OpenHatch bug importers Documentation

Release dev

OpenHatch contributors

April 29, 2015

Contents

1 About this project 3

2 Installation 5

3 Running the test suite 7

4 License 9

5 Basic concepts 11

6 To add support for a new bug tracker 13

7 The role of multiple BugParsers 15

8 How this integrates with the main OpenHatch site 17

9 Input configuration 19

10 Downloading with scrapy 21

11 On the web: /customs/ 23

12 Roadmap 25

13 Bugs 27

14 Command line interface 29
- 14.1 Configure some Tracker Model data . . . 29
- 14.2 Run the command line interface . . . 29
- 14.3 Flaws . . . 30

15 Indices and tables 31

Contents:

CHAPTER 1

About this project

The OpenHatch bug importers code is a standalone Python package, independent of Django or other web application dependencies, that can download and process information from bug trackers across the web.

Its main intended use is as part of collecting data for the OpenHatch.org "volunteer opportunity finder," but if you find it useful, please go ahead and use it for another purpose! We do cheerfully accept changes.

This package is maintained by the OpenHatch community, so when you want to share code with us, you'll probably want to read the OpenHatch patch contribution guide.

CHAPTER 2

Installation

If you want to use oh-bugimporters as a standalone Python package, which is the recommended way to develop it, you'll need to run the following commands in your command prompt/terminal emulator.

1. **Get the code**:

   ```
   git clone https://github.com/openhatch/oh-bugimporters.git
   ```

2. **Switch into its directory**:

   ```
   cd oh-bugimporters
   ```

3. **Create a virtualenv for the project**:

   ```
   virtualenv env
   ```

   Note: On Debian/Ubuntu systems, you'll need to run "`apt-get install python-virtualenv`" before this will work.

   Note: On **OSX** (tested on 10.8.5), you may need to use the more specific command `virtualenv -p python2.7 env`.

4. **Install compile-time dependences**

   If on Debian or Ubuntu, run:

   ```
   sudo apt-get install -y libxml2-dev
   sudo apt-get build-dep -y python-lxml
   sudo apt-get install -y libffi-dev
   sudo apt-get install -y libssl-dev
   sudo apt-get build-dep -y python-yaml
   ```

   If on Fedora (tested on Fedora 21), run:

   ```
   sudo yum install libxml2-devel
   sudo yum install libxslt-devel
   sudo yum install libffi-devel
   sudo yum install openssl-devel
   ```

   If on MacOS (10.9), a user reported the following way to get started using both brew (a common package manager on Mac) and pip:

   ```
   brew install libxml2
   env/bin/pip install lxml
   ```

   If on Windows, then you might run into some errors. If you get this working there, please let us know what commands make it work on this platform!

5. **Build a working virtualenv**

   Tell the virtualenv we want to "develop" this app, which also has the side-effect of downloading and installing any dependencies.:

   ```
   env/bin/python setup.py develop
   ```

   Note: You may run into errors here if software mirrors are down. If this is the case, you can alternatively run:

   ```
   env/bin/pip install -e .
   ```

6. **Install the test framework**

   This is optional, but hightly recommended, particularly if you want to run the tests:

   ```
   env/bin/pip install -r devrequirements.txt
   ```

CHAPTER 3

Running the test suite

This set of code comes with a set of automated tests that verify the behavior of the code. We like to keep the code in a clean state where all of those tests pass.

You can run them like so:

```
env/bin/py.test
```

You will see a bunch of output that indicates the "pytest" system is looking for, finding, and running tests. Each "." (dot) character indicates a test that passed.

The code generates a coverage.xml file that with information to help understand which parts of the code are "covered" by the test suite. You can read more about code coverage.

CHAPTER 4

License

Right now, the code is under the AGPLv3 license. We should probably, one day, move it to be under the Apache License 2.0 or another more permissive license.

CHAPTER 5

Basic concepts

This Python package has the following basic components:

- BugImporter classes, which *download* bug data from remote bug trackers.
- BugParser classes, which *process* bug data after the download and normalize it into simple items.
- The ParsedBug object (in bugimporters/items.py), which is what all bug data from the 'net eventually becomes.

