bugs-everywhere Documentation

Release v2.0.0-rc2 (unknown)

W. Trevor King

Jan 06, 2018

Contents

Bugs Everywhere (BE) is a bugtracker built on distributed version control. It works with Bazaar, Darcs, Git, and Mercurial at the moment, but is easily extensible. It can also function with no VCS at all.

The idea is to package the bug information with the source code, so that bugs can be marked "fixed" in the branches that fix them. Other architectures—such as keeping all the bugs in their own branch—are also possible.

Contents:

CHAPTER 1

Installing BE

1.1 Distribution packages

Some distributions (Debian , Ubuntu , others?) package an old version of BE. If you're running one of those distributions, you can install the package with your regular package manager. For Debian, Ubuntu, and related distros, that's:

```
$ apt-get install bugs-everywhere
```

While, the official packages are not based on this fork, they are compatible.

1.2 Dependencies

Not all of these dependencies are strictly required. See *Minimal installs* for possible shortcuts.

Package	Role	Debian	Gentoo_
Jinja	HTML templating	python-jinja2	dev-python/jinja
CherryPy	serve repos over HTTPS	python-cherrypy3	dev-python/cherrypy
Sphinx	see *Producing this documentation*	python-sphinx	dev-python/sphinx
numpydoc	see *Producing this documentation*	python-numpydoc	dev-python/numpydoc
Docutils	manpage generation	python-docutils	dev-python/docutils

1.3 Git repository

BE is available as a Git repository:

```
$ git clone https://gitlab.com/bugseverywhere/bugseverywhere.git be
```

See the homepage for details. If you do branch the Git repo, you'll need to run:

```
$ make
```

to build some auto-generated files (e.g. `libbe._version`), and:

```
$ make install
```

to install BE. By default BE will install into your home directory, but you can tweak the `INSTALL_OPTIONS` variable in `Makefile` to install to another location. With the default installation, you may need to add `~/.local/bin/` to your `PATH` so that your shell can find the installed `be` script.

1.3.1 Minimal installs

By default, `make` builds both a man page for `be` and the HTML Sphinx documentation (*Producing this documentation*). You can customize the documentation targets (if, for example, you don't want to install Sphinx) by overriding the `DOC` variable. For example, to disable all documentation during a build/install, run:

```
$ make DOC= install
```

Note that `setup.py` (called during `make install`) will install the man page (`doc/man/be.1`) if it exists, so:

```
$ make
$ make DOC= install
```

will build (first `make`) and install (second `make`) the man page.

Also note that there is no need to edit the `Makefile` to change any of its internal variables. You can override them from the command line, as we did for `DOC` above.

Finally, if you want to do the absolute minimum required to install BE locally, you can skip the `Makefile` entirely, and just use `setup.py` directly:

```
$ python setup.py install
```

See:

```
$ python setup.py install --help
```

for a list of installation options.

Jinja is only used by the `html` command, so there's no need to install Jinja if you don't mind avoiding that command. Similarly, CherryPy is only used for the `html` and `serve-*` commands with the `--ssl` option set. The other dependencies are only used for *building these docs*, so feel free to skip them and just use the docs wherever you're currently reading them.

1.4 Release tarballs

For those not interested in the development version, or those who don't want to worry about installing Git, we'll post release tarballs. After you've downloaded the release tarball, unpack it with:

```
$ tar -xzvf be-<VERSION>.tar.gz
```

And install it with::

```
$ cd be-<VERSION>
$ make install
```

CHAPTER 2

Tutorial

2.1 Introduction

Bugs Everywhere (BE) is a bugtracker built on distributed revision control. The idea is to package the bug information with the source code, so that developers working on the code can make appropriate changes to the bug repository as they go. For example, by marking a bug as "fixed" and applying the fixing changes in the same commit. This makes it easy to see what's been going on in a particular branch and helps keep the bug repository in sync with the code.

