hg-review Documentation

Release pre-alpha

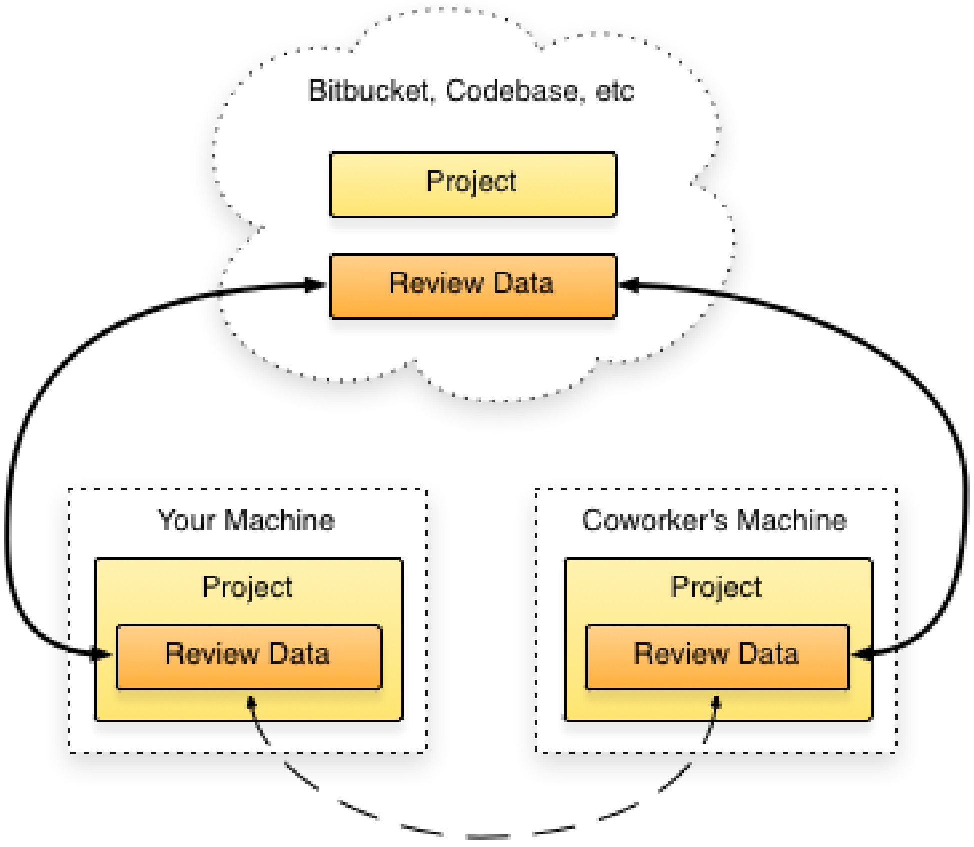
Steve Losh

Nov 16, 2017

Contents

hg-review is a Mercurial extension for performing distributed code review.

“Distributed code review” means you can do cool stuff like this:



CHAPTER 1

Quickstart

If you're impatient and want to play with hg-review right away, here's what you need to do.

First, clone the extension somewhere:

```
hg clone http://bitbucket.org/sjl/hg-review/
```

Then add it to your `~/.hgrc` file:

```
[extensions]
review = [path to]/hg-review/review/
```

Now you need a repository that has code review enabled. Luckily, you've already got one – hg-review uses itself for code review.

`cd` into the directory you cloned hg-review to and start the web interface:

```
cd hg-review
hg review --web
```

Open http://localhost:8080/ in your browser of choice and poke around. Check out the *Overview* when you're ready to learn more.

CHAPTER 2

User's Guide

If you want to use hg-review for anything more than some simple poking around, this is the place to start.

2.1 Overview

Let's get started using hg-review. No matter how you want to use it, you need to install it first.

2.1.1 Installation

hg-review requires Python 2.5 or later and Mercurial 1.6 or later.

You probably have both of these requirements already, but if you encounter problems you might want to check these first with `python --version` and `hg --version`.

hg-review also depends on a couple of other things like Flask and Jinja2, but it bundles these requirements so you don't need to worry about them.

To install hg-review, first clone the extension somewhere:

```
hg clone http://bitbucket.org/sjl/hg-review/
```

Then add it to your `~/.hgrc` file:

```
[extensions]
review = [path to]/hg-review/review/
```

TortoiseHG

People using TortoiseHG on Windows platforms need to update the tortoisehg *library.zip*. This is easily done by running the *contrib\windows\update_tortoisehg_libs.py* script.

