FREE eBook

LEARNING mercurial

Free unaffiliated eBook created from
Stack Overflow contributors.

#mercurial

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: mercurial

It is an unofficial and free mercurial ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official mercurial.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with mercurial

Remarks

Mercurial is a modern, open-source, distributed version control system. You may have heard of git, which is somewhat more popular today; git and Mercurial are very comparable and offer mostly similar functionality.

Developers use Mercurial to keep track of changes to the source code of their applications. It can be used to keep track of changes to any directory of files, although, like most source control systems, it does best when those files are text files (as opposed to binary files). Mercurial also allows multiple developers to work on the same directory of source code simultaneously, and it manages the process of keeping track of each developer's changes and then merging those changes.

Versions

Mercurial follows a time-based release plan and publishes a major release every three months, in February, May, August, and November. A minor release is published each remaining month. Note that Mercurial does not use sematic versioning, therefore there is no significant difference between 2.9.2 (the last of the 2.9 releases) and 3.0.

Backwards-compatibility is a critical requirement for Mercurial, therefore it's generally safe to simply upgrade your Mercurial installations as needed. It's a good idea however to do so at least once every three months, as some features can render old clients unable to operate on newer repositories.

The projects release notes detail what's changing between versions, while the upgrade notes offer clear guidance on what users need to be aware of when upgrading.

A selection of noteworthy versions:

Version	Notes	Date
3.5	Drops support for Python 2.4 and 2.5	2015-07-31
2.1	Introduces changeset "phases" enabling safe history modification	2012-02-01
1.9	Introduces a Command Server API to support better application integration	2011-07-01
1.7	New repository format supports uncommon filenames	2010-11-01

Examples

Installation and Setup

You can download Mercurial from the project's website, and there are graphical utilities for Windows, Linux and OSX if you'd prefer that to a command line interface. Most Unix package managers include Mercurial, for example on Debian/Ubuntu:

```
$ apt-get install mercurial
```

You can verify Mercurial is installed by running:

```
$ hg --version
```

Setup

Mercurial works out of the box, but you'll likely want to configure Mercurial to know who you are before you go further. To associate a username with your commits edit `~/.hgrc` (or `mercurial.ini` in your home directory on Windows) and add the following lines:

```
[ui]
username = Your Name <your@email.address>
```

If you don't want to do this you can always specify a username when you commit with the `-u` flag, e.g.:

```
$ hg commit -u "Your Name <your@email.address>"
```

Getting Started

See also the Mercurial Tutorial

Creating a Mercurial Repository

A Mercurial repository is simply a directory (referred to as the "working directory") containing an `.hg` directory with metadata about the contents of the repository. This makes Mercurial very lightweight and easy to start using. To create a new repository simply run:

```
$ hg init project
```

Where `project` is the name of the directory you'd like to create. This creates a `project` directory along with a `project/.hg` directory containing the repository itself.

```
$ cd project
$ echo Hello World > hello.txt
```

```
$ hg stat
? hello.txt
```

We just created a `hello.txt` file in the repository and ran `hg status` (or `stat` for short) to see the current status of our repository. As you can see `hello.txt` is annotated with a `?`, meaning Mercurial isn't yet aware of it. The `add` command registers this new file with Mercurial so it will be included in the next commit.

```
$ hg add hello.txt
```

Now that Mercurial is aware of a changed file you can run `diff` to see exactly what's changed since the last commit - in this case we're adding the full contents of `hello.txt`:

```
$ hg diff
diff -r 000000000000 hello.txt
--- /dev/null   Thu Jan 01 00:00:00 1970 +0000
+++ b/hello.txt Sat Jul 23 01:38:44 2016 -0400
@@ -0,0 +1,1 @@
+Hello
```

And once we're happy with them and ready to check in our changes we can run `commit`:

```
$ hg commit -m "Created a hello world file."
```

Note that we included a commit message with `-m` - if you don't specify `-m` Mercurial will launch a text editor you can enter a commit message in. This is useful if you'd like to provide a longer multi-line message.

Once you've committed your changes they no longer show up if you run `hg stat` since the repository is now in sync with the contents of the working directory. You can run `log` to see a list of commits, and `-v` includes additional details like the files each commit touched:

```
$ hg log -v
changeset:   0:b4c06cc77a42
tag:         tip
user:        Michael Diamond@Aodh <dimo414@gmail.com>
date:        Sat Jul 23 01:44:23 2016 -0400
files:       hello.txt
description:
Created a hello world file.
```

Pushing and Pulling

Mercurial makes it easy to share your work, and to pull in contributions from other developers. This involves three key steps; cloning, pulling, and pushing.