However, there are some differences compared to centralized bugtrackers. Because bugs and comments can be created by several users in parallel, they have globally unique `IDs` rather than numbers. There is also a developer-friendly *command-line* interface to compliment the user-friendly *web* and *email* interfaces. This tutorial will focus on the command-line interface as the most powerful, and leave the web and email interfaces to other documents.

2.2 Installation

If your distribution packages BE, it will be easiest to use their package. For example, most Debian-based distributions support:

```
$ apt-get install bugs-everywhere
```

See *the install page* for more information and alternative methods.

2.3 Bugs

If you have any problems with BE, you can look for matching bugs:

```
$ be --repo http://bugs.bugseverywhere.org/ list
```

If your bug isn't listed, please open a new bug:

```
$ be --repo http://bugs.bugseverywhere.org/ new 'bug'
Created bug with ID bea/abc
$ be --repo http://bugs.bugseverywhere.org/ comment bea/def
<editor spawned for comments>
```

2.4 Command-line interface

2.4.1 Help

All of the following information elaborates on the command help text, which is stored in the code itself, and therefore more likely to be up to date. You can get a list of commands and topics with:

```
$ be help
```

Or see specific help on `COMMAND` with

    $ be help COMMAND

for example:

```
$ be help init
```

will give help on the `init` command.

2.4.2 Initialization

You're happily coding in your Bazaar / Darcs / Git / Mercurial versioned project and you discover a bug. You think, "Hmm, I'll need a simple way to track these things". This is where BE comes in. One of the benefits of distributed versioning systems is the ease of repository creation, and BE follows this trend. Just type:

```
$ be init
Using <VCS> for revision control.
BE repository initialized.
```

in your project's root directory. This will create a `.be` directory containing the bug repository and notify your VCS so it will be versioned starting with your next commit. See:

```
$ be help init
```

for specific details about where the `.be` directory will end up if you call it from a directory besides your project's root.

Inside the `.be` directory (among other things) there will be a long UUID directory. This is your bug directory. The idea is that you could keep several bug directories in the same repository, using one to track bugs, another to track roadmap issues, etc. See `IDs` for details. For BE itself, the bug directory is `bea86499-824e-4e77-b085-2d581fa9ccab`, which is why all the bug and comment IDs in this tutorial will start with `bea/`.

2.4.3 Creating bugs

Create new bugs with:

```
$ be new <SUMMARY>
```

For example:

```
$ be new 'Missing demuxalizer functionality'
Created bug with ID bea/28f
```

If you are entering a bug reported by another person, take advantage of the `--reporter` option to give them credit:

```
$ be new --reporter 'John Doe <jdoe@example.com>' 'Missing whatsit...'
Created bug with ID bea/81a
```

See `be help new` for more details.

While the bug summary should include the appropriate keywords, it should also be brief. Unlike other bug trackers, the bug itself cannot contain a multi-line description. So you should probably add a comment immediately giving a more elaborate explanation of the problem so that the developer understands what you want and when the bug can be considered fixed.

2.4.4 Commenting on bugs

Bugs are like little mailing lists, and you can comment on the bug itself or previous comments, attach files, etc. For example:

```
$ be comment abc/28f "Thoughts about demuxalizers..."
Created comment with ID abc/28f/97a
$ be comment abc/def/012 "Oops, I forgot to mention..."
Created comment with ID abc/28f/e88
```

Usually comments will be long enough that you'll want to compose them in a text editor, not on the command line itself. Running `be comment` without providing a `COMMENT` argument will try to spawn an editor automatically (using your environment's `VISUAL` or `EDITOR`, see Guide to Unix, Environmental Variables).

You can also pipe the comment body in on stdin, which is especially useful for binary attachments, etc.:

```
$ cat screenshot.png | be comment --content-type image/png bea/28f -
Created comment with ID bea/28f/35d
```

It's polite to insert binary attachments under comments that explain the content and why you're attaching it, so the above should have been:

```
$ be comment bea/28f "Whosit dissapears when you mouse-over whatsit."
Created comment with ID bea/28f/41d
$ cat screenshot.png | be comment --content-type image/png bea/28f/41d -
Created comment with ID bea/28f/35d
```

For more details, see `be help comment`.