Do to that, you need to have python 2.7 installed:

```
python contrib\windows\update_tortoisehg_libs.py
```

2.1.2 Usage

The easiest way to work with hg-review is with the *web interface*. There's also a *command-line interface*, but it's easiest to work with the web interface.

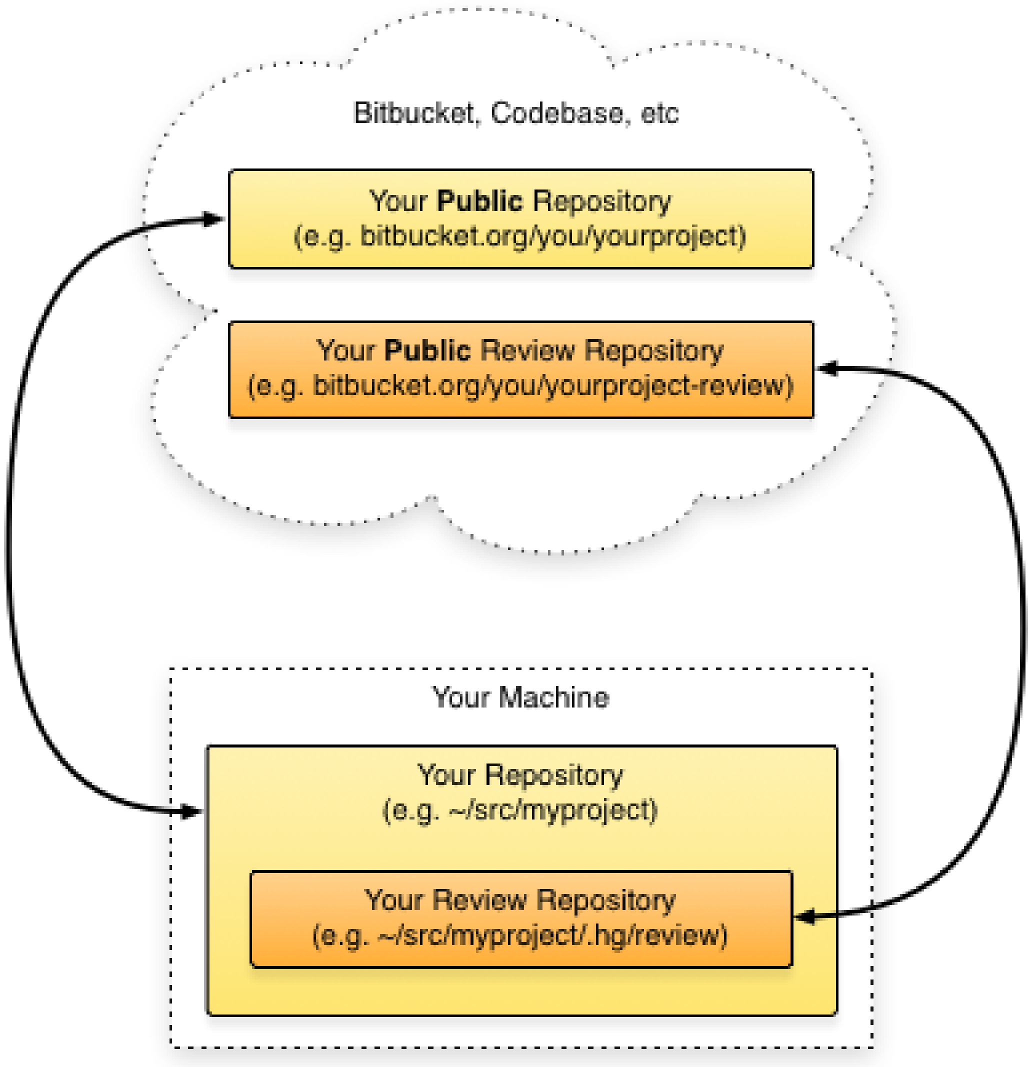
If you want to work with a repository that already has code review set up all you need to do is cd into that repository, and fire up the web ui:

```
cd ~/src/someproject
hg review --web
```

Once that's done you can visit http://localhost:8080/ in your browser to start reviewing.