Clone

To copy a remote repository to your local disk you "clone" it. To do so simply pass the remote URL

you'd like to clone from. To clone the Mercurial source code simply run:

```
$ hg clone https://selenic.com/hg
```

This creates a local `hg` directory containing a copy of the Mercurial repository you can build, edit, and commit to (though you can't publish your commits back to the parent repository).

Pull

Once you have a repository checked out, you'll want to keep it in sync as others publish changes to it. You can pull down new changes by simply running:

```
$ hg pull
```

This pulls in new commits but doesn't update your working directory, so you won't see any changes immediately. To update the contents of the working directory run:

```
$ hg up
```

Which updates your working directory to the tip (most recent) revision in the repository.

You can also run:

```
$ hg pull -u
```

To pull in new changes and update the working directory in one step.

Push

Assuming you have write-access to the remote repository you can publish any commits you've made locally to the remote repository just as easily with:

```
$ hg push
```

This uploads your changes as long as there haven't been any other commits since the last time you pulled. If your `push` is rejected because it would "create additional heads" that means you need to pull in those new changes and merge them with your own.

```
$ hg pull
$ hg merge  # this creates a new changeset merging your changes with the remote changes
$ hg commit -m "Merged in remote changes"
$ hg push
```

Most of the time this is all you'll have to do since Mercurial handles merging your changes automatically, however sometimes you'll need to resolve merge conflicts manually (see merging topic). If you need to you can always cancel a merge and get back to a clean working directory

with:

```
$ hg up -c
```

But remember this is a destructive operation; any changes in the working directory will be erased.

Branching

When we're first starting our work, we have to decide if this is a separate area of work we're working on, or is this part of an existing line of work. If it's existing, we can work off of that branch. If it's new, we'll start a new branch.

Our workflow then is:

- `hg branch MyNewFeature`
- work work work
- `hg commit -m "committing my changes"`
- work work work
- `hg commit -m "more changes"`

At this point, we want to push our work up to the remote server. But before pushing the changes (ignore this if it is a new branch you haven't pushed before), we need to check if there are any incoming change to this branch. We can check this with:

```
hg incoming -b .
```

If there are any incoming changesets on our branch, we need to do a pull and rebase our changes to the top of the list of changes.

```
hg pull -b . --rebase
```

Once this is done or if there are no incoming changesets, we can proceed with the Push.

We only ever want to push our current work, not everything we've ever done. I really never push my entire repository, but my current line of work. The reasoning is that pushing the entire repository assumes I'm integrating multiple lines of work. But I only want to integrate my current line of work, and I only want to work in one line at a time.

If this is the first time I'm pushing this branch:

```
hg push -b . --new-branch
```

If I've already pushed this branch:

```
hg push -b .
```

The "-b ." command means just push the current branch, and not anything else.

To change between the working branches:

```
hg update myBranchName
```

Read Getting started with mercurial online: https://riptutorial.com/mercurial/topic/2075/getting-started-with-mercurial

Chapter 2: Common operations

Examples

Using the bisect command to find a bug

The `bisect` command helps you to track down the changeset that introduced a bug.

- Reset the bisect state and mark the current revision as bad (it contains the bug!)

```
hg bisect --reset
hg bisect --bad
```

- Go back to a point where you think the bug isn't present

```
hg update -r -200
```

- Now you've to test the software and if your assumption was right (bug not present), mark the revision as good:

```
hg bisect --good
```

    Testing changeset 800:12ab34cd56ef (x changesets remaining, ~y tests)

- Mercurial updates the current revision (somewhere in the middle between the bad and good changeset)
- Test again the software and mark appropriately the current revision. E.g.

```
hg bisect --good
```

    Testing changeset 900:21ba43dc65fe (x changesets remaining, ~y tests)

- ...
- Continue until Mercurial has narrowed the search down to a single changeset:

```
hg bisect --bad
```

    The first bad revision is:

    changeset: 987:1234bad99889

    user: John Doe ____@gmail.com

    date: Jul 28 16:00:00 2016

The `hg bisect` command uses its knowledge of your project's revision history to perform a search in time proportional to the logarithm of the number of changesets to check and has no problems dealing with branches, merges or multiple heads.