2.4.5 Showing bugs

Ok, you understand how to enter bugs, but how do you get that information back out? If you know the ID of the item you're interested in (e.g. bug bea/28f), try:

```
$ be show bea/28f
          ID : 28fb711c-5124-4128-88fe-a88a995fc519
  Short name : bea/28f
    Severity : minor
      Status : open
    Assigned :
    Reporter :
     Creator : ...
     Created : ...
Missing demuxalizer functionality
--------- Comment ---------
Name: bea/28f/97a
From: ...
Date: ...

Thoughts about demuxalizers...
  --------- Comment ---------
  Name: bea/28f/e88
  From: ...
  Date: ...

  Thoughts about demuxalizers...
--------- Comment ---------
Name: bea/28f/41d
From: ...
Date: ...

Whosit dissapears when you mouse-over whatsit.
  --------- Comment ---------
  Name: bea/28f/35d
  From: ...
  Date: ...

  Content type image/png not printable.  Try XML output instead
```

You can also get a single comment body, which is useful for extracting binary attachments:

```
$ be show --only-raw-body bea/28f/35d > screenshot.png
```

There is also an XML output format, which can be useful for emailing entries around, scripting BE, etc.:

```
$ be show --xml bea/35d
<?xml version="1.0" encoding="UTF-8" ?>
<be-xml>
...
```

2.4.6 Listing bugs

If you *don't* know which bug you're interested in, you can query the whole bug directory:

```
$ be list
bea/28f:om: Missing demuxalizer functionality
bea/81a:om: Missing whatsit...
```

There are a whole slew of options for filtering the list of bugs. See `be help list` for details.

2.4.7 Showing changes

Often you will want to see what's going on in another dev's branch or remind yourself what you've been working on recently. All VCSs have some sort of `diff` command that shows what's changed since revision `XYZ`. BE has its own command that formats the bug-repository portion of those changes in an easy-to-understand summary format. To compare your working tree with the last commit:

```
$ be diff
New bugs:
  bea/01c:om: Need command output abstraction for flexible UIs
Modified bugs:
  bea/343:om: Attach tests to bugs
    Changed bug settings:
      creator: None -> W. Trevor King <wking@drexel.edu>
```

Compare with a previous revision `1.1.0`:

```
$ be diff 1.1.0
...
```

The format of revision names passed to `diff` will depend on your VCS. For Git, look to gitrevisions for inspiration.

Compare your BE branch with the trunk:

```
$ be diff --repo http://bugs.bugseverywhere.org/
```

2.4.8 Manipulating bugs

There are several commands that allow to to set bug properties. They are all fairly straightforward, so we will merely point them out here, and refer you to `be help COMMAND` for more details.

- `assign`, Assign an individual or group to fix a bug
- `depend`, Add/remove bug dependencies
- `due`, Set bug due dates
- `status`, Change a bug's status level
- `severity`, Change a bug's severity level
- `tag`, Tag a bug, or search bugs for tags
- `target`, Assorted bug target manipulations and queries

You can also remove bugs you feel are no longer useful with `be remove`, and merge duplicate bugs with `be merge`.

2.4.9 Subscriptions

Since BE bugs act as mini mailing lists, we provide `be subscribe` as a way to manage change notification. You can subscribe to all the changes with:

```
$ be subscribe --types all DIR
```

Subscribe only to bug creaton on bugseverywhere.org with:

```
$ be subscribe --server bugseverywhere.org --types new DIR
```

Subscribe to get all the details about bug `bea/28f`:

```
$ be subscribe --types new bea/28f
```

To unsubscribe, simply repeat the subscription command adding the `--unsubscribe` option, but be aware that it may take some time for these changes to propogate between distributed repositories. If you don't feel confident in your ability to filter email, it's best to only subscribe to the repository for which you have direct write access.