You should read over the *concepts* documentation to make sure you know how hg-review works and the *web interface* documentation for a quick tour of how to use the web UI.

If you want to *start* using hg-review with a repository, you need to do a few things to get it ready. You'll want to end up with something like this:


Bitbucket, Codebase, etc
Your Public Repository
(e.g. bitbucket.org/you/yourproject)
Your Public Review Repository
(e.g. bitbucket.org/you/yourproject-review)
Your Machine
Your Repository
(e.g. ~/src/myproject)
Your Review Repository
(e.g. ~/src/myproject/.hg/review)

First, create a public repository to hold the code review data. This repository should be in a location that's accessible by anyone that needs to see the review data.

For example, if you're working on an open-source project that's hosted at http://bitbucket.org/you/project/ you should create a new repository for the review data at http://bitbucket.org/you/project-review/

Next you'll need to initialize the review data in your project. `cd` into you project's directory and run:

```
hg review --init --remote-path URL
```

The `URL` should be the *public* URL of the review repo you just created.

This command will create a *local* review data repo for you in `.hg/review`, as well as an `.hgreview` file in your project. You need to commit this `.hgreview` file to your project with the command that hg-review suggested.

Don't worry, this is the only time hg-review will make you commit something to your project's repository and clutter up its changelog.

Now you can get to work reviewing changesets with the web interface by running `hg review --web` in your project.

You should read over the *concepts* documentation to make sure you know how hg-review works, and the *web interface* documentation for a quick tour of how to use the web UI.

2.1.3 Reporting Bugs

If you encounter any errors while using hg-review please post a bug.

2.2 Concepts

You’re not perfect.

Your code is not perfect.

If you’re the only person that’s reading your code, it’s wrong.

As developers we need to review each other’s code. This helps us catch errors before they find our users. It also makes us take greater care when writing code because we know someone will be looking at it.

2.2.1 Code Review Basics

The simplest form of code review is asking a friend to look at the code you just wrote. Often a second set of eyes can find problems you might not have seen, especially if that person has more experience than you.

Unfortunately this isn’t always practical. You might work remotely with people thousands of miles away and not have a chance to simply turn around and say: “Hey, could you look at this?”

Code review tools (like hg-review) exist to make reviewing other people’s code easier.

Their goal is to make it as easy as possible to tell another developer: “No, you did *this* wrong. Fix it.”

2.2.2 Other Code Review Tools

There are a lot of “code review tools” out there.

The primary author of hg-review has a lot of experience with Atlassian Crucible, but some other popular tools include:

- Rietveld
- Reviewboard
- Gerrit
- Code Collaborator

All of these tools try to accomplish the same goal: making it easy for developers to tell each other how to write better code.

hg-review has the same goal, but it goes about it a little differently.

2.2.3 Distributed Code Review

Let’s back up for just a second and talk about version control. Some of the most popular version control systems a few years ago were *centralized* systems like Subversion and CVS.

With these systems you had a central server that contained all the history of your project. You would push changes to this central server and it would store them.

In the past half-decade or so there has been a move toward *decentralized* or *distributed* version control systems. With these systems you commit to your local machine and then *push* and *pull* your commits to other people.

Code review tools, however, seem to have remained rooted in the “centralized server” approach. Even the tools that support decentralized version control systems like git and Mercurial rely on a central server to store the code review data.

hg-review does away with the “centralized data store” model and embraces Mercurial’s distributed nature. Code review data is held in a normal Mercurial repository and can be pushed and pulled like any other type of data.

This has several advantages, the biggest one being that you can review code while offline without sacrificing any functionality.

It also means that the full power of Mercurial (such as tracking history and signing changesets with GPG) can be used on the review data.

2.2.4 Review Data

hg-review tracks two kinds of code review data: comments and signoffs.

Comments are simple comments that people make about changesets. People can comment on:

- A changeset as a whole.
- A specific file within a changeset.
- One or more lines of a specific file within a changeset.

Signoffs, on the other hand, *always* apply to a changeset as a whole. Each person can have one signoff for any particular changeset (though they can edit their signoff later).

Signoffs can be used for whatever purpose your project might find useful, but the author of hg-review recommends that signoffs of “yes” mean:

> I approve of this changeset and think it should make its way to production.

And signoffs of “No” mean:

> I do not approve of this changeset and do not think it should make its way to production without another changeset on top of it that fixes the problems I have listed.