The BugImporter class has a bunch of machinery for doing the downloading in parallel. However, the development community around this project thinks that we should remove that machinery and switch to depending on "Scrapy" and using that for downloading.

In the bugimporters/ directory, you will find one Python file per different type of bug tracker supported by this codebase, along with some helper files.

CHAPTER 6

To add support for a new bug tracker

Generally, every bug tracker supported by this codebase must provide:

- A subclass of BugImporter, and
- A subclass of BugParser.

The difference is that the BugImporter subclass is designed to accept a list of bug numbers and perform a bunch of HTTP requests to download information about the bug. In this sense, a BugImporter is aware of the network. BugParser objects are unaware of the network.

Generally, one usually only needs a single BugParser and BugImporter subclass per *type* of bug tracker that is supported. For example, bugimporters/github.py contains one BugImporter subclass that manages the downloading of data via the Github API, and it contains one BugParser subclass that converts data from that API into instances of bugimporters.items.ParsedBug, massaging data as necessary.

(Note that it is possible to write a BugImporter that generates the ParsedBug objects without a BugParser... in theory. We don't recommend doing things this way, but bugimporters/google.py is an example of one.)

CHAPTER 7

The role of multiple BugParsers

A BugImporter, by default, uses one particular BugParser to process bug data. For example, the Bugzilla bug importer has a generic Bugzilla parser that processes the XML data that Bugzilla returns.

The Bugzilla bug importer is an example of a BugImporter that can work with any of a few different BugParser subclasses. You can see those in bugimporters/bugzilla.py.

This is usually helpful when a specific open source community uses its bug tracker in some unusual way, and therefore special code is required to massage the data into the format of a bugimporters.items.ParsedBug. (For an example, see bugimporters/bugzilla.py and the KDEBugzilla class – in particular, the generate_bug_project_name() method. This method exists because the KDE communities names projects in ways that we want to smooth out for consumers of the data, such as the OpenHatch website.)

If you want to add a new custom BugParser, here is what you would do:

- Find the file corresponding to the bug tracker *type* you're adding a custom bug parser for. For example, if you're adding support for a special Bugzilla instance, open up bugimporters/bugzilla.py in your favorite text editor.
- Add a new subclass of BugParser at the bottom of that file, probably overriding the extract_tracker_specific_data method. Make sure to subclass from the specific version of BugParser to the kind of bug tracker you're modifying; for example, if you are adding custom code for a special Bugzilla withi bugimporters/bugzilla.py, your new class should be a subclass of BugzillaBugParser.
- Write a test. For now, this package only has tests covering the Trac bug importers and parsers. If you're adding a new bug parser for Trac, simply:
  - Copy the test_bug_parser() into a new method
  - Change the sample data, and the assertions, for the behavior you need.
  - Run the new test. Make sure it fails.
  - Now, write a new BugParser subclass that impements the behavior you need.
  - Make sure the test passes. (Then submit it for review and inclusion!)

By focusing on this test-driven workflow, you are sure that the code you add is required and correct.

CHAPTER 8

How this integrates with the main OpenHatch site

In this section of the oh-bugimporters documentation, we discuss how the OpenHatch web app integrates with this "oh-bugimporters" project. (If you want to integrate your own project with oh-bugimporters, you can use this to understand the architecture.)

To understand that, we'll go through a few elements at a time.

CHAPTER 9

Input configuration

In order to run oh-bugimporters and actually download bugs, you must configure a list of bug trackers that you want to pull data from.

The file should be a YAML file.

You can use a sample configuration file bundled in examples/sample_configuration.yaml.

CHAPTER 10

Downloading with scrapy

The process of doing the actual downloading is done using the "scrapy" command. Scrapy is a framework for running web crawlers, and you can use it to run the bug importers.

If you have a virtualenv in which you have run "setup.py develop" for this code in env/, the following command will run a scrapy-based import:

```
env/bin/scrapy runspider bugimporters/main.py  -a input_filename=/tmp/input-configuration.yaml  -s FE
```

Note that you must have a configuration file at /tmp/input-configuration.yaml for this command to work. If you need a sample configuration file, copy it out of examples/ as described above in the "Input configuration" section.

CHAPTER 11

On the web: /customs/

Within the OpenHatch code, "customs" is the name for data "import" and "export." (It's a pun.)