2.4.10 Managing bug directories

`be set` lets you configure a bug directory. You can set

- `active_status` The allowed active bug states and their descriptions.
- `inactive_status` The allowed inactive bug states and their descriptions.
- `severities` The allowed bug severities and their descriptions.
- `target` The current project development target (bug UUID).
- `extra_strings` Space for an array of extra strings. You usually won't bother with this directly.

For example, to set the current target to '1.2.3':

```
$ be set target $(be target --resolve '1.2.3')
```

2.4.11 Import XML

For serializing bug information (e.g. to email to a mailing list), use:

```
$ be show --xml bea/28f > bug.xml
```

This information can be imported into (another) bug directory via

    $ be import-xml bug.xml

Also distributed with BE are some utilities to convert mailboxes into BE-XML (`be-mail-to-xml`) and convert BE-XML into mbox format for reading in your mail client.

2.4.12 Export HTML

To create a static dump of your bug directory, use:

```
$ be html
```

This is a fairly flexible command, see `be help html` for details. It works pretty well as the browsable part of a public interface using the *Email Interface* for interactive access.

2.4.13 BE over HTTP

Besides using BE to work directly with local VCS-based repositories, you can use:

```
$ be serve-storage
```

To serve a repository over HTTP. For example:

```
$ be serve-storage > server.log 2>&1 &
$ be --repo http://localhost:8000 list
```

Of course, be careful about serving over insecure networks, since malicous users could fill your disk with endless bugs, etc. You can disabled write access by using the `--read-only` option, which would make serving on a public network safer.

Serving the storage interface is flexible, but it can be inefficient. For example, a call to `be list` against a remote backend requires all bug information to be transfered over the wire. As a faster alternative, you may want to serve your repository at the command level:

```
$ be serve-commands > server.log 2>&1 &
$ be --server http://localhost:8000 list
```

Take a look at the server logs to get a feel for the bandwidth you're saving! Serving commands over insecure networks is at least as dangerous as serving storage. Take appropriate precautions for your network.

2.4.14 Driving the VCS through BE

Since BE uses internal storage drivers for its various backends, it seemed useful to provide a uniform interface to some of the common functionality. These commands are not intended to replace the usually much more powerful native VCS commands, but to provide an easy means of simple VCS-agnostic scripting for BE user interfaces, etc.

Commit

Currently, we only expose `be commit`, which commits all currently pending changes.

CHAPTER 3

Configuration

3.1 Config file format and location

Most of the information that BE needs lives in the bug repository itself, but there is user-specific information that does not fit into a shared repository. This per-user configuration information is stored in an INI-style config file:

```
[default]
user = 'John Doe <jdoe@example.com>'
```

The config file is located at `~/.config/bugs-everywhere` by default, but you can override the path by setting environment variables (see `path()` for details).

3.2 Settings

Currently the only information stored in the configuration file is a user ID (see `get_user_id()`), as shown in the example above. However, many version control systems allow you to specify your name and email address, and BE will fall back to the VCS-configured values, so you probably don't need to set a BE-specific configuration.

CHAPTER 4

Email Interface

4.1 Overview

The interactive email interface to Bugs Everywhere (BE) attempts to provide a Debian-bug-tracking-system-style interface to a BE repository. Users can mail in bug reports, comments, or control requests, which will be committed to the served repository. Developers can then pull the changes they approve of from the served repository into their other repositories and push updates back onto the served repository.

4.2 Architecture

In order to reduce setup costs, the entire interface can piggyback on an existing email address, although from a security standpoint it's probably best to create a dedicated user. Incoming email is filtered by procmail, with matching emails being piped into `be-handle-mail` for execution.

Once `be-handle-mail` receives the email, the parsing method is selected according to the subject tag that procmail used grab the email in the first place. There are four parsing styles:

Style	Subject
creating bugs	[be-bug:submit] new bug summary
commenting on bugs	[be-bug:<bug-id>] commit message
control	[be-bug] commit message

These are analogous to `submit@bugs.debian.org`, `nnn@bugs.debian.org`, and `control@bugs.debian.org` respectively.