Signoffs of “neutral” might mean:

> This changeset doesn’t really impact me, so I don’t care.

Or perhaps:

> I’ve looked at this code but don’t have the expertise to provide a useful opinion.

2.2.5 Repository Structure

While it’s not necessary to know exactly how the guts of hg-review work, it *is* helpful to understand the basic idea behind it.

Let’s say you have a project with a Mercurial repository in `~/src/yourproject/` and you’d like to start using hg-review with it.

The first thing to understand is that Mercurial stores data about this local repository in `~/src/yourproject/.hg/`, and that data is local to your machine. It is never committed or tracked by Mercurial, but is instead used by the Mercurial program itself to work with your repository.

hg-review creates a *separate* Mercurial repository to keep track of its data. It stores this repository in `~/src/yourproject/.hg/review/`.

Because this is inside of Mercurial's internal `.hg` directory of your project changes to the review data (like comments and signoffs) won't be tracked in your project's repository.

hg-review manages its own data in its own repository to avoid cluttering up your project's log with useless "added a comment"-type commits.

This structure means that you can `cd` into the review data repository itself and interact with it just as you would a normal Mercurial repository. You can `push` and `pull` to and from other people, backout changesets and do anything else you could with a normal Mercurial repository.

2.3 Web Interface

The web interface of hg-review is probably what you're going to use the most.

2.3.1 Running Locally

To start the web interface for a local repository that you want to review you can run `hg review --web`. Visit http://localhost:8080/ to use it.

When you add comments or signoffs hg-review will use your normal Mercurial username as the author.

This command can take a few extra options:

`--address ADDRESS` The address to bind to. Use `0.0.0.0` if you want other people to be able to access it.

> **Be careful!** Because the web interface uses your Mercurial username by default, binding to `0.0.0.0` will let anyone add comments and signoffs in your name! You'll probably want to use the `--read-only` option to prevent this.
>
> Default: `127.0.0.1`

`--port PORT` The port to listen on.

> Default: `8080`

`--read-only` Run the server in read-only mode. This will not allow data to be pushed or pulled, comments to be made or signoffs to be added.

> This can be useful when combined with `--address` to let other people view the UI without letting them add comments in your name.
>
> Default: `false`

`--allow-anon` Allow comments (not not signoffs) to be added even if `--read-only` is used, and set the username to `Anonymous <anonymous@example.com>` instead of your Mercurial username.

> This option is most useful when you're deploying a permanent web interface to a server and want to allow anonymous viewers to add comments. See the *Deployment to a Server* section for more information.
>
> Default: `false`

2.3.2 Deployment to a Server

Although hg-review is built for *distributed* code review it's sometimes nice to provide a public interface. This will let people can comment easily without using the extension (or even cloning your project).

Initial Deployment

You can use any WSGI server you like to provide a public instance of hg-review. Before you start you'll need to have Mercurial installed on your web server.

Once you've got Mercurial running on the server you'll need to clone copies of hg-review, your project, and your project's review data to the web server. First create a directory where everything will live:

```
mkdir /var/www/myproject-review-interface/
cd /var/www/myproject-review-interface/
```

Then grab a copy of hg-review:

```
hg clone http://bitbucket.org/sjl/hg-review/
```

Grab a copy of your project and configure it to use the hg-review extension as well as the built-in fetch extension (to automatically merge updates):

```
hg clone -U http://bitbucket.org/you/yourproject/
cd yourproject

echo '[extensions]' >> .hg/hgrc
echo 'review = /var/www/myproject-review-interface/hg-review/review' >> .hg/hgrc
echo 'fetch = ' >> .hg/hgrc
```

Use hg-review to pull down the review data:

```
hg review --init
```

Now that you've got all the necessary data you can set up the WSGI script. Start by copying the included sample script:

```
cd /var/www/myproject-review-interface/
cp hg-review/contrib/deploy/wsgi.py wsgi.py
```

Edit the script to configure your project to your liking. For reference, the relevant part of the script should look something like this:

```
# An example WSGI script for serving hg-review's web UI.
# Edit as necessary.

# If hg-review is not on your webserver's PYTHONPATH, uncomment the lines
# below and point it at the hg-review directory.
import sys
sys.path.insert(0, "/var/www/myproject-review-interface/hg-review")

REPO = '/var/www/myproject-review-interface/myproject'
READ_ONLY = True
ALLOW_ANON_COMMENTS = False
ANON_USER = 'Anonymous <anonymous@example.com>'
SITE_ROOT = 'http://yoursite.com/optional/path'
TITLE = 'Your Project'
PROJECT_URL = 'http://bitbucket.org/your/project/' # or None
```

All that's left is to point your WSGI server at this script and fire it up. How you do that depends on your WSGI server. A sample configuration file for Gunicorn is provided in `contrib/deploy/gunicorn.conf.py`.

