hamster-lib Documentation

Release 0.11.1

Eric Goller

Jul 06, 2016

Contents

Contents:

CHAPTER 1

hamsterlib

(A badges refer to `master`)

A library for common timetracking functionality.

`hamster-lib` aims to be a replacement for `projecthamster` backend library. While we are not able to function as a straight forward drop-in replacement we try very hard to stay as compatible as possible. As a consequence clients are able to switch to `hamster-lib` merely by changing some basic calls. Most of the semantics and return values will be as before.

This itself points to a major architectural shift in the way `hamster-lib` approaches timetracking. We are firm believers in *do one thing, and do it well*. The tried and tested unix toolbox principle. As such we focus on providing useful backend functionality and helper methods so clients (dbus interfaces, CLIs or graphical UIs) can build upon a solid and consistent base and focus on their specific requirements.

1.1 Features

- Full python >=2.7 and >=3.4 compatibility
- Full unicode support
- >= 95% test coverage
- Extensive documentation
- Focus on clean, maintainable code.
- Free software: GPL3
- All you need for production, test or dev environments comes out of the box with regular python tools.

1.2 First Steps

- Build dev environment: `make develop`
- Build the documentation locally: `make docs`
- Run just the tests: `make test`
- Run entire test suite including linters and coverage: `make test-all`

1.3 Additional Resources

- Documentation by 'read the docs'
- CI thanks to Travis-CI
- Coverage reports by 'codecov'
- Dependency monitoring by 'requires.io'

1.4 News: Version 0.11.0

This is the first release of `hamster-lib` as official part of projecthamster. As such it includes a lot of internal adjustments and minor fixes. Besides such housekeeping however, is also offers some genuine new features. You can now query `ActivityManaget.get_all` to return *all* activities, where it previously only returned *all for given category*. We also made `Category` , `Activity` and `Fact` hashable, so you can now use them as `dict` keys or `set` elements. For a more detailed overview about what new, please refer to the changelog. Happy tracking; Eric.

1.5 Todo

This early release is mainly meant as a rough proof-of-concept at this stage. It showcases our API as well as our general design decisions. As such there are a few functionalities/details of the original `projecthamster` backend that we wish to allow for, but are not provided so far. These are:

- Tags (We accept them but they are not stored in the backend.)
- Autocomplete related methods
- Trophies (The jury is still out on if and how we want to support those.)
- Migrations from old databases.

1.6 Incompatibilities

Despite our efforts to stay backwards compatible we did deliberately break the way `Facts` without end dates are handled. We think allowing for them in any persistent backend creates a data consistency nightmare and so far there seems no conceivable use case for it, let alone an obvious semantic. What we do allow for is *one* `ongoing fact` . That is a fact that has a start, but no end date. For details, please refer to the documentation.

1.7 Credits

Tools used in rendering this package:

- Cookiecutter
- cookiecutter-pypackage

CHAPTER 2

Installation

At the command line:

```
$ easy_install hamsterlib
```

Or, if you have virtualenvwrapper installed:

```
$ mkvirtualenv hamsterlib
$ pip install hamsterlib
```

CHAPTER 3

Usage

To use hamsterlib in a project:

```
import hamsterlib
```

The main point of entry is `hamsterlib.HamsterControl`. Your friendly timetracking controler. All that is required to initialize it is that you pass it a `dict` with basic configuration information. Right now, all that is needed are the following key/value pairs:

```
'work_dir': ``path``; Where to store any temporary data
'store': 'sqlalchemy'; refer to ``hamsterlib.lib.REGISTERED_BACKENDS``
'db_path': ``sqlalchemy db path``,
'tmpfile_name': filename; under which any 'ongoing fact' will be saved
'fact_min_delta': integer; Amount of seconds under which fact creation will be␣
↪prohibited.
```

`hamsterlib.HamsterControl` initializes the store and provides a general logger. Besides that `HamsterControl.categories`, `HamsterControl.activities` and `HamsterControl.facts` are the main interfaces to communicate with the storage backend.

The second cornerstone are the dedicated classes `Category`, `Activity` and `Fact` which, for convinience, can be imported right from `hamsterlib`. In particular `Fact.create_from_raw_fact` might be of insterest They provide easy and consistent facilities to create, store and manage data relevant to your timetracking needs. Of particular interest is `hamsterlib.Fact.create_from_raw` which allows you to pass a `raw_fact` string and reciceve a fully populated `Fact` instance in return. The class will take care of all the tedious parsing and normalizing of data present in the `raw_fact`.

For clients aiming to utilize the new and sanitized backend API a look into `hamsterlib.storage` may be worthwile. These classes describe our baseline API that is to be implemented by any valid backend of ours. Note that some general methods are provided on this level already, so backend developers don't have to each time anew. Of cause they are always free to overload them in order to implement solutions optimized to their concrete backend infrastructure.

Besides this general controler `hamsterlib.helpers` provides convinience functions that help with normalization and general intermediate computation clients may have need for.

3.1 Basic Terminology

The following is intended as a rough description of the basic semantics of terminology used as part of this project. For technical details please refer to the module reference, in particular `hamsterlib.objects`.

