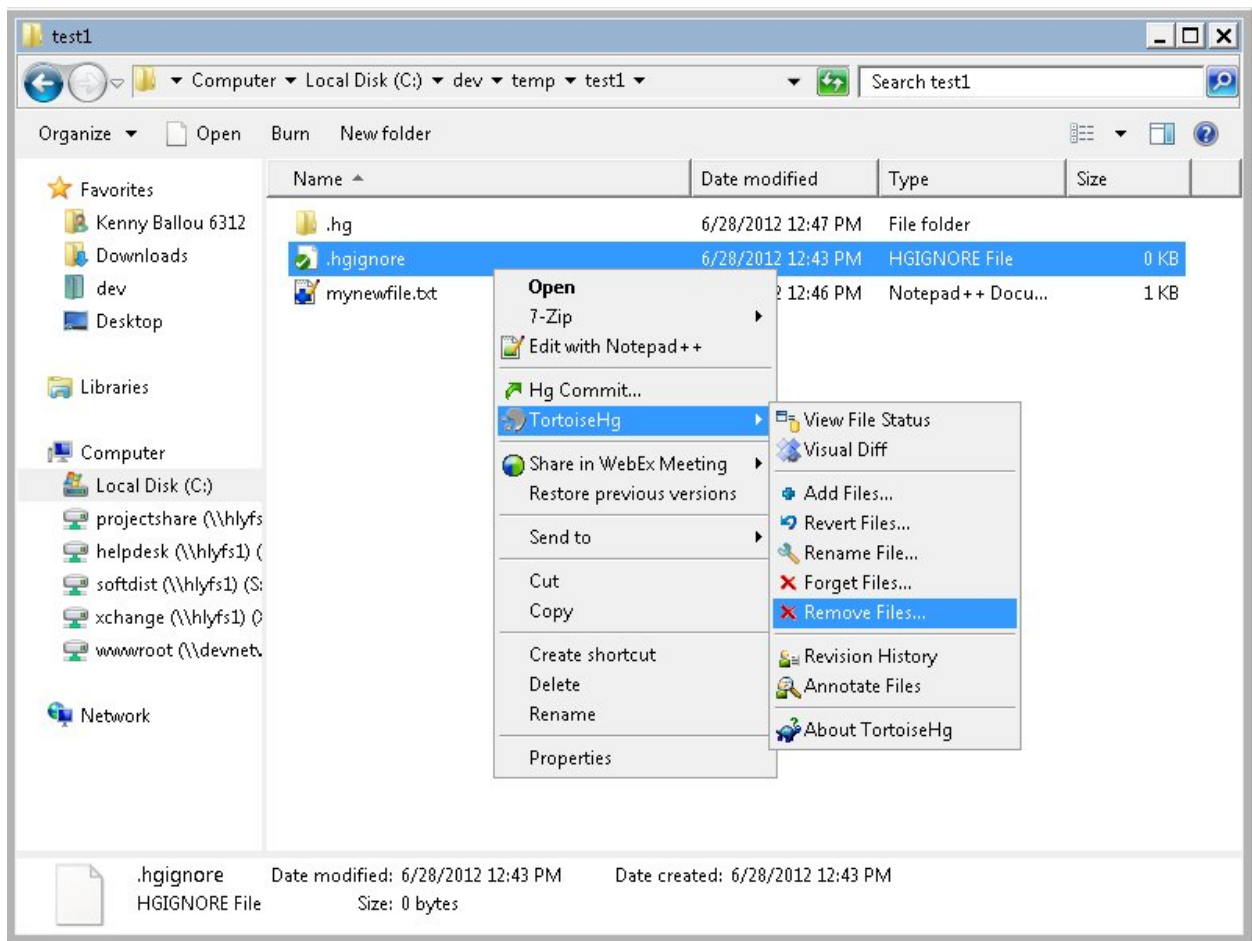


Figure 16: Context Remove files

7.6 Committing Changes

Committing files is no different than before. We use the context menu to tell TortoiseHg we are ready to commit and it prompts us a dialog, we provide the commit message, optionally (de)select the files we want to commit and commit we do. Notice, however, the commit dialog will stay active until you explicitly close it. This, if I were to guess, is because part of your (de)selection process, you might still want to commit those files, just with a different commit message.

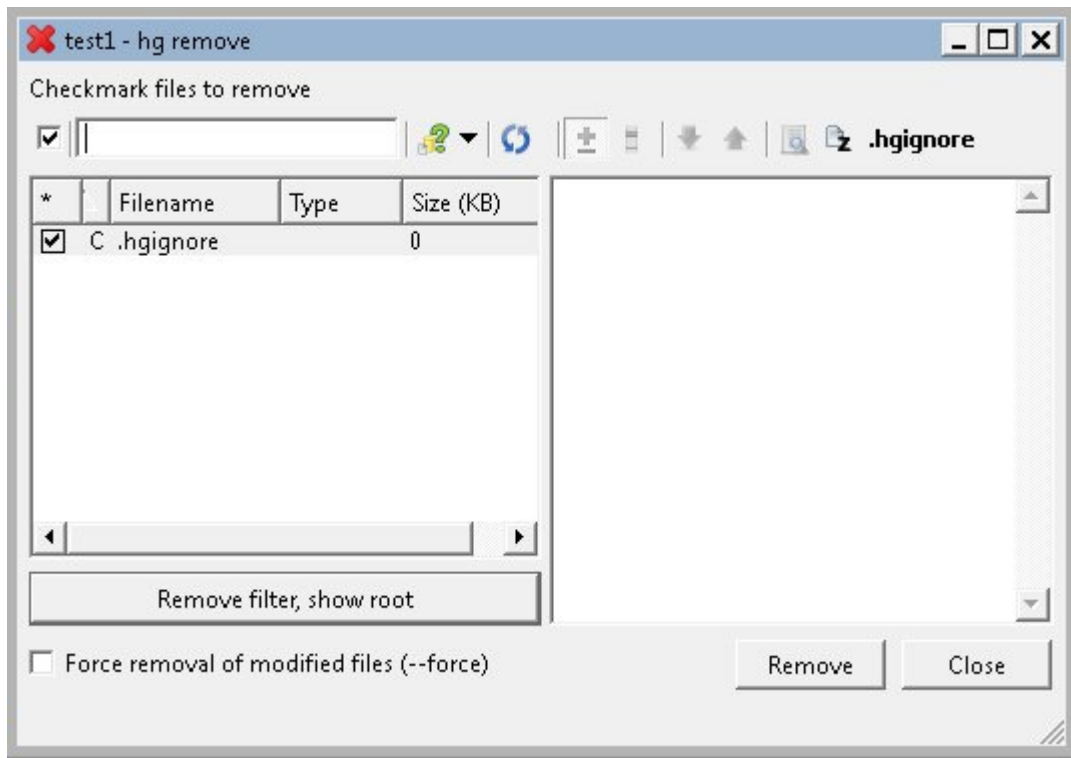


Figure 17: Remove file dialog

7.6.1 Steps

1. Right-click the repository's folder or any sub-folders or files to be committed
2. In the context menu, select Hg Commit...
3. In the Hg Commit dialog, (de)select any files you wish to commit.
4. Provide a commit message
5. Click "Commit"
6. Repeat 3-5 as necessary
7. Click "Close"

7.6.2 Example

View Figures 19 through 22.

