mercurial

Tassilo Horn

22 January 2011

Contents

1 Mercurial for IST-Dummies

1.1 Installing Mercurial

The current release of Mercurial for any common operationg system can be fetched at http://mercurial.selenic.com/downloads/. That includes the Windows TortoiseHG GUI.

On UNIX derivates like Linux or BSD, you might want to use your package manager:

```
# Debian/Ubuntu
$ apt-get install mercurial
# Fedora
$ yum install mercurial
```

```
# Gentoo
$ emerge mercurial
# FreeBSD
$ cd /usr/ports/devel/mercurial
$ make install
# Solaris 11 Express
$ pkg install SUNWmercurial
```

Some distributions like Debian or Ubuntu LTS ship pretty old mercurial versions. So you might want to install one of the some backports repository (e.g. "ppa:mercurial-ppa/releases").

For **Eclipse**, there's HG Eclipse plugin at http://javaforge.com/project/HGE.

1.2 Getting Help

Mercurial is pretty good documented. There's a very good guide at http://mercurial.selenic.com/guide/. Addinitonally, mercurial has an online help system.

```
hg help        # List all topics
hg help <cmd>  # Show help for <topic>
```

Furthermore, the people on `#mercurial` on irc.freenode.net are very helpful. And there's the mailinglist that can be accessed as newsgroup via Gmane (gmane.comp.version-control.mercurial.general).

1.3 Mercurial Configuration Files

Mercurial basically has 3 config files:

- /etc/mercurial/hgrc (UNIX) / C:/mercurial/mercurial.ini (Windows)

  Applies to all users on that machine. Since we all have our own computers, we'll simply ignore those.

- ~/.hgrc (UNIX) / %USERPROFILE%/.hgrc or %HOME%/.hgrc (Windows)

  User-specific configurations.

- <repo>/.hg/hgrc (UNIX and Windows)

  Repository-specific configurations.

1.4 Basic Configurations

These configurations might go into ~/.hgrc.

Optionally, you can tell mercurial who your are, like so:

```
[ui]
username = Tassilo Horn <horn@uni-koblenz.de>
```

So any commit I do will use that name and mail address in the log. But **warning**, do that only if you want to use that name/mail each and every project. If you participate in other projects using mercurial, it is better to specify that on a per-repository basis in `<repo>/.hg/hgrc`.

Now, let's tell mercurial what **editor** it should fire up for writing commit messages.

```
[ui]
editor = emacsclient     # Use a running emacs instance
# editor = vim           # Use VIM
# editor = ed            # Ed is the standart text editor
                         # http://www.gnu.org/fun/jokes/ed.msg.html
```

Next, we might want to set up **authentication**.

```
[auth]
uni.prefix = hg.uni-koblenz.de
uni.username = horn
# uni.password = XXX
uni.schemes = https
```

`uni` is a freely definable prefix. Here we say, that we want to use the username horn for all repositories provided by `hg.uni-koblenz.de`. We can specify the password here, too, but we don't want to do that in plain text. If no password is specified, mercurial will query at every push. With the external Keyring Extension [1], it is possible to save passwords in the system keyring (GNOME Keyring, KWallet, ...), but how to set that up is left as an exercise to the reader.

Next, we might want to activate some nice builtin **extensions**:

```
[extensions]
color =           # Colored diffs
hgk =             # enable the hgk viewer
```

[1] http://mercurial.selenic.com/wiki/KeyringExtension

Finally, we tell mercurial where to find the neccessary **certificates** to authenticate with the university's mercurial server. That is not strictly needed, but else you get a bunch of warnings when pulling/pushing.

```
[web]
cacerts = /etc/ssl/certs/ca-certificates.crt # Either that, if OpenSSL is installe
# cacerts = ~/.uni.crt                       # or point to the uni's certificate.
```

1.5 Getting Started: Cloning a Repository

Ok, now let's really get started. The first thing we have to do is to clone a repository, that is, we create a local repository that contains everything the remote repository contains, and that we'll use for our development.

```
hg clone https://hg.uni-koblenz.de/horn/hgtest
```

After that command finished, we end up with a directory `hgtest` that contains the complete code and history of the mercurial project. We can use it as a test bed. Anyone has read and pull permissions. Feel free to use it for experimenting with mercurial.

1.6 The Basic Update/Edit/Save/Publish Workflow

This is the most basic workflow, corresponding to the usual update/edit/commit cycle known from SVN.

1. Update your local repository with the most recent changes from the university's server.

   ```
   cd hgtest
   hg pull
   ```

   This fetches the latest changesets from the remote repository. But it does not merge those changes into your local checkout, yet. To do so, use:

   ```
   hg update
   ```

   You can do these two steps in one go by using `hg pull -u`.

2. Edit files, add new files with `hg add <file>`, move/rename files with `hg move <old> <new>`, copy with `hg copy <file> <new>`, or delete files with `hg remove <file>`.

   FAQ: How do I add a directory? Answer: You cannot. Mercurial doesn't track directories at all. If you add a file contained in some new directory, then the directory will be added implicitly. When you delete the last file contained in some directory, then the directory will be deleted implicitly, too. If you really feel the need to have an empty directory, add some `.keep` file.