Sometimes you have an idea of the incriminated files and you can give an hint to Mercurial:

```
hg bisect --skip "!( file('path:foo') & file('path:bar') )"
```

This skips all revisions that do not touch directories `foo` or `bar`.

Using the revert command to discard unwanted changes.

The `revert` command allows discarding unwanted uncommitted changes.

- Reverting changes to a single file.

```
hg revert example.c
```

- Reverting all changes.

This will discard **all** changes not just the current directory.

```
hg revert --all
```

hg will output which files were reverted.

    reverting example.c

    reverting mydir\example.cpp

    forgetting file.txt

Backup files are produced for discarded changes to previously committed files, in the form `filename.orig`

Read Common operations online: https://riptutorial.com/mercurial/topic/2490/common-operations

Chapter 3: List of commands

Examples

Commands for preparing commits

- **add**: add the specified files on the next commit
- **addremove**: add all new files, delete all missing files
- **backout**: reverse effect of earlier changeset
- **commit, ci**: commit the specified files or all outstanding changes
- **copy, cp**: mark files as copied for the next commit
- **forget**: forget the specified files on the next commit
- **merge**: merge another revision into working directory
- **remove, rm**: remove the specified files on the next commit
- **rename, move, mv**: rename files; equivalent of copy + remove
- **resolve**: redo merges or set/view the merge status of files
- **revert**: restore files to their freshly-checked-out state

Inspecting the history

- **annotate, blame**: show changeset information by line for each file
- **bisect**: subdivision search of changesets
- **cat**: output the current or given revision of files
- **diff**: diff repository (or selected files)
- **grep**: search for a pattern in specified files and revisions
- **log, history**: show revision history of entire repository or files

Exchanging changesets with remote repos

- **archive**: create an unversioned archive of a repository revision
- **bundle**: create a changegroup file
- **clone**: make a copy of an existing repository
- **export**: dump the header and diffs for one or more changesets
- **graft**: copy changes from other branches onto the current branch
- **incoming**: show new changesets found in source
- **import, patch**: import an ordered set of patches
- **init**: create a new repository in the given directory
- **outgoing**: show changesets not found in the destination
- **phase**: set or show the current phase name
- **pull**: pull changes from the specified source
- **push**: push changes to the specified destination
- **recover**: roll back an interrupted transaction
- **rollback**: roll back the last transaction (DANGEROUS) (DEPRECATED)
- **serve**: start stand-alone webserver
- **unbundle**: apply one or more changegroup files

Status: where are you now?

- **bookmarks, bookmark**: create a new bookmark or list existing bookmarks
- **branch**: set or show the current branch name
- **branches**: list repository named branches
- **config, showconfig, debugconfig**: show combined config settings from all hgrc files
- **files**: list tracked files
- **help**: show help for a given topic or a help overview
- **identify, id**: identify the working directory or specified revision
- **incoming, in**: show new changesets found in source
- **locate**: locate files matching specific patterns (DEPRECATED)
- **manifest**: output the current or given revision of the project manifest
- **outgoing, out**: show changesets not found in the destination
- **parents**: show the parents of the working directory or revision (DEPRECATED)
- **paths**: show aliases for remote repositories
- **phase**: set or show the current phase name
- **root**: print the root (top) of the current working directory
- **status, st**: show changed files in the working directory
- **summary, sum**: summarize working directory state
- **tags**: list repository tags
- **tip**: show the tip revision (DEPRECATED)
- **verify**: verify the integrity of the repository
- **version**: output version and copyright information

Workflow: branches, tags, and moving about

- **bookmarks, bookmark**: create a new bookmark or list existing bookmarks
- **branch**: set or show the current branch name
- **tag**: add one or more tags for the current or given revision
- **update, up, checkout, co**: update working directory (or switch revisions)

Read List of commands online: https://riptutorial.com/mercurial/topic/4170/list-of-commands

Chapter 4: Mercurial Queues

Syntax

- hg qnew -m "My commit message" myPatch
- hg qpop
- hg qpush
- hg qrefresh -m "My new commit message"
- hg qapplied
- hg qseries
- hg qfinish
- hg qdelete myPatch
- hg qfold myPatch
- hg qqueue --list
- hg qqueue --create myNewQueue
- hg qqueue --delete myNewQueue