4.3 Creating bugs

This interface creates a bug whose summary is given by the email's post-tag subject. The body of the email must begin with a pseudo-header containing at least the `Version` field. Anything after the pseudo-header and before a line

starting with -- is, if present, attached as the bug's first comment.:

```
From jdoe@example.com Fri Apr 18 12:00:00 2008
From: John Doe <jdoe@example.com>
Date: Fri, 18 Apr 2008 12:00:00 +0000
Content-Type: text/plain; charset=UTF-8
Content-Transfer-Encoding: 8bit
Subject: [be-bug:submit] Need tests for the email interface.

Version: XYZ
Severity: minor

Someone should write up a series of test emails to send into
be-handle-mail so we can test changes quickly without having to
use procmail.

--
Goofy tagline not included.
```

Available pseudo-headers are `Version`, `Reporter`, `Assign`, `Depend`, `Severity`, `Status`, `Tag`, and `Target`.

4.4 Commenting on bugs

This interface appends a comment to the bug specified in the subject tag. The the first non-multipart body is attached with the appropriate content-type. In the case of `text/plain` contents, anything following a line starting with -- is stripped.:

```
From jdoe@example.com Fri Apr 18 12:00:00 2008
From: John Doe <jdoe@example.com>
Date: Fri, 18 Apr 2008 12:00:00 +0000
Content-Type: text/plain; charset=UTF-8
Content-Transfer-Encoding: 8bit
Subject: [be-bug:XYZ] Isolated problem in baz()

Finally tracked it down to the bar() call.  Some sort of
string<->unicode conversion problem.  Solution ideas?

--
Goofy tagline not included.
```

4.5 Controlling bugs

This interface consists of a list of allowed be commands, with one command per line. Blank lines and lines beginning with # are ignored, as well anything following a line starting with --. All the listed commands are executed in order and their output returned. The commands are split into arguments with the POSIX-compliant shlex.split().:

```
From jdoe@example.com Fri Apr 18 12:00:00 2008
From: John Doe <jdoe@example.com>
Date: Fri, 18 Apr 2008 12:00:00 +0000
Content-Type: text/plain; charset=UTF-8
Content-Transfer-Encoding: 8bit
Subject: [be-bug] I'll handle XYZ by release 1.2.3
```

```
assign XYZ "John Doe <jdoe@example.com>"
status XYZ assigned
severity XYZ critical
target XYZ 1.2.3

--
Goofy tagline ignored.
```

4.6 Example emails

Take a look at `interfaces/email/interactive/examples` for some more examples.

4.7 Procmail rules

The file `_procmailrc` as it stands is fairly appropriate for as a dedicated user's `~/.procmailrc`. It forwards matching mail to `be-handle-mail`, which should be installed somewhere in the user's path. All non-matching mail is dumped into `/dev/null`. Everything procmail does will be logged to `~/be-mail/procmail.log`.

If you're piggybacking the interface on top of an existing account, you probably only need to add the `be-handle-mail` stanza to your existing `~/.procmailrc`, since you will still want to receive non-bug emails.

Note that you will probably have to add a:

```
--repo /path/to/served/repository
```

option to the `be-handle-mail` invocation so it knows what repository to serve.

Multiple repositories may be served by the same email address by adding multiple `be-handle-mail` stanzas, each matching a different tag, for example the `[be-bug` portion of the stanza could be `[projectX-bug`, `[projectY-bug`, etc. If you change the base tag, be sure to add a:

```
--tag-base "projectX-bug"
```

or equivalent to your `be-handle-mail` invocation.

4.8 Testing

Send test emails in to `be-handle-mail` with something like:

```
cat examples/blank | ./be-handle-mail -o -l - -a
```

CHAPTER 5

HTTP Interface

BE bundles Cherry-flavored BE, an interactive HTML interface originally developed by Steve Losh.