3. Commit your changes locally.

   First, let's check what we've done.

   ```
   hg status
   ```

   This lists all changed and added files. You might want to review the changes you've done using:

   ```
   hg diff           # view all changes since the last commit
   hg diff <file>    # view only changes to <file>
   ```

   If you are satisfied, commit your changes:

   ```
   hg commit
   ```

   This will fire up an editor where you have to specify a commit message. **Important**: It is a good convention (for `hg log` or the web view), if all commit messages start with a one-line summary, followed by as many lines as you want.

4. Merge changes pushed to upstream during your work.

   ```
   hg pull -u
   ```

   If someone pushed some changes in the meantime, you'll get a message like:

   ```
   not updating, since new heads added
   (run 'hg heads' to see heads, 'hg merge' to merge)
   ```

Let's do as it tells us:

```
hg heads
```

Oh, as we can see, our current branch has two heads, because I and someone else added changesets to some revision that was tip before I started my work. So now we have to merge the updstream changes into our changes.

```
hg merge
```

If there are conflicts, resolve them with `hg resolve` (Check `hg help resolve`). If not, commit the merge.

```
hg commit -m "merged upstream changes"
```

Alternative: Instead of merging upstream changes and committing them with some merge commit, it is also possible to do it the git-way by *rebasing* your local commits on top of the newly fetched upstream changesets. However, this edits the history which is a very dangerous business. As a rule of thumb: Rebasing is ok for commits that you have not yet pushed somewhere, because that edits only parts of the history that nobody knows about yet.

Ok, you've been warned. So how do I do that? First, enable the rebase extension by putting

```
[extensions]
rebase =
```

into your ~/.hgrc.

Ok, now assume you did some changes locally and committed them. Again, before pushing, you have to integrate the changes your collegues made in the meantime. So with rebase, you can do:

```
hg pull --rebase
```

That will pull the latest upstream changesets, merge them into your local repository, and *rebase* your local changesets *on top of those new pulled* changes. That is, your changes will come after your collegues changes in the history.

5. Push your changes to the upstream repository.

   You might want to check which commits would be sent by the pull. To do so, use:

   ```
   hg outgoing
   ```

   If that's ok, then push'em.

   ```
   hg push
   ```

   Your changes are now propagated upstreams.

1.7 Branching and Merging

A good overview of the different branching concepts in mercurial can be found at http://stevelosh.com/blog/2009/08/a-guide-to-branching-in-mercurial/.

1.7.1 Named Branches

- Checking For Existing Branches
  To see what branches exist in the repository, use:

  ```
  hg branches
  ```

  To see also branches that have been closed, use:

  ```
  hg branches --closed
  ```

- Switching to an Existing Branch
  To update the working copy to the head of an existing branch, use:

  ```
  hg update <branchname>
  ```

- Creating a New Branch
  To create a new branch, use:

  ```
  hg branch <branchname>
  ```

That also switches to the new branch. Now you can start working (edit, commit).

Note: Branches are (just like in SVN) global, and pushing by default pushes all branches. Mercurial will complain when pushing a new branch. Use `hg push -new-branch` to tell it's ok to do so.

- Merging a Branch
  To merge the changes of the branch `feature1` into the default branch, use:

```
hg update default  # switch to the default branch
hg merge feature1  # merge changes
```

  If there a conflicts, resolve them:

```
hg resolve --list  # List all conflicting files
```

  Fix the problems in the files listed, and mark them as resolved.

```
hg resolve -m <file>  # Resolve file
hg resolve --all      # Resolve all conflicting files
```

  Now commit your resolution changes with

```
hg commit -m "Merged branch feature1"
```

  and you are done.

- Closing a Branch
  Once a branch has been merged back to `default`, you can close it (or you close it before merging). That's done with

```
hg up ~feature1~
hg commit --close-branch -m "closing feature1 branch"
```

  After that, the branch won't be listed in `hg branches`.

1.7.2 Anonymous Branches Using Clones

Say, you want to implement some experimental feature where you are not too sure that it's useful. You think, that you are through with your implementation in one or two days, so you don't want to create a new named branch for that short period.

In that situation, you can simply clone your original repository, implement your stuff on top of `default`, and once you've done, merge the `default` branch from the feature clone back to the `default` branch of your original repository.

```
hg clone jgralab jgralab-exp   # create a feature clone
# work on jgralab-exp...
cd jgralab                     # back to original repository
hg pull ../jgralab-exp         # pull the experimental features
hg merge                       # merge them in the current branch
```

Note: One annoyance with feature clones and Eclipse is that an Eclipse project must have a unique name. That means, that you cannot have both `jgralab` and `jgralab-exp` in your workspace. As a workaround on UNIX, you could use a symlink that either points to `jgralab` or `jgralab-exp`, so you can switch the feature clone by switching the symlink followed by a refresh in Eclipse.

Note: Feature clone is basically a synonym of *fork*, which is the term more commonly used in the git community. For example, when you fork a project on github.com, you basically create a feature clone of that repository.