7.7 Pulling Changes

To get changes from another repository, we must "synchronize". TortoiseHg is not changing Mercurial's internals when it requires us to synchronize; it is merely naming the context menu differently. This one option will let us both push and pull.

We will also be doing an update with the context menus here.

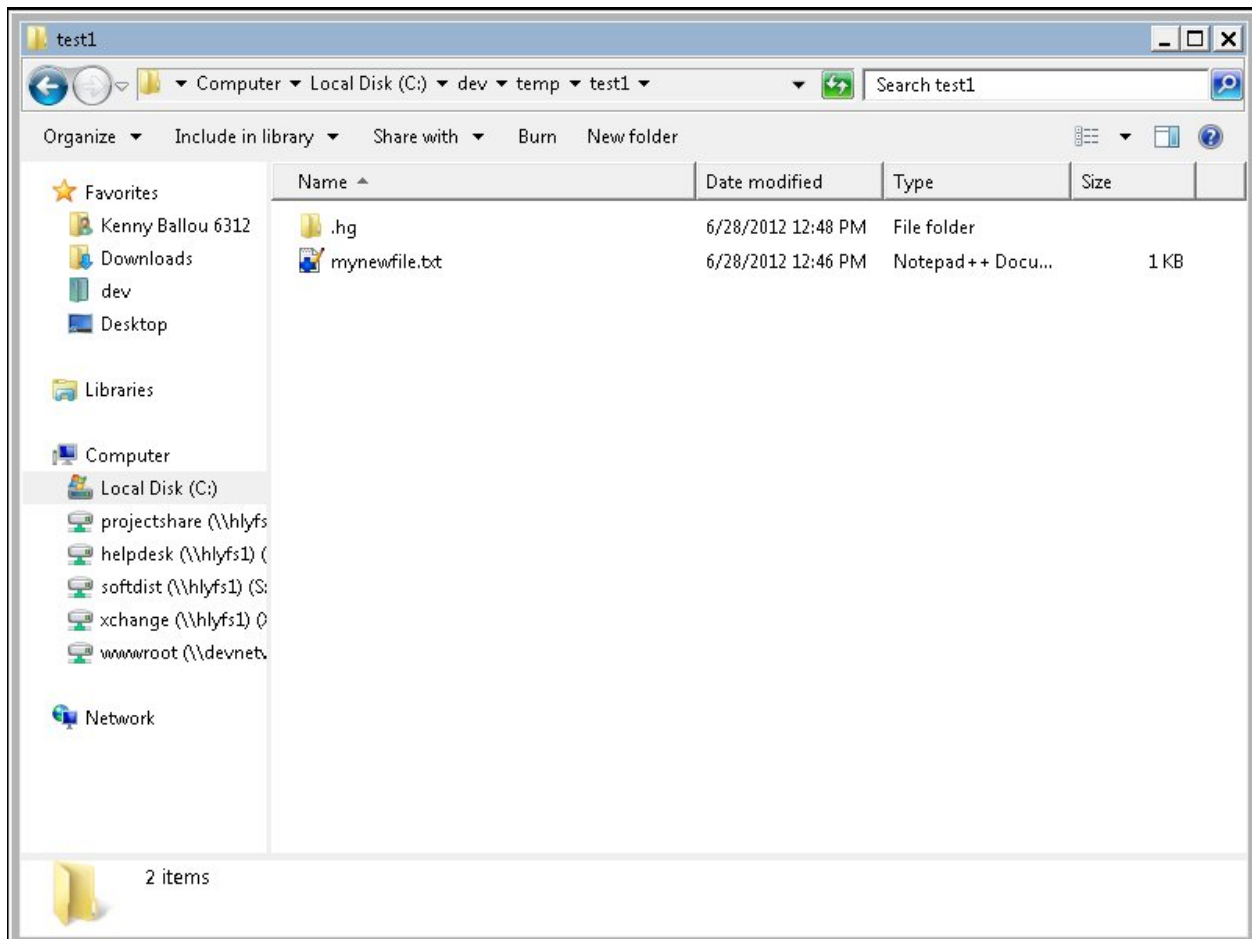


Figure 18: Examine new file status (or lack of file)