You can run it from the BE source directory with:

```
$ python interfaces/web/cfbe.py PATH_TO_REPO
```

Eventually we'll move it into `libbe.ui` so it will be installed automatically with every BE installation.

CHAPTER 6

Distributed Bugtracking

6.1 Usage Cases

6.1.1 Case 1: Tracking the status of bugs in remote repo branches

See the discussion in #bea86499-824e-4e77-b085-2d581fa9ccab/12c986be-d19a-4b8b-b1b5-68248ff4d331#. Here, it doesn't matter whether the remote repository is a branch of the local repository, or a completely separate project (e.g. upstream, ...). So long as the remote project provides access via some REPO format, you can use:

```
$ be --repo REPO ...
```

to run your query, or:

```
$ be diff REPO
```

to see the changes between the local and remote repositories.

6.1.2 Case 2: Importing bugs from other repositories

Case 2.1: If the remote repository is a branch of the local repository:

```
$ <VCS> merge <REPO>
```

Case 2.2: If the remote repository is not a branch of the local repository (Hypothetical command):

```
$ be import <REPO> <ID>
```

6.2 Notes

6.2.1 Providing public repositories

e.g. for non-dev users. These are just branches that expose a public interface (HTML, email, ...). Merge and query like any other development branch.

6.2.2 Managing permissions

Many bugtrackers implement some sort of permissions system, and they are certainly required for a central system with diverse user roles. However DVCSs also support the "pull my changes" workflow, where permissions are irrelevant.

CHAPTER 7

Power features

BE comes with a number of additional utilities and features that may be useful to power users. We'll try to keep an up to date list here, but your best bet may be poking around in the source on your own.

7.1 Autocompletion

`misc/completion` contains completion scripts for common shells (if we don't have a completion script for your favorite shell, submit one!). Basic instructions for installing the completion file for a given shell should be given in the completion script comments.

Packagers should install these completion scripts in their system's usual spot (on Gentoo, the Bash completion script should be installed as `/usr/share/bash_completion/be` and Z shell completion script should be installed as `/usr/share/zsh/site-functions/_be`).

7.2 XML-handling utilities

Email threads are quite similar to the bugs/issues that BE tracks. There are a number of useful scripts in `misc/xml` to go back and forth between the two formats using BE's XML format. The commands should be well documented. Use the usual `<command> --help` for more details on a given command.

CHAPTER 8

Hacking BE

8.1 Adding commands

To write a plugin, you simply create a new file in the `libbe/command/` directory. Take a look at one of the simpler plugins (e.g. `libbe.command.remove`) for an example of how that looks, and to start getting a feel for the libbe interface.

See `libbe.command.base` for the definition of the important classes `Option`, `Argument`, `Command`, `InputOutput`, `StorageCallbacks`, and `UserInterface`. You'll be subclassing `Command` for your command, but all those classes will be important.

8.1.1 Command completion

BE implements a general framework to make it easy to support command completion for arbitrary plugins. In order to support this system, any of your completable `Argument` instances (in your command's `.options` or `.args`) should be initialized with some valid completion_callback function. Some common cases are defined in `libbe.command.util`. If you need more flexibility, see `libbe.command.list`'s `--sort` option for an example of extensions via `libbe.command.util.Completer`, or write a custom completion function from scratch.

8.2 Adding user interfaces

Take a look at `libbe.ui.command_line` for an example. Basically you'll need to setup a `UserInterface` instance for running commands. More details to come after I write an HTML UI...

8.3 Testing

Run any tests in your module with:

```
be$ python test.py <python.module.name>
```

for example:

    be$ python test.py libbe.command.merge

For a definition of "any tests", see `test.py`'s `add_module_tests()` function.

Note that you will need to run `make` before testing a clean BE branch to auto-generate required files like `libbe/_version.py`.