Examples

Enable Extension

Edit **Mercurial.ini** (Windows) or **.hgrc** (Linux/OSX):

```
[extenstions]
mq =
```

Create and Update Patches

Create new patches with `hg qnew patch-name` and then update them with new changes with `hg qrefresh`:

```
hg qnew myFirstPatch  // Creates a new patch called "myFirstPatch"
...                   // Edit some files
hg qrefresh           // Updates "myFirstPatch" with changes
hg qnew mySecondPatch // Creates a new patch called "mySecondPatch" on top of "myFirstPatch"
...                   // Edid some files
hg qrefresh           // Updates "mySecondPatch" with changes
```

Create new patch with commit message:

```
hg qnew -m "My first patch" myFirstPatch
```

Update commit message of current patch:

```
hg qrefresh -m "My new message"
```

Update commit message (multiline) of current patch:

```
hg qrefresh -e
```

Push and Pop Patches

Pushing a patch applies the patch to the repository while poping a patch unapplies the patch from the repository.

Pop the current patch off the queue with `hg qpop` and push it back onto the queue with `hg qpush`:

```
hg qpop
hg qpush
```

Pop all patches:

```
hg qpop -a
```

PUsh all patches:

```
hg qpush -a
```

Finish Patches

Finishing patches makes them permanent changesets.

Finish **first** applied patch in the queue:

```
hg qfinish
```

Finish **all** applied patches in the queue:

```
hg qfinish -a
```

Rebase

To rebase with latest changesets in upstream repository:

```
hg qpop -a    // Pop all patches
hg pull -u    // Pull in changesets from upstream repo and update
hg qpush -a   // Push all patches on top of new changesets
```

If there are any conflicts you will be forced to merge your patches.

Fold/Combine Patches

To fold (combine) two patches:

```
hg qnew firstPatch   // Create first patch
hg qnew secondPatch  // Create second patch
hg qpop              // Pop secondPatch
hg qfold secondPatch // Fold secondPatch with firstPatch
```

Multiples Queues

More than one queue may be created. Each queue can be thought of as a separate branch.

```
hg qqueue --create foo // Create a new queue called "foo"
hg qqueue --list       // List all queues
hg qqueue --active     // Print name of active queue
hg qqueue --rename bar // Rename active queue "foo" to "bar"
hg qqueue --delete bar // delete queue "bar"
```

Create two queues and switch between them:

```
hg qqueue --create foo // Create a new queue called "foo" and make it active
hg qqueue --create bar // Create a new queue called "bar" and make it active
hg qqueue foo          // Switch back to queue "foo"
```

Commands

- **qnew**: create a new patch
- **qpop**: pop the current patch off the stack
- **qpush**: push the next patch onto the stack
- **qrefresh**: update the current patch
- **qapplied**: print the patches already applied
- **qseries**: print the entire series file
- **qfinish**: move applied patches into repository history
- **qdelete**: remove patches from queue
- **qdiff**: diff of the current patch and subsequent modifications
- **qclone**: clone main and patch repository at same time
- **qfold**: fold the named patches into the current patch
- **qgoto**: push or pop patches until named patch is at top of stack
- **qguard**: set or print guards for a patch
- **qheader**: print the header of the topmost or specified patch
- **qimport**: import a patch or existing changeset
- **qnext**: print the name of the next pushable patch
- **qprev**: print the name of the preceding applied patch
- **qqueue**: manage multiple patch queues
- **qrecord**: interactively record a new patch
- **qrename**: rename a patch
- **qselect**: set or print quarded patches to push
- **qtop**: print the name of the current patch
- **qunapplied**: print the patches not yet applied

Deprecated:

- **qinit**: init a new queue repository (DEPRECATED)
- **qcommit**: commit changes in the queue repository (DEPRECATED)
- **qsave**: save current queue state (DEPRECATED)
- **qsave**: restore the queue state saved by a revision (DEPRECATED)

Read Mercurial Queues online: https://riptutorial.com/mercurial/topic/5379/mercurial-queues

Credits

S. No	Chapters	Contributors
1	Getting started with mercurial	Community, dimo414, Frank Schmitt, Joel Spolsky, Tom, Vivek Vijayan
2	Common operations	manlio, Tom
3	List of commands	Esteis, jenglert
4	Mercurial Queues	jenglert, mcarlin