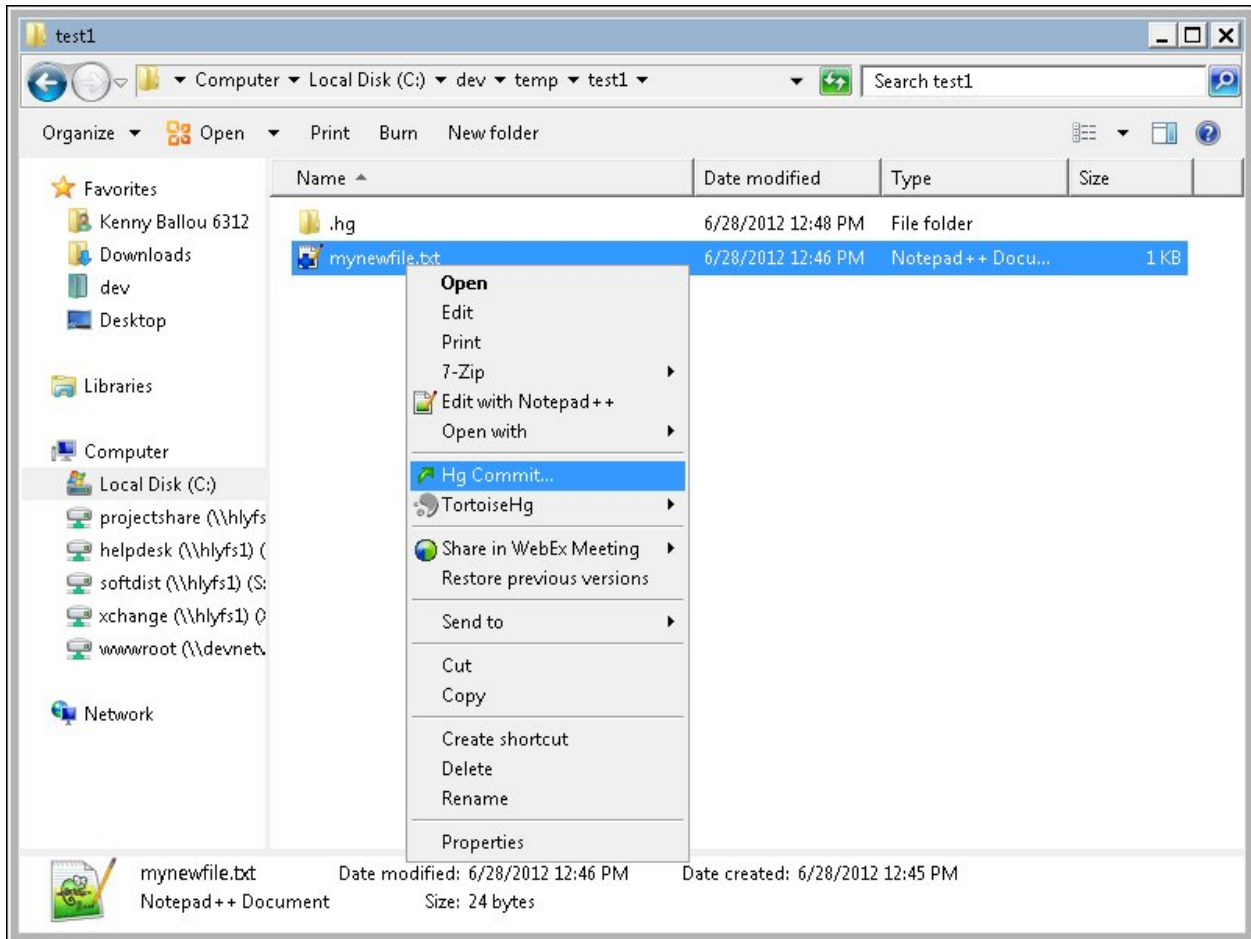


Figure 19: Context Commit Changes

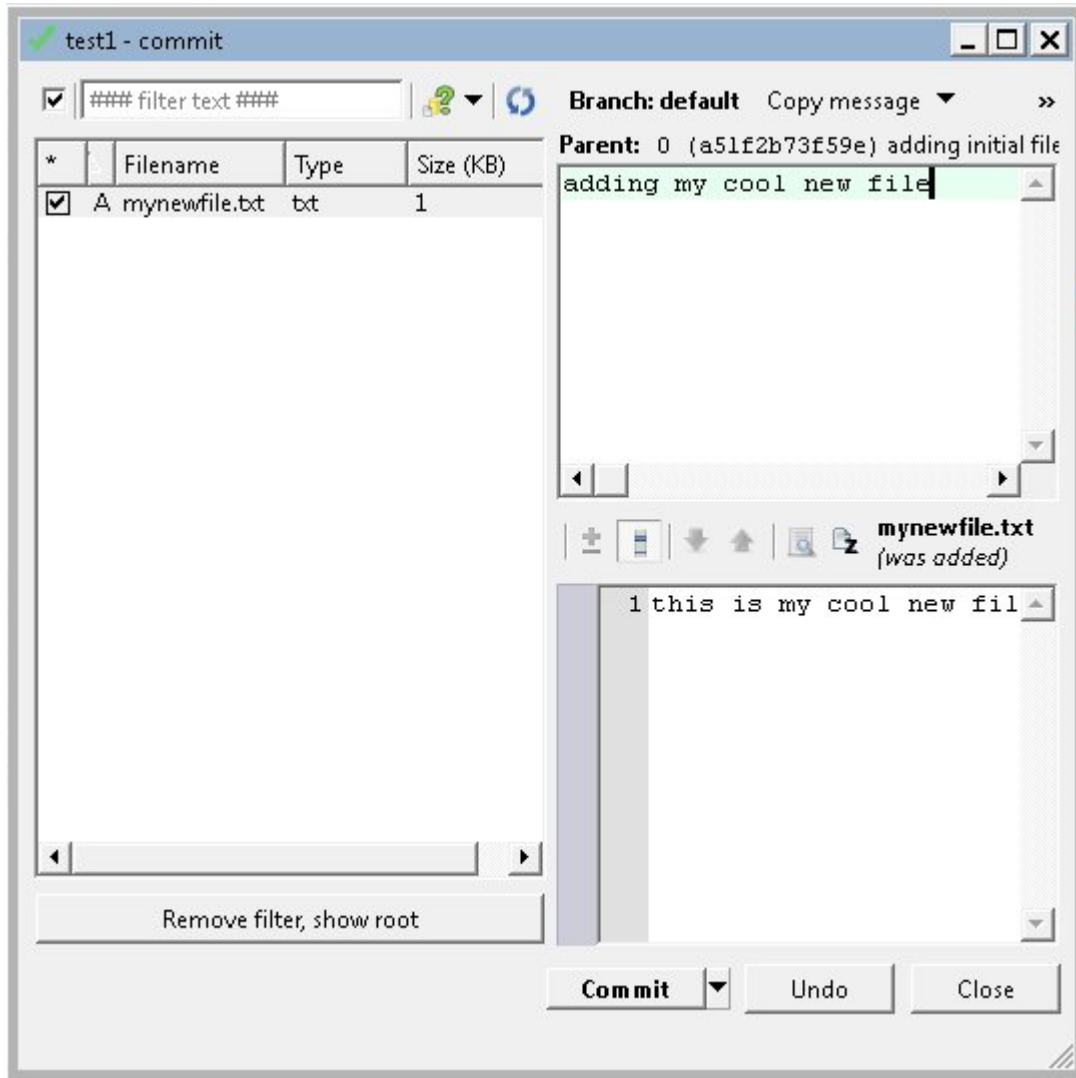


Figure 20: Commit Dialog with message

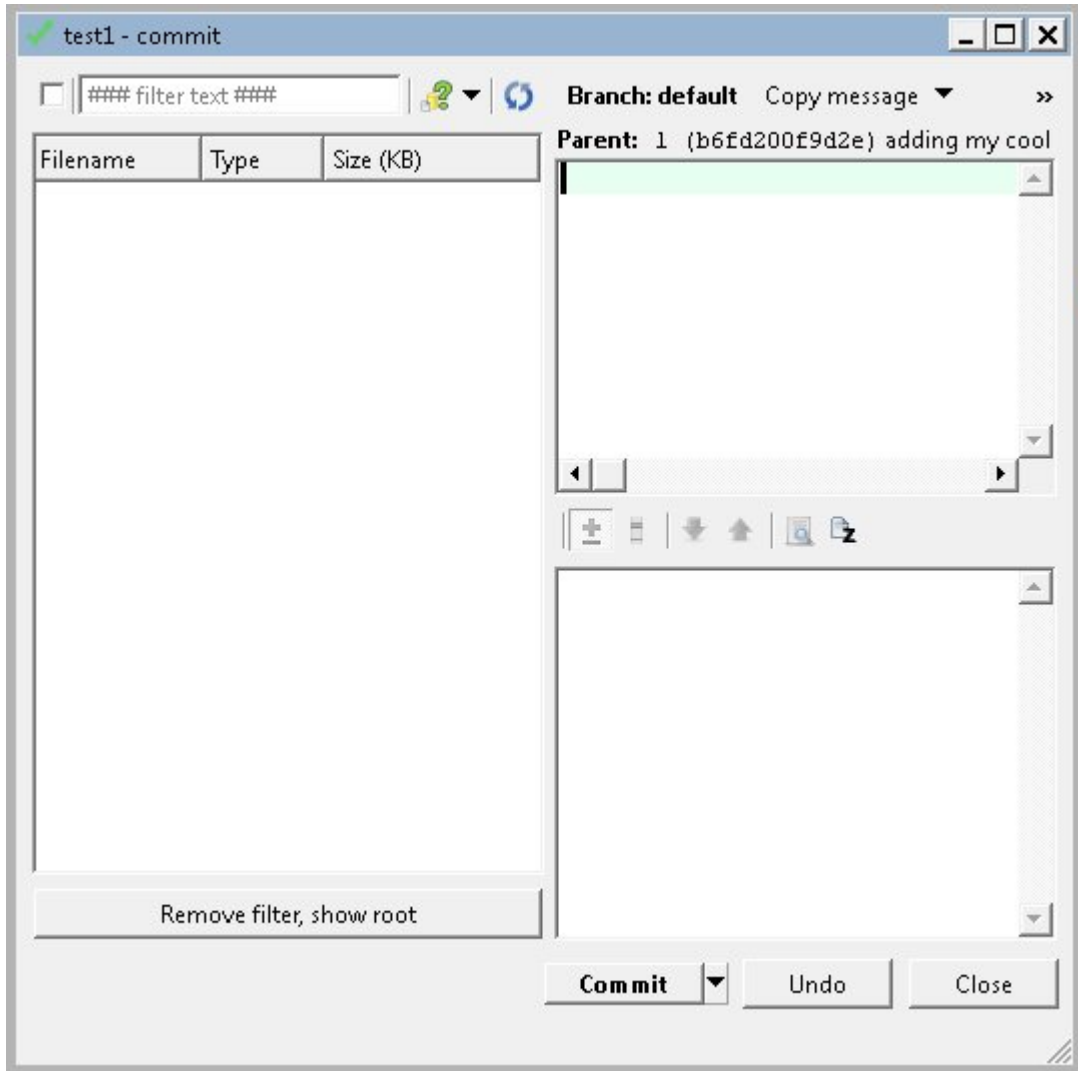


Figure 21: Commit Dialog (we're done).