8.4 Profiling

Find out which 20 calls take the most cumulative time (time of execution + childrens' times):

```
$ python -m cProfile -o profile be [command] [args]
$ python -c "import pstats; p=pstats.Stats('profile'); p.sort_stats('cumulative').
↪print_stats(20)"
```

If you want to find out who's calling your expensive function (e.g. `libbe.util.subproc.invoke()`), try:

```
$ python -c "import pstats; p=pstats.Stats('profile'); p.sort_stats('cumulative').
↪print_callers(20)"
```

You can also toss:

```
import sys, traceback
print >> sys.stderr, '-'*60, '\n', '\n'.join(traceback.format_stack()[-10:])
```

into the function itself for a depth-first caller list.

For a more top-down approach, try:

```
$ python -c "import pstats; p=pstats.Stats('profile'); p.sort_stats('cumulative').
↪print_callees(20)"
```

CHAPTER 9

Data Format

9.1 Bugdir

target The current project development target.

severities The allowed bug severities and their descriptions.

active_status The allowed active bug states and their descriptions.

inactive_status The allowed inactive bug states and their descriptions.

extra_strings Space for an array of extra strings. Useful for storing state for functionality implemented purely in becommands/<some_function>.py.

9.2 Bug

severity A measure of the bug's importance

status The bug's current status

creator The user who entered the bug into the system

reporter The user who reported the bug

time An RFC 2822 timestamp for bug creation

extra_strings Space for an array of extra strings. Useful for storing state for functionality implemented purely in becommands/<some_function>.py.

comment_root The trunk of the comment tree. We use a dummy root comment by default, because there can be several comment threads rooted on the same parent bug. To simplify comment interaction, we condense these threads into a single thread with a Comment dummy root.

9.3 Comment

Alt-Id Alternate ID for linking imported comments. Internally comments are linked (via In-reply-to) to the parent's UUID. However, these UUIDs are generated internally, so Alt-id is provided as a user-controlled linking target.

Author The author of the comment

In-reply-to UUID for parent comment or bug

Content-type Mime type for comment body

Date An RFC 2822 timestamp for comment creation

body The meat of the comment

extra_strings Space for an array of extra strings. Useful for storing state for functionality implemented purely in becommands/<some_function>.py.

vim: ft=rst

CHAPTER 10

Dealing with spam

In the case that some spam or inappropriate comment makes its way through you interface, you can (sometimes) remove the offending commit `XYZ`.

10.1 If the offending commit is the last commit

bzr	bzr uncommit && bzr revert
darcs	darcs obliterate –last=1
git	git reset –hard HEAD^
hg	hg rollback && hg revert

10.2 If the offending commit is not the last commit

bzr[1]	bzr rebase -r <XYZ+1>..-1 –onto before:XYZ .
darcs	darcs obliterate –matches 'name XYZ'
git	git rebase –onto XYZ~1 XYZ
hg[2]	

10.3 Warnings about changing history

Note that all of these *change the repo history* , so only do this on your interface-specific repo before it interacts with any other repo. Otherwise, you'll have to survive by cherry-picking only the good commits.

[1] Requires the `bzr-rebase plugin'_. Note, you have to increment `XYZ` by hand for `<XYZ+1>`, because `bzr` does not support `after:XYZ`.

[2] From Mercurial, The Definitive Guide:

"Mercurial also does not provide a way to make a file or changeset completely disappear from history, because there is no way to enforce its disappearance"

CHAPTER 11

Producing this documentation

This documentation is written in reStructuredText, and produced using Sphinx and the numpydoc extension. The documentation source should be fairly readable without processing, but you can compile the documentation, you'll need to install Sphinx and numpydoc:

```
$ easy_install Sphinx
$ easy_install numpydoc
```

See the reStructuredText quick reference and the NumPy/SciPy documentation guide for an introduction to the documentation syntax.

11.1 Man page

The man-page source `be.1.txt` is writen in reStructuredText. The `Makefile` converts it to *`roff(7)`* format using Docutils `rst2man`.

The man page should conform to Debian policy.

CHAPTER 12

Indices and tables

- genindex
- modindex
- search