Updating the Data

You'll want to keep the review data for this interface current so users can see all the latest comments and signoffs.

To do this you simply need to pull in the main repository (to receive new changesets in your project) and fetch in the review data repository (to receive new comments and signoffs):

```
hg -R /var/www/myproject-review-interface/ pull
hg -R /var/www/myproject-review-interface/.hg/review fetch
```

New comments and signoffs will be visible immediately – you don't need to restart your WSGI server.

You'll probably want to set this up as a cron job or use a hook of some kind to automate the updates.

If you allow anonymous comments and want people that are using the extension locally (instead of this public instance) to see these comments, you'll need to *fetch and push* the review data repo as well:

```
hg -R /var/www/myproject-review-interface/.hg/review/ fetch
hg -R /var/www/myproject-review-interface/.hg/review/ push
```

hg-review is designed to never encounter merge conflicts with its data, but there's always the chance that someone has done something manually that could cause a problem.

If your interface doesn't seem to be receiving new comments/signoffs you'll want to take a look at the review data repository to see what's wrong:

```
cd /var/www/myproject-review-interface/.hg/review
hg heads
```

There should only ever be one head in this repository. If there are more you'll need to merge them (and push back to your public review data repo so others won't encounter the same problem).

2.4 Command Line Interface

hg-review provides a command line interface. Except for initializing the review data, starting the web ui, and possibly some scripting, you'll probably want to use the *web interface* for most tasks.

When you enable the hg-review extension Mercurial will gain a new command: `review`. This command on its own will display review data for a changeset, but it also has several subcommands detailed below.

You can always get help on a given topic right from the command line with `hg help review` or `hg help review-topic`.

2.4.1 `review`

View code review data for a changeset. Usage:

```
hg review [-r REV] [-U CONTEXT] [--quiet] [FILE]
```

Diffs of all changed files will be shown with comments inline.

The line numbers printed are the ones that should be used to add line-level comments.

Options:

`--unified VALUE` The number of lines of context to show for diffs in this changeset (default: `5`).

`--rev VALUE` The revision to show (default: `.`).

--quiet Do not show diffs – only show review-level comments and signoffs (default: `false`).

--verbose Show the short identifier of each comment and signoff, mainly for use with the *edit* subcommand (default: `false`).

--debug Show the full identifier of each comment and signoff, mainly for use with the *edit* subcommand (default: `false`).

2.4.2 --init

Initialize code review for a repository. Usage:

```
hg review --init --remote-path PATH
```

When run for the first time in a project, it will do two things:

- Create a new repository to hold the review data at `.hg/review/`.
- Create and `hg add` a `.hgreview` file in the current repository. You will need to commit this file yourself with: `hg commmit .hgreview -m 'initialize code review data'`

The `--remote-path` option is required and specifies the path where the canonical code review data for this project will live. This is the path that will be cloned when someone else runs `hg review --init` on the project.

Options:

--remote-path VALUE The URL to the public code review data repository.

2.4.3 --comment

Add a code review comment for a changeset. Usage:

```
hg review --comment [-m MESSAGE] [--mdown] [-r REV] [-l LINES] [FILE]
```

If no files are given the comment will be attached to the changeset as a whole.

If one or more files are given but no lines are given, the comment will be attached to each file as a whole.

If a file is given and lines are given the comment will be attached to those specific lines. Lines should be specified as a comma-separated list of line numbers (as numbered in the output of "hg review"), such as `3` or `2,3`.

Options:

--rev VALUE The revision to add a comment to (default: `.`).

--lines VALUE Comment on the given lines (specified as a comma-separated list of line numbers) of the file (default: `None`).

--message VALUE Use `VALUE` as the comment instead of opening an editor (default: `None` (i.e. "open an editor")).

--mdown Use Markdown to format the comment (default: `False`).