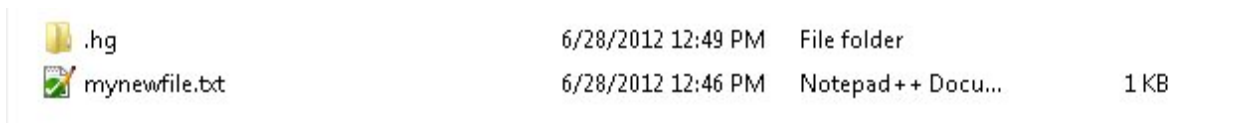


Figure 22: Examine new file status

7.7.1 Steps

1. Right click the repository you wish to pull changes for
2. In the context menu, select TortoiseHg->Synchronize
3. Click the second button in from the left, or the “Pull Changes” button
4. Close the synchronize dialog
5. Right-click the repository folder
6. In the context menu, select TortoiseHg->Update...
7. Specify the update options you desire (in this case, defaults are fine)
8. Click “Update”

7.7.2 Example

View Figures 23 through 28.

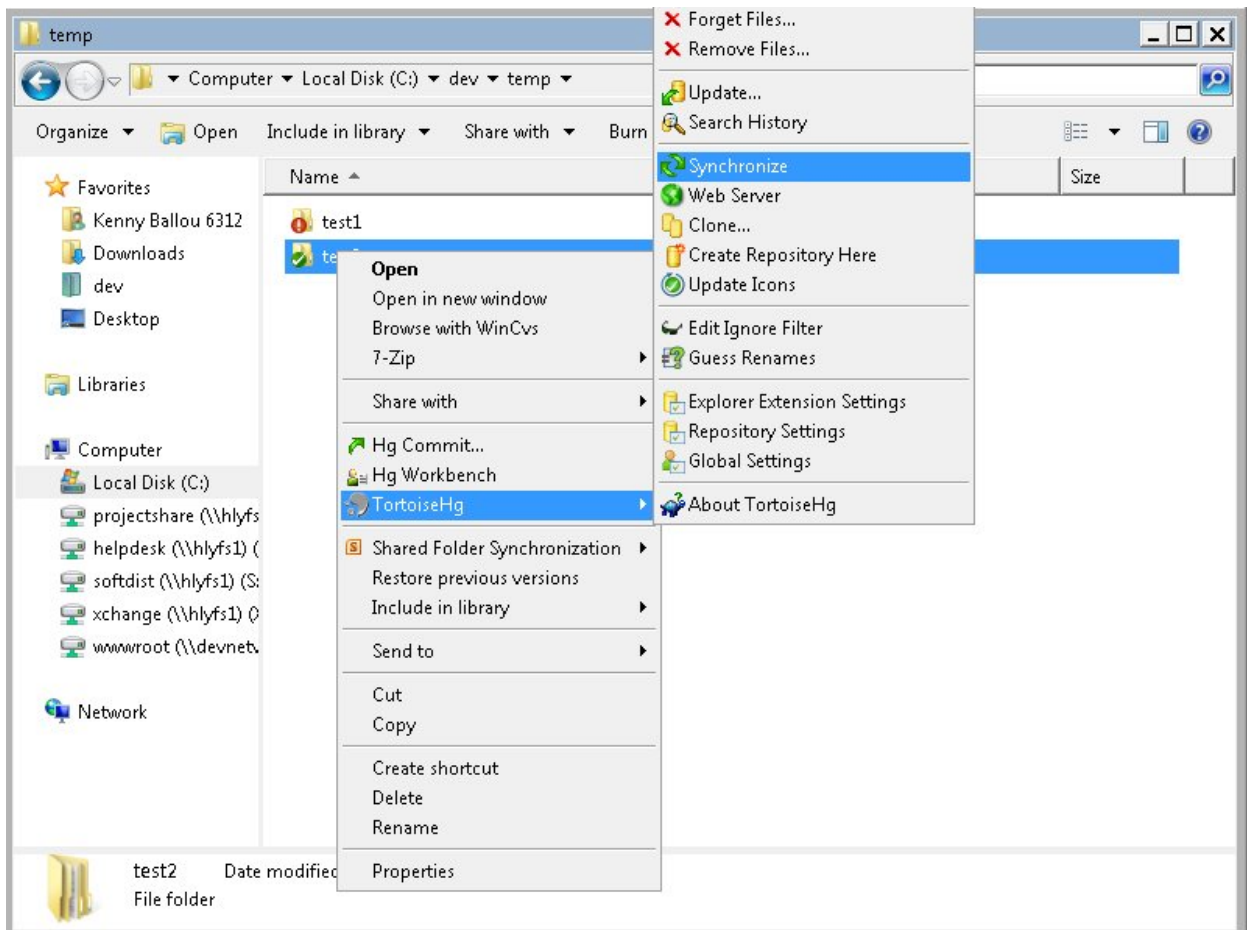


Figure 23: Context Menu Synchronize

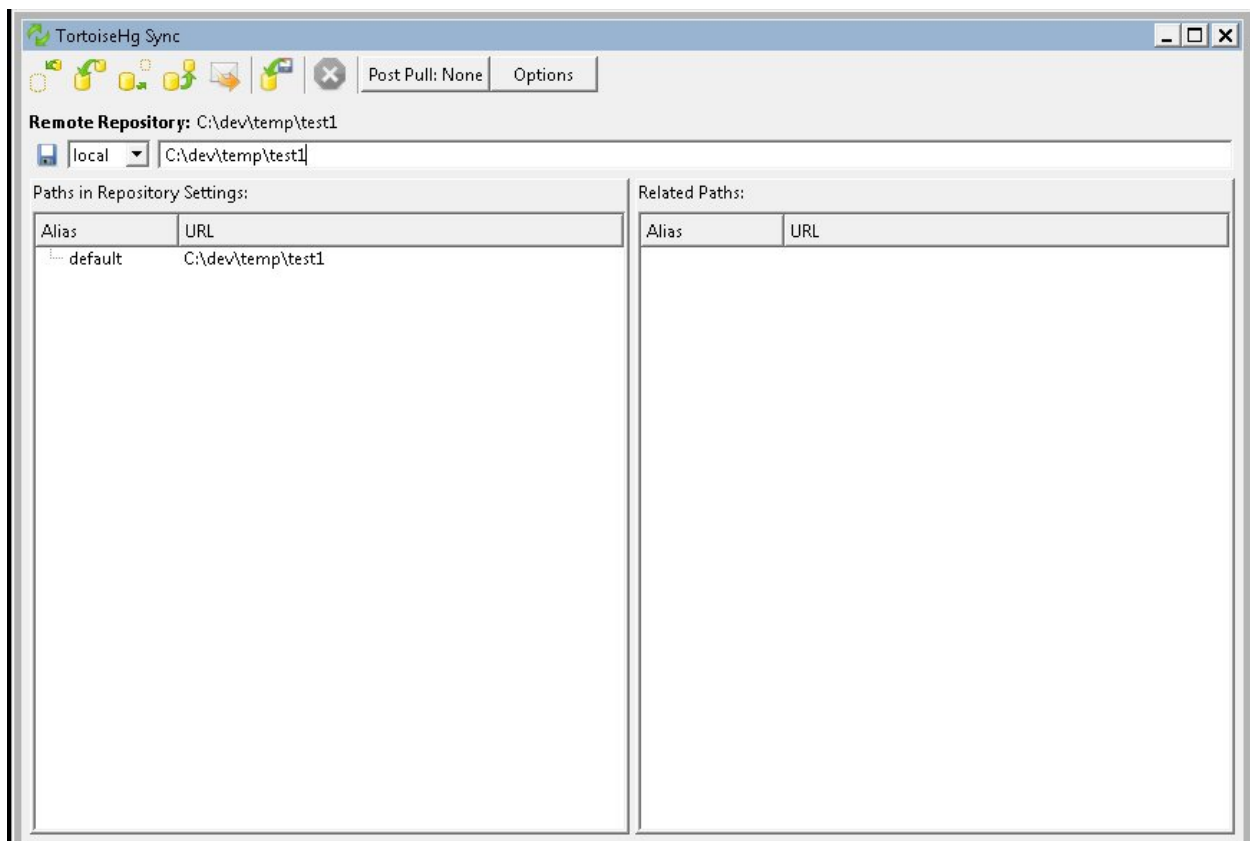


Figure 24: Synchronize Dialog

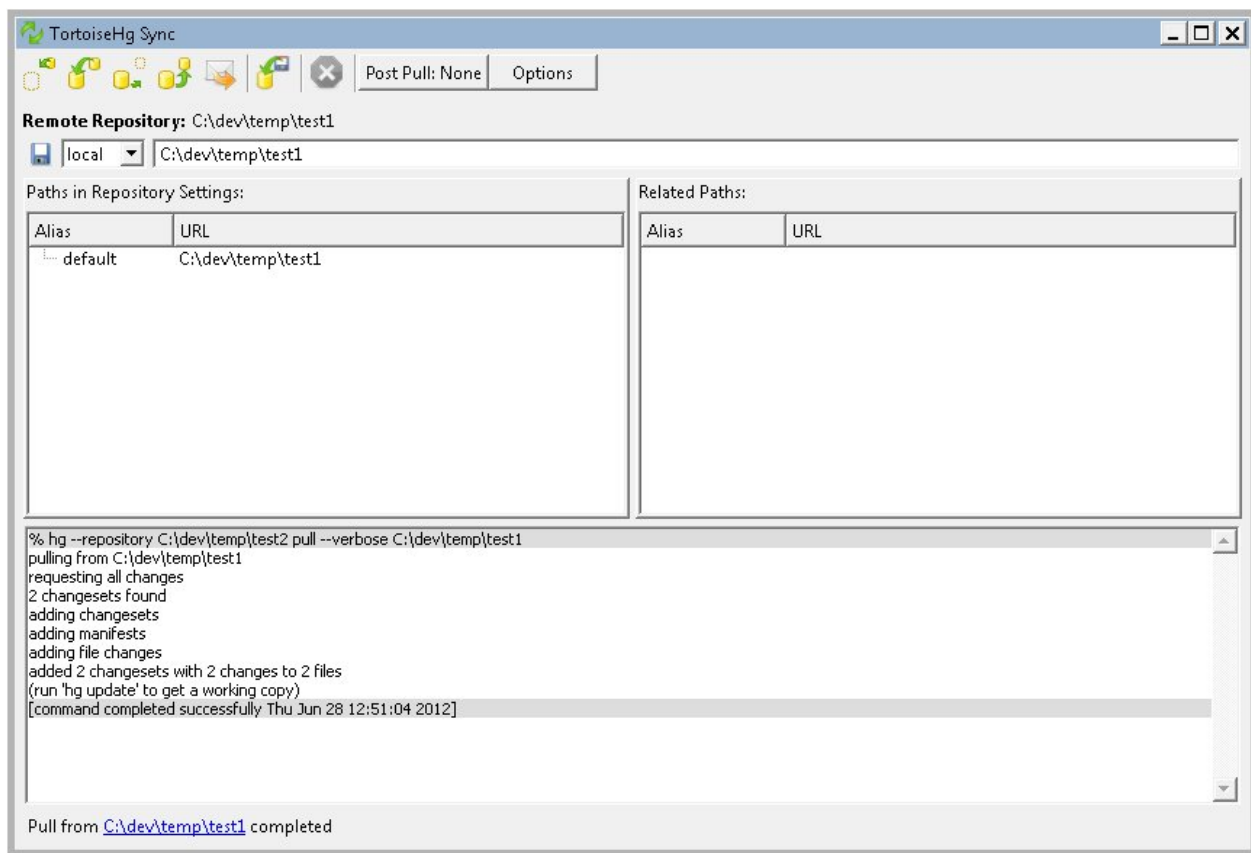


Figure 25: Synchronize Dialog after pulling

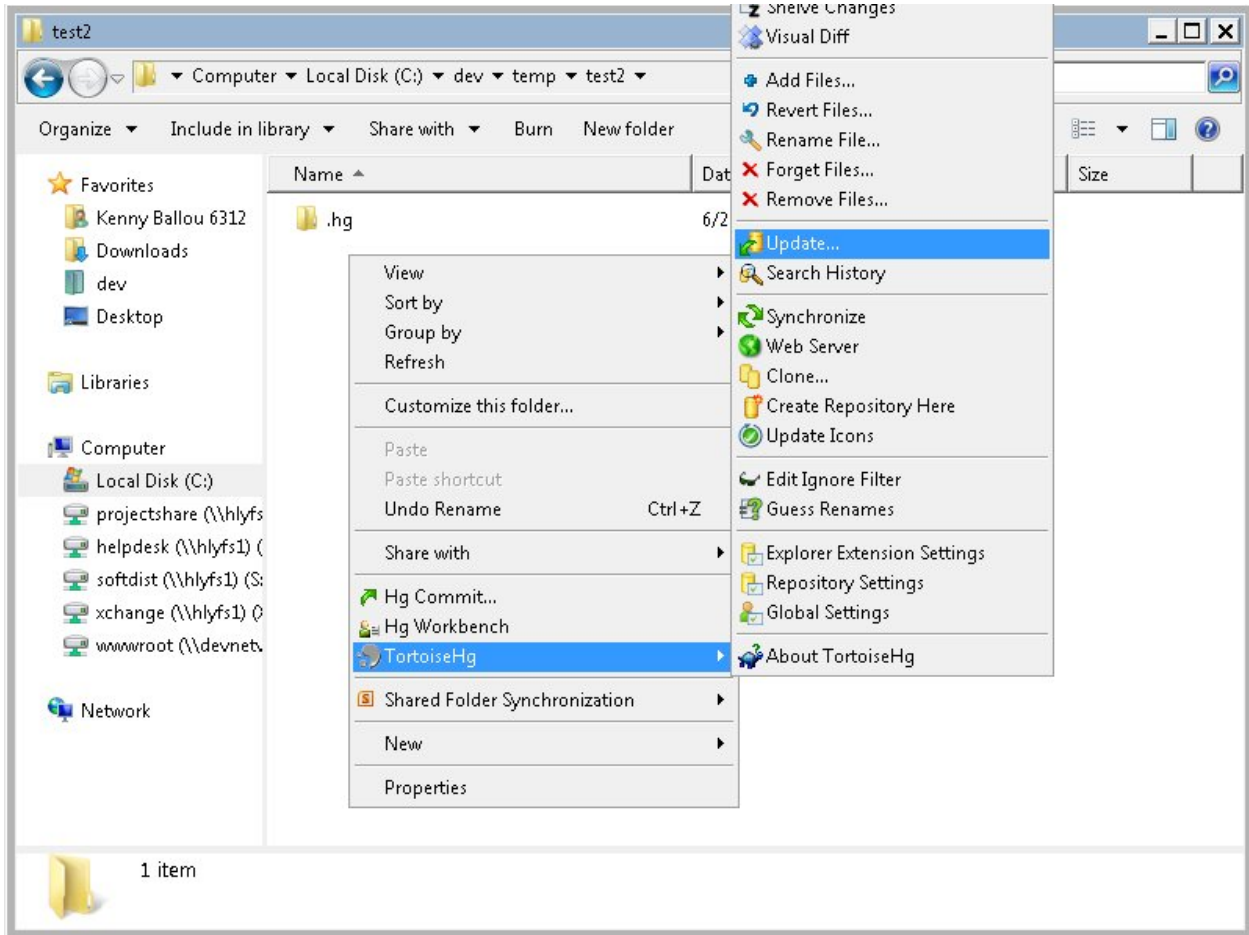


Figure 26: Context Menu Update

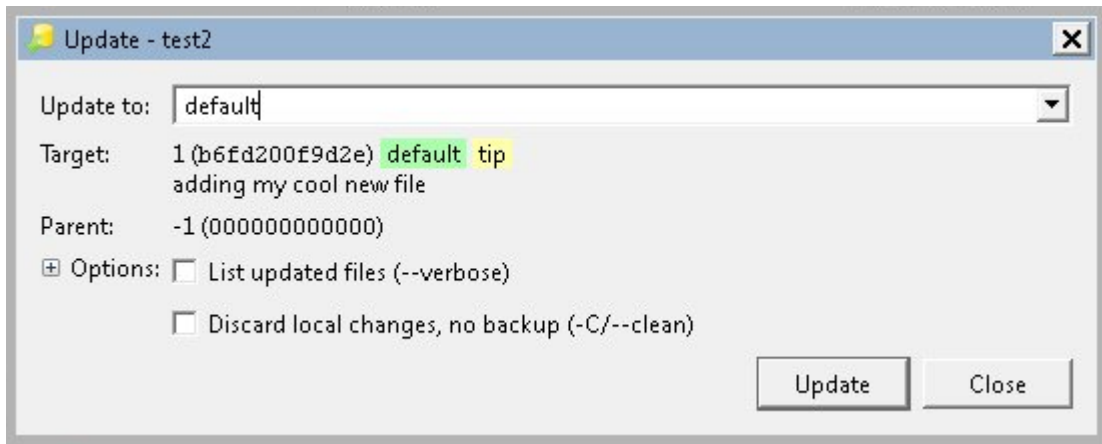