On the live OpenHatch website, there is a small bit of web code for letting site administrators manage the list of bug trackers that we download data from. This lives at https://openhatch.org/customs/

(A note about security: At the moment, there is no authorization; any logged-in user of the OpenHatch site can visit the /customs/ management interface and change this configuration. Right now, we consider this a good idea because it makes using the site a smooth process – whenever a project maintainer wants to add their project, they don't need to wait to receive permission.)

The "Tracker type" drop-down is generated from data stored in mysite/customs/core_bugimporters.py – in particular, the all_trackers object defined at the top of the file. As you choose options in the drop-down, JavaScript on the page automatically submits the form and shows you the list of bug trackers stored in the database that correspond to that type of tracker.

By adjusting the information configured in this interface, project maintainers alter the contents of the OpenHatch database (via models and forms in mysite/customs/).

Once per day, the live OpenHatch site exports its data into an input configuration file, and then it runs scrapy to actually download data from the bug trackers in question.

CHAPTER 12

Roadmap

Right now, this package has the following functionality:

- Asynchronous bug fetching
- Pluggable support for new bug tracker types, and for special-cased BugParser objects

For the next release, I would expect the following goals:

- Very high coverage of the bugimporters code from within the test suite. (Note that most of the code actually *is* tested, but the tests haven't been moved over from oh-mainline.)
- Documentation on how to visualize the coverage.xml file from something other than Jenkins. (Perhaps there's an HTML report we can generate.)

For releases after that:

- Contacting the contributors and getting them to agree to the Apache License 2.0 for this code (or at least not AGPLv3; perhaps GPLv3 or LGPLv3; but my vote is for Apache License 2.0).
- Adding support for other bug tracking backends, such as sourceforge.net's Allura, and the older sourceforge.net tracker.
- Fixing the "old_trac" support to work again. (In the past, we relied on a Django model called TracBugTimes that stored the content of some RSS feeds. In oh-bugimporters, we can instead cache those RSS feeds on the filesystem somewhere, and thereby stop using the database as a cache.)
- Refactoring to use scrapy to manage the crawl, so we can delete all our messy async download management code.
- Documentation describing how to create a simple Python dict describing a bug tracker, and pass that through the machinery to get a dump of that bug tracker's bug data. (After the move to Scrapy, this should be fairly easy.)

CHAPTER 13

Bugs

If you want to file specific bugs against this package, use the main OpenHatch bug tracker. Please add the "bugimporters" tag.

Relevant links:

- OpenHatch bug tracker
- List of open bugs tagged as bugimporters

CHAPTER 14

Command line interface

The oh-bugimporters package has the capability to crawl remote bug trackers and store the parsed data in YAML files. To use this functionality, you must create a YAML file with information about the remote bug trackers you want to crawl.

This document steps you through that. The command line interface is really just a way to call Scrapy.

In order to do a bug crawl, you'll need to follow these steps:

14.1 Configure some Tracker Model data

Create a YAML file in, for example, /tmp/configuration.yaml, that is a list of dictionaries.

The dictionaries must have the following keys:

- tracker_name (string)
- base_url (string)

The following key is optional, and if present, is used when annotating the the bug data with the project name. By default, this is the same as the tracker_name.

- bug_project_name_format (string)

The following keys are optional, and are used during bug data processing to annotate the bug data with information like if the bug is good for first-time contributors, or if the bug is oriented entirely around documentation.

- bitesized_type (string)
- bitesized_text (string)
- documentation_type (string)
- documentation_text (string)

A sample valid yaml file can be found in examples/sample_configuration.yaml.

14.2 Run the command line interface

Run this command:

```
./env/bin/python bugimporters/main.py -i /tmp/configuration.yaml -o /tmp/output.jsonlines
```

This will read the configuration YAML file you have named, and go off and download bugs. When it exits, /tmp/output.jsonlines will have the parsed bug data.

Note that the output is always in Scrapy's "jsonlines" format. You can read more about jsonlines here.

14.3 Flaws

Flaws at the moment:

- We need to modify the output format to support requesting deletion of data for a bug.

CHAPTER 15

Indices and tables

- *genindex*
- *modindex*
- *search*