2.4.4 --signoff

Add a code review signoff for a changeset. Usage:

```
hg review --signoff [-m MESSAGE] [--mdown] [--yes | --no] [-r REV]
```

The `--yes` and `--no` options can be used to indicate whether you think the changeset is "good" or "bad".

It's up to the collaborators of each individual project to decide exactly what that means. If neither option is given the signoff will be marked as "neutral".

Options:

`--rev VALUE` The revision to sign off on (default: `.`).

`--yes` Sign off as "yes" for the changeset (default: `False` (i.e. "neutral")).

`--no` Sign off as "no" for the changeset (default: `False` (i.e. "neutral")).

`--message VALUE` Use `VALUE` as the signoff message instead of opening an editor (default: `None` (i.e. "open an editor")).

`--mdown` Use Markdown to format the signoff message (default: `False`).

2.4.5 `--edit`

Edit a comment or signoff. Usage:

```
hg review --edit IDENTIFIER [--yes | --no] [-m MESSAGE] [-l LINES] [--mdown] [FILE]
```

Edit the comment or changeset with the given identifier.

You can find the identifier of the item you would like to edit by running `hg review --verbose` to display identifiers.

Any other options given (such as `--message`, `--yes` or filenames) will replace the content of the item you edit.

`--message VALUE` Replace the comment or signoff message with VALUE (default: `None` (i.e. "open an editor")).

`--mdown` Use Markdown to format the comment or signoff message (default: `False` (i.e. "Use the same formatting the item already has)).

`--lines` The line(s) of the file to comment on (default: `None` (i.e. "use the same line the comment already has)). Returns an error if you're editing a signoff or a review-level comment.

`--yes` Change the signoff to state the the changeset is "good" (default: `False`). Returns an error if you are not editing a signoff.

`--no` Change the signoff to state the the changeset is "bad" (default: `False`). Returns an error if you are not editing a signoff.

2.4.6 `--check`

Check the review status of a changeset. Usage:

```
hg review --check [-r REV] [--no-nos] [--yeses NUM] [--seen]
```

Check that the given changeset "passes" the given tests of review status. If no tests are given an error is returned.

Tests are checked in the following order:

- `--no-nos`
- `--yeses`
- `--seen`

If any tests fail the command returns a status of 1 with a message describing the failure on stderr, otherwise it returns 0 and prints nothing.

`--rev VALUE` The revision to check (default: `.`).

`--no-nos` Ensure this revision does *not* have any signoffs of "no" (default: `False` (i.e. "Don't perform this check")).

`--yeses VALUE` Ensure this revision has at least `VALUE` signoffs of "yes" (default: `None` (i.e. "Don't perform this check").

`--seen` Ensure this revision has at least one comment or signoff (default: `False` (i.e. "Don't perform this check")).

2.4.7 `--web`

Start the web interface. Usage:

```
hg review --web [--read-only] [--allow-anon] [--address ADDRESS] [--port PORT]
```

Visit http://localhost:8080/ (replace the port number if you specified a different port) in a modern browser of your choice to use the web interface.

Use `Ctrl+C` to stop the interface.

Options:

`--read-only` Make the web interface read-only; disallowing comments, signoffs, pushes and pulls (default: `False`).

`--allow-anon` Allow anonymous comments on the web interface and set the username for comments to an anonymous username (default: `False` (i.e. allow comments and use your Mercurial username)).

`--address VALUE` Run the web interface on the specified address (default: `127.0.0.1`).

`--port VALUE` Run the web interface on the specified port (default: `8080`).

CHAPTER 3

Developer's Guide

If you want to integrate hg-review with your own application or Mercurial extension, or hack on hg-review itself, this is what you need to know.

3.1 API

hg-review takes Mercurial's approach to API stability:

- The command line interface is fairly stable and will not break often.
- File formats will not change often.
- The internal implementation may change frequently – there are no guarantees of stability.

Providing a stable CLI means that (possibly non-GPL) programs can interact with hg-review easily without fear of constant breaking.

Stable file formats mean that older versions of hg-review will be able to work with review data from newer versions (albeit with reduced functionality).

Not providing a stable internal implementation allows hg-review's code to be kept clean and elegant. It means that Python programs will needs to use subprocesses to avoid breaking, but this is a tradeoff that the author feels is worth making.