Figure 27: Update Dialog

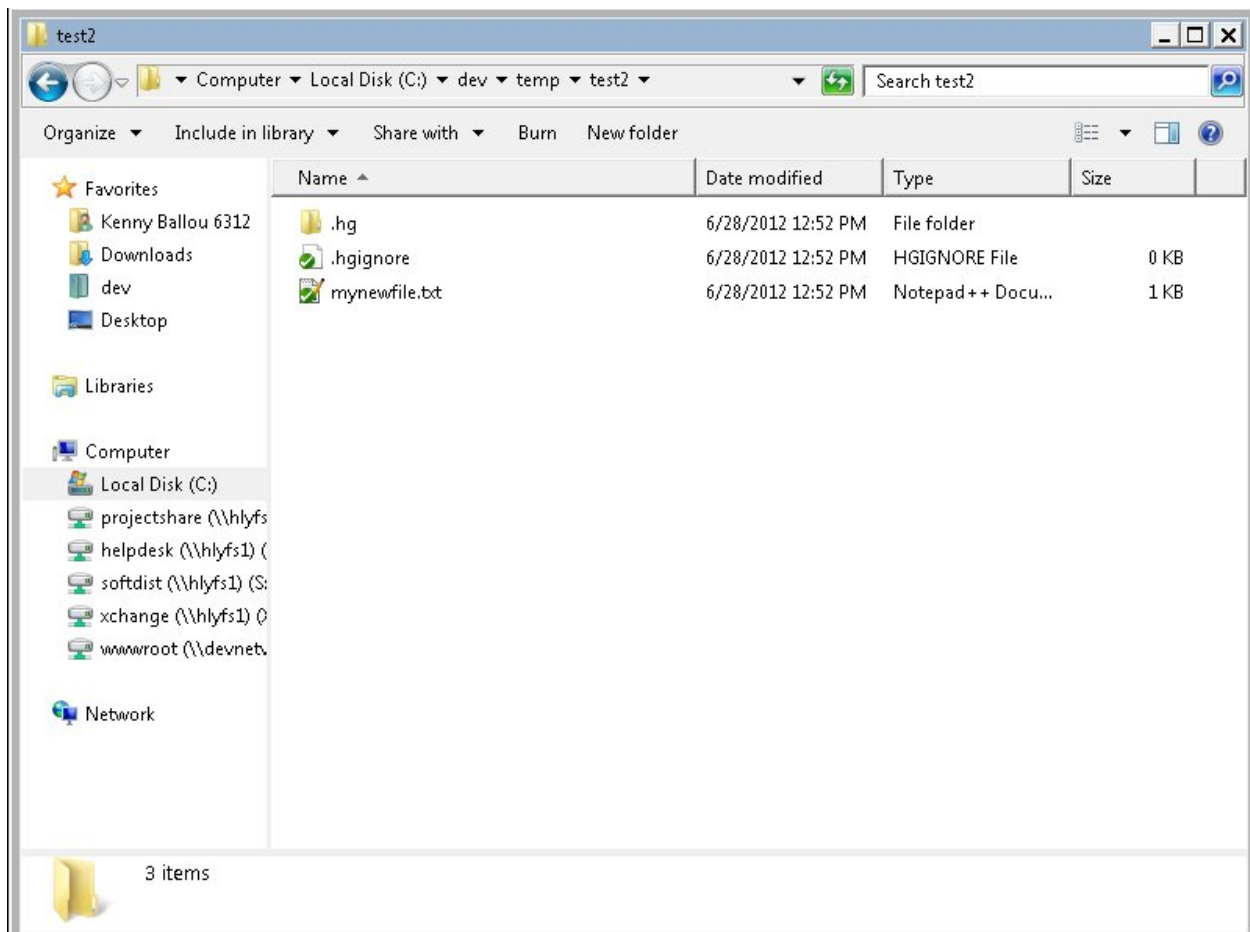


Figure 28: Examine new file status

7.8 Pushing Changes

Like with pulling, we will use the synchronize context menu option and then, instead, the push button. This is relatively straight forward. However, TortoiseHg should not be used for local repository pushing. See the collaboration section for notes on this. However, we are going to demonstrate a push, for the sake of demonstration.