3.1.1 Data Repository Layout

The structure of hg-review's data repository looks like this:

```
your-project/
|
+-- .hg/
|    |
|    +-- review
|    |    |
|   |   +-- {{ changeset hash }}
|   |   |   |
|   |   |   +-- .exists
|   |   |   |
|   |   |   +-- comments
|   |   |   |   |
|   |   |   |   +-- {{ comment hash }}
|   |   |   |   |
|   |   |   |   `-- other comments...
|   |   |   |
|   |   |   +-- signoffs
|   |   |       |
|   |   |       +-- {{ signoff hash }}
|   |   |       |
|   |   |       `-- other signoffs ...
|   |   |
|   |   `-- other changesets ...
|   |
|   `-- other files ...
|
`-- other files ...
```

All review data for a changeset is stored in:

```
.hg/review/{{ changeset hash }}/
```

A `.exists` file is included in that directory when code review for that changeset is initialized. This allows us to check if a given changeset has been initialized for code review very quickly.

Comments for a changeset are stored in:

```
.hg/review/{{ changeset hash }}/comments/{{ comment hash }}
```

Signoffs for a changeset are stored in:

```
.hg/review/{{ changeset hash }}/signoffs/{{ signoff hash }}
```

3.1.2 File Formats

hg-review's file format is (fairly) stable and is designed to be easily parsed to enable export to other code review systems.

Comment and signoff files are stored as JSON. The files are indented four spaces per level to make them more human-readable.

`.exists` Files

The `.exists` file is always empty. It simply exists to make looking up whether a given changeset has been initialized faster. It may go away in the future – do not depend on it.

Comment Files

Here is a sample comment file:

```
{
    "author": "Steve Losh <steve@stevelosh.com>",
    "file": [
        "reykjavi\u0301k.txt",
        "cmV5YWphdmnMgWsudHh0"
    ],
    "hgdate": "Mon Jul 12 23:55:51 2010 -0400",
    "lines": [
        0
    ],
    "message": "Sample.",
    "node": "0e987f91e9b6628b26a30c5d00668a15fae8f22f",
    "style": "markdown"
}
```

Comment files have some or all of the following fields:

author The Mercurial username of the person that added this comment.

file A list of two strings. The first string is a (JSON-encoded) representation of the UTF-8 filename. The second string is a base64 encoded version of the actual bytes of the filename (which is what Mercurial gives and expects to receive internally). If this is a review-level comment both strings will be blank.

hgdate The date and time the comment was added (or last edited).

lines A list of integers representing the lines of the file that this comment applies to. If this is a file-level or review-level comment the list will be empty.

message A string representing the raw comment message.

node A string representing the hash of the changeset this comment belongs to, for easy lookup later.

style A string representing the style of this comment – this will be `markdown` for Markdown comments and blank for plain-text comments. More styles may be added in the future.

Signoff Files

Here is a sample signoff file:

```
{
    "author": "Steve Losh <steve@stevelosh.com>",
    "hgdate": "Tue Jul 13 00:16:00 2010 -0400",
    "message": "Sample.",
    "node": "0e987f91e9b6628b26a30c5d00668a15fae8f22f",
    "opinion": "yes",
    "style": "markdown"
}
```

Signoff files have some or all of the following fields:

author The Mercurial username of the person that added this comment.

hgdate The date and time the comment was added (or last edited).

message A string representing the raw comment message.

node A string representing the hash of the changeset this comment belongs to, for easy lookup later.

opinion A string representing the signoff opinion. This will be `yes`, `no`, or a blank string (for a neutral signoff).

style A string representing the style of this comment – this will be `markdown` for Markdown comments and blank for plain-text comments. More styles may be added in the future.

3.1.3 Command Line Interface

hg-review's command line interface is (fairly) stable. If you want to interact with review data for a repository this is the safest method to use.

See the *command line interface documentation* for more details.

3.1.4 Internal Python API

hg-review's internal Python implementation is *not* stable. It may change at any time. Relying on it virtually guarantees your application will break at some point.

For a more stable API you should use the command line interface.

The Python API will be documented later, but is not a high priority at the moment because of its volatility.

3.2 Hacking hg-review

Want to improve hg-review? Great!

The easiest way is to make some changes, push them somewhere public and send a pull request on Bitbucket (or email Steve).

3.2.1 Rough Guidelines

Here's a few tips that will make hg-review's maintainer happier.

Basic Coding Style

Keep lines of code under 85 characters, unless it makes things *really* ugly.

Indentation is four spaces. Tabs are evil.

Commit Messages

Commit messages should start with a line like:

```
api: add feature X
```

The first part is the component the change affects, like `api`, `cli`, `web`, `docs/api`, or `guts`.

`guts` is a catchall for changesets that affect everything at once – using it means that the changeset could probably be split up into separate smallet changesets.

The rest of the commit message should describe the change.

Tests

Update the tests *in the same changeset as your change*. This makes bisection by running the test suite easier.

If your changeset changes the CLI output, make sure you've read the next section and then add a test for it *in the same changeset*.

If your changeset adds a new feature, add a test for it *in the same changeset*.

If your changeset fixes a bug, add a test that would reproduce the bug *in the same changeset*.

Backwards Compatibility

hg-review's internal implementation is not stable. Feel free to modify it however you like. Patches that clean up the code and/or enhance performance will be gladly accepted.

hg-review's file format is stable, but new fields may be added at any time. Removing a field or changing its format is not allowed without a very good reason. Adding an entirely new file format may be acceptable if there is a compelling reason.

hg-review's command line interface is stable. Adding new commands or adding new options to existing commands is fine if they prove useful. Removing commands or radically changing the default output of existing commands is not acceptable except in extreme cases.

hg-review is currently compatible with Python 2.5+ and Mercurial 1.6+. Patches that break this compatibility will be met with a large dose of skepticism.

3.2.2 Layout

hg-review's basic structure looks like this:

```
hg-review/
|
+-- bundled/
|   |
|   `-- ... bundled third-party modules ...
|
+-- contrib/
|   |
|   `-- ... useful items not critical to hg-review's core ...
|
+-- docs/
|   |
|   `-- ... the documentation (and theme) ...
|
+-- review/
|   |
|   +-- static/
|   |   |
|   |   `-- ... static media for the web ui ...
|   |
|   +-- templates/
|   |   |
|   |   `-- ... jinja2 templates for the web ui ...
|   |
|   +-- tests/
|   |   |
|   |   ` ... unit test files and accompanying utilities ...
|   |
|   +-- api.py          # the core hg-review backend
|   |
|   +-- cli.py          # the hg-review Mercurial extension CLI
|   |
|   +-- messages.py     # messages used by the CLI
|   |
|   +-- helps.py        # help text for the CLI commands
|   |
|   +-- rutil.py        # useful utilities
|   |
|   `-- web.py          # the web interface
|
+-- README.markdown
|
+-- LICENSE
|
+-- fabfile.py
|
`-- kick.py
```

3.2.3 Testing

hg-review contains a test suite for the command line interface (and therefore the backend API as well).

The tests can be run easily with nose. If you don't have node, you'll need to install it first:

```
pip install nose
```

Once you've got it you can run the suite by cd'ing to the hg-review directory and running `nosetests`.

Before submitting a changeset please make sure it doesn't break any tests.

If your changeset adds a new feature, add a test for it *in the same changeset*.

If your changeset fixes a bug, add a test that would reproduce the bug *in the same changeset*.

3.2.4 Documentation

If you want to submit a patch, please update the documentation to reflect your change (if necessary) *in the same changeset*.

The documentation is formatted as restructured text and built with Sphinx (version 0.6.7).

The CSS for the documentation is written with LessCSS. If you want to update the style you should update the `docs/hgreview/static/review.less` file and render it to CSS. Include the changes to the `.less` file *and* the `.css` file in your changeset.

3.3 Licensing

hg-review is distributed under the same license as Mercurial itself: GPL version 2 or any later version.

If you want to create a program that works with hg-review you should look at Mercurial's License FAQ page to learn about how this might affect you.

The basic idea is:

- If you review code with hg-review, you are not affected by the license.
- If you bundle hg-review with another application and don't change anything, you are not affected by the license.
- If you create an application that interacts with hg-review solely through its command line interface or web interface, you are not affected by the license.
- If you create an application that interactes with hg-review by calling its internal Python API, you *are* affected by the license and will need to license your application's code as GPL version 2 or later.

Note that the last item (using hg-review's internal Python API) is probably the one you *won't* want to do anyway, since the Python API is *not* stable.

If you have any questions please email Steve, but remember that he's not a lawyer and might not have a fast answer for tricky questions.