7.8.1 Steps

1. Right click the repository you wish to push from
2. In the context menu, select **TortoiseHg->Synchronize**
3. Click the fourth button from the left, or the “Push Changes” button
4. Close the synchronize dialog

7.8.2 Example

View Figures 29 through 32.

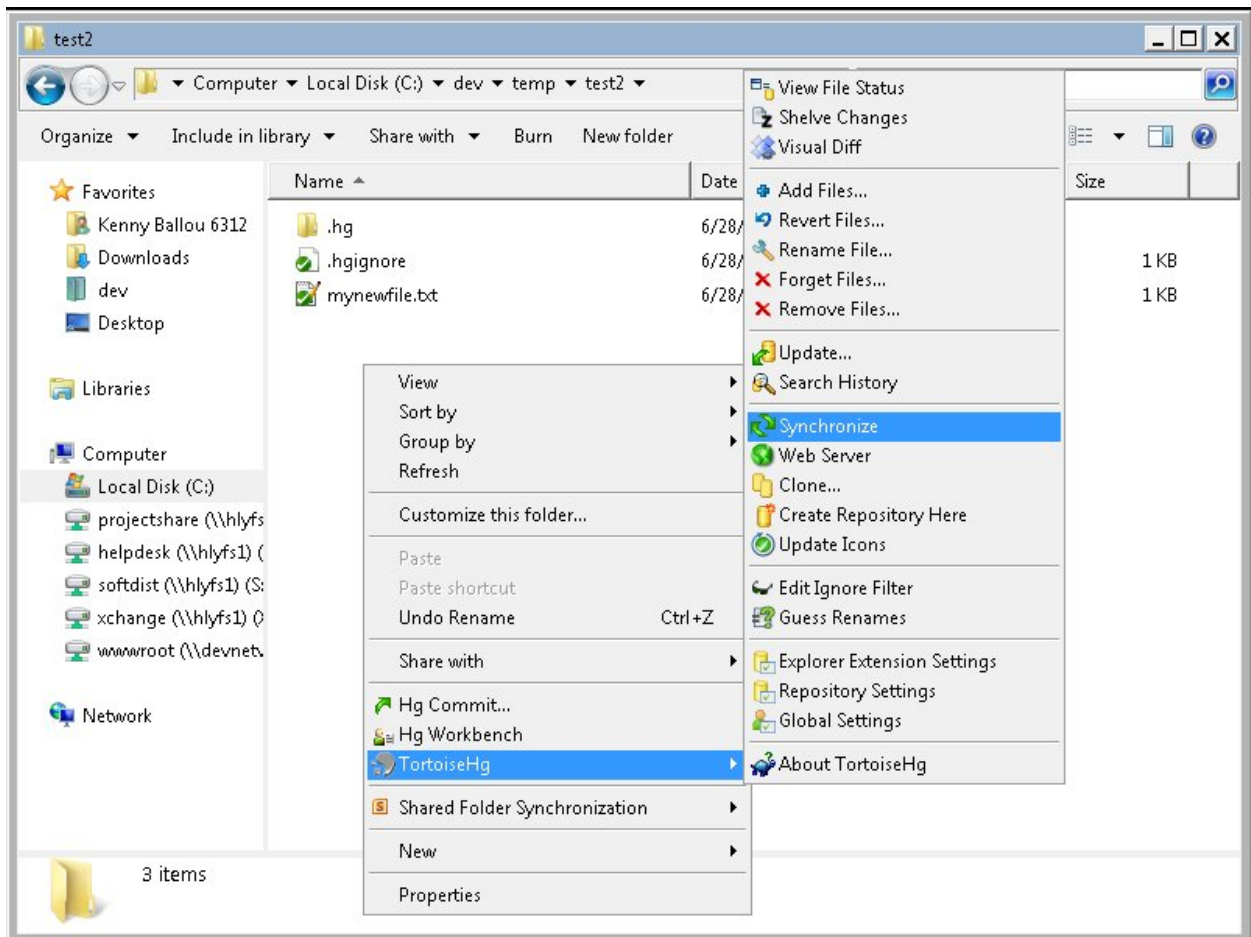


Figure 29: Context Menu Synchronize

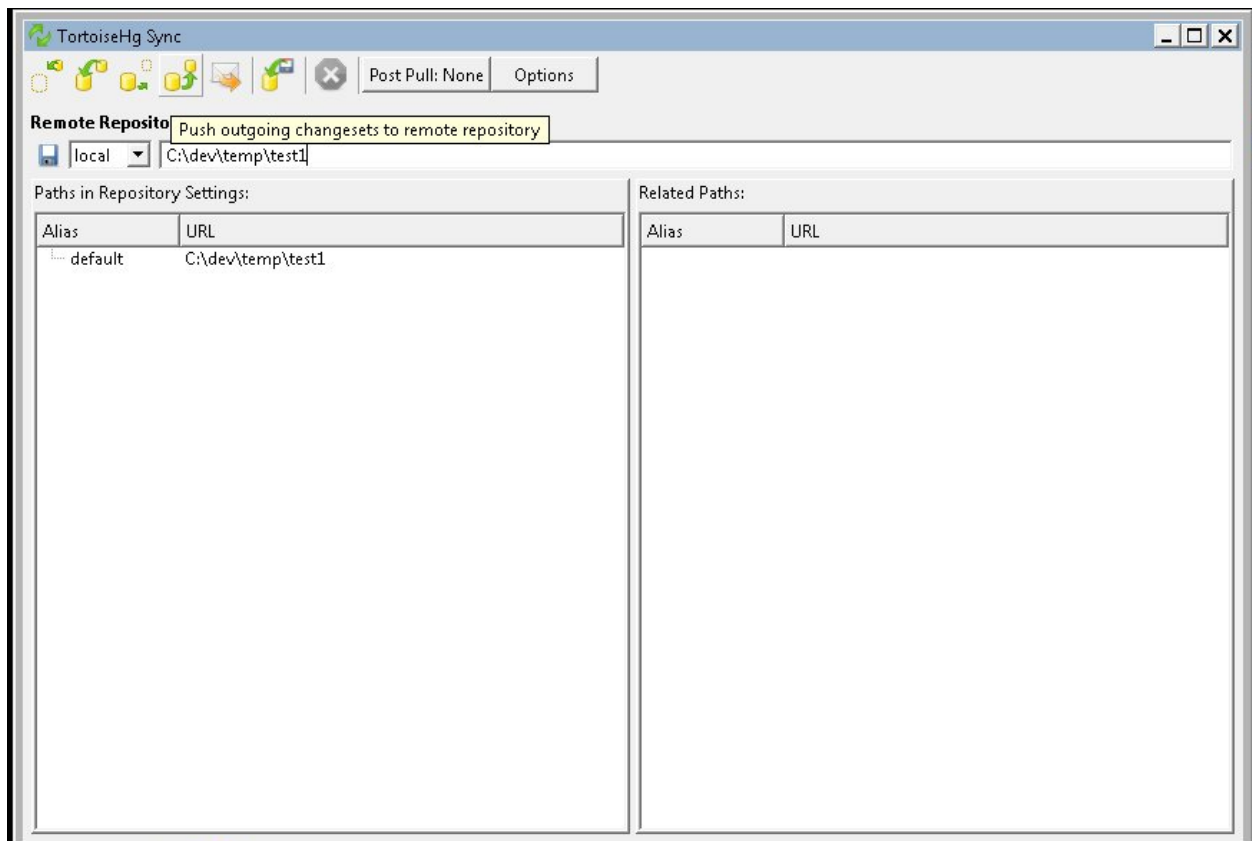


Figure 30: Synchronize Dialog

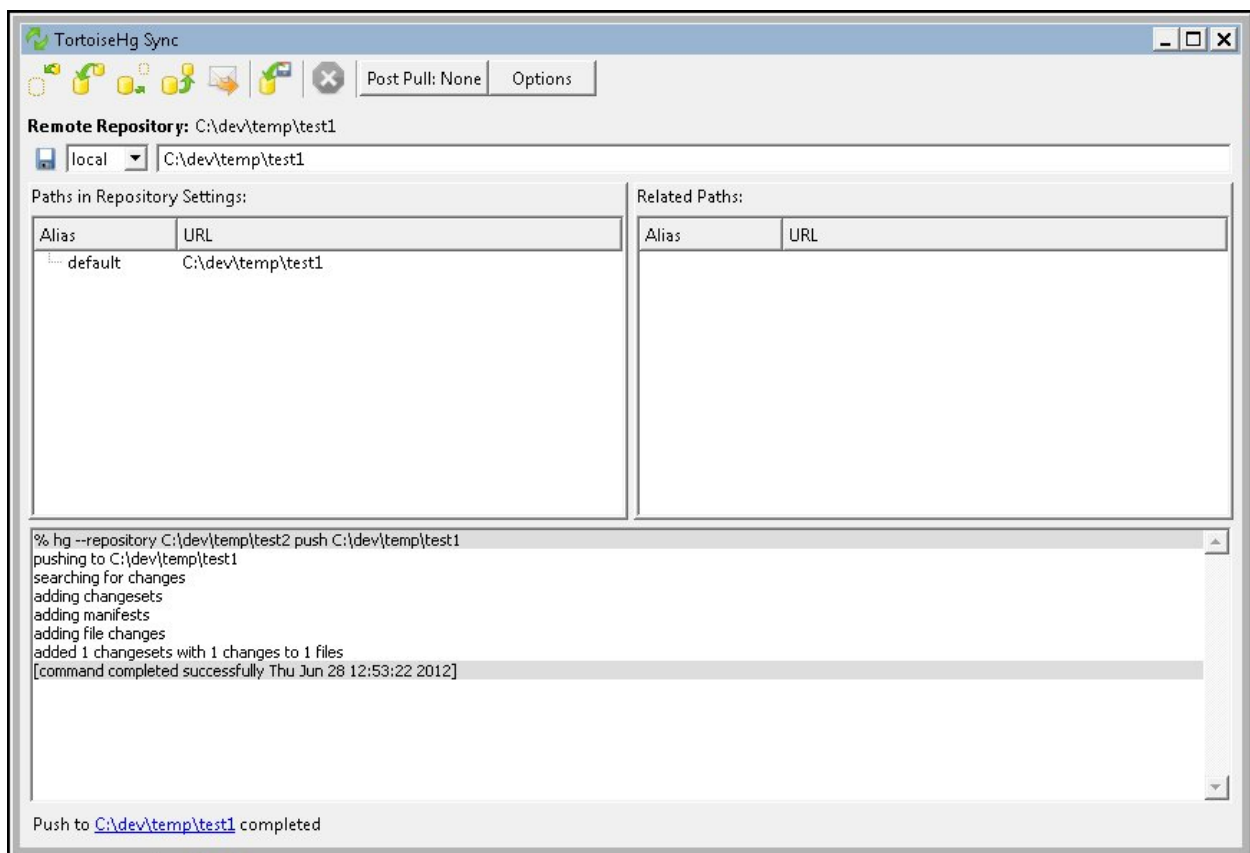


Figure 31: Dialog after push

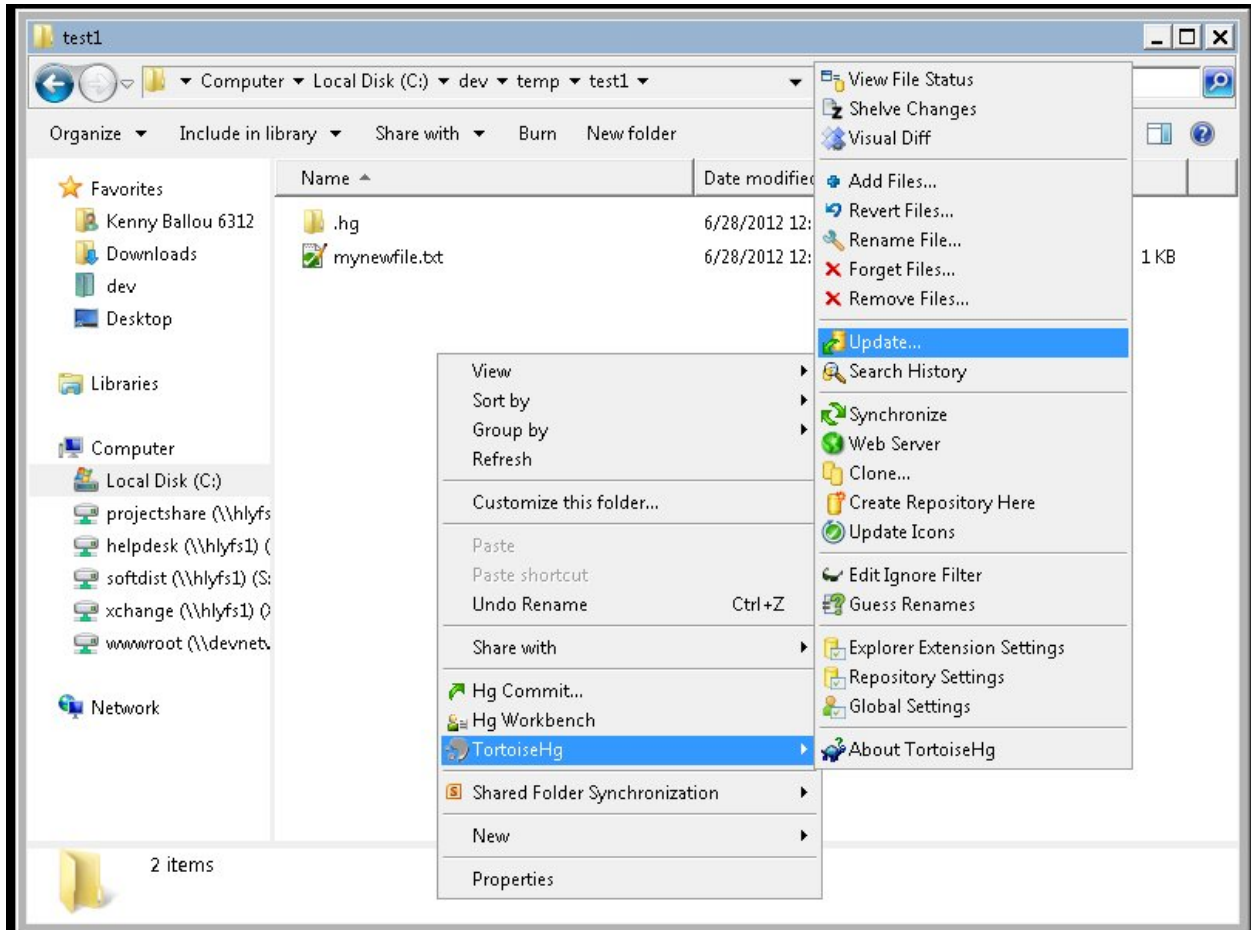


Figure 32: Update receiving repository