Category What it says on the tin. A user friendly way to group accitities that relate to each other. Their names are unique.

Activity 'What you are doing'. This is a brief and easy to remember describtion of the (you guessed it) 'activity' you want to track. An *activity* can be filed under a category in order to provide some structure or just stay *uncategrized*. While one 'activity name' can be used with multiple *categories* it will be considered as a different thing all together as far as we are concerned. E.g. an activity called 'meeting' filed under the 'private' category will be absolutly seperate from an activity named 'meeting' filed under 'bussiness'. Within each *category*, activitynames will be unique.

Fact An actually timetracked activity. That is, an entry about 'what did you do from start to end'. As such it connects an general *Activity* with timetracking information as well as additional optional context infos (tags and description). A *fact* is usually what you are ultimativly interested in. What shows up in your report and allows you to see what you did when.

Ongoing fact Legacy hamster allowed for facts without an end to be saved to the database. We do not. However, to address the common use case that a client may want to start tracking an activity, but does not know its end, we provide a convinient solution so clients don't have to implement this each by anew. We provide an API for creating one and only one persistent *ongoing fact*. A fact without specified end. This fact is treated seperatly the others in almost any regard internaly. As far as the client is concerned it is however just a regular fact without specified end. Fact manager methods relevant to this carry `tmp_fact` in their name.

This documentation need to be expanded, but hopefully it is enough for now to get you started. For detail please see the module reference and tests.

3.2 Assumptions and Premisses

As any software, we make assumptions and work on premises. Here we try to make them transparent.

- There can be only one fact any given point in time. We do not support multiple concurrent facts.

CHAPTER 4

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given. Further details on labels and their respective meaning can be found in the wiki.

You can contribute in many ways:

4.1 Types of Contributions

4.1.1 Report Bugs

Report bugs at https://github.com/projecthamster/hamster-lib/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

4.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

4.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "feature" is open to whoever wants to implement it.

4.1.4 Write Documentation

'hamster-lib' could always use more documentation, whether as part of the official 'hamster-lib' docs, in docstrings, or even on the web in blog posts, articles, and such.

4.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/projecthamster/hamster-lib/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

4.2 Get Started!

Ready to contribute? Here's how to set up *hamster-lib* for local development.

1. Fork the *hamster-lib* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:projecthamster/hamster-lib.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development. It will also take care of installing all packes required for a dev environment:

```
$ mkvirtualenv hamster-lib
$ cd hamster-lib/
$ make develop
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ make test-all
```

   For your intermediate quick-and-dirty testruns that include just the unittests, run:

```
$ make test
```

   If you just want to check against a specific python (`py27` or `py34`) version, run:

```
$ tox -e py27
```

   or:

```
$ tox -e py34
```

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

4.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests. Preferably they will not lower the total test coverage of the project.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.7 and 3.4. Check Travis and make sure that the tests pass for all supported Python versions.

4.4 Tips

To run a subset of tests:

```
$ python -m unittest tests.test_hamster_lib
```

CHAPTER 5

Packaging

`hamsterlib` follows the semantic versioning scheme. Each release is packaged and uploaded to pypi. We provide a compliant `setup.py` which contains all the meta information relevant to users of `hamsterlib`. If you stumble upon any incompatibilities or dependency issue please let us know. If you are interested in packaging `hamsterlib` for your preferred distribution or in some other context we would love to hear from you!

5.1 About requirements/*.txt

We do fully follow Donald Stuffts argument that information given `setup.py` is of fundamentally different nature than what may be located under `requirements.txt` (Additional comments can be found in the packaging guide and with Hynek Schlawack). As far as packaging goes `setup.py` is authoritative. We provide a set of specific environments under `requirements/*` that mainly developers and 3rd parties may find useful. This way we can easily enable contributers to get a suitable `virtualenv` running or specify our test environment in one central location. If for example you wanted to package `hamsterlib` for `debian-stable`, it would be mighty convenient to just provide another requirements.txt with all the relevant dependencies pinned to what your target distro would provide. Now you can run the entire test suit against a reliable representation of said target system.

CHAPTER 6

Labels and Milestones

Each issue should have at least one label from the Type and Status section assigned.

6.1 Type (#cc317c)

Enhancement Issues that introduce new functionality. Original post should include the following information:

- Description of desired functionality
- A (prosaic) description of a use case
- Optionally add suggestions/ideas about how to implement the feature. As always, PRs are welcome. :)

Bug Issues about something not working as intended. Original post should include the following information:

- Platform (operating system, architecture)
- Version of package in question
- Steps suitable to reproduce the problem
- Error message/console output
- If you are willing to share: log files

In case of an error within the documentation the above requirements can be omitted.

Duplicate Indicates that the issues topic has been addressed before and all discussion/comments should happen in the referenced issue instead.

Question Indicates that the issue is less about actual code/functionality but addresses a general (design) question that needs to be decided upon before an implementation can be considered.

Story Used to track multiple related issues.

6.2 Topic (#fbca04)

Issues without a specific topic assigned deal with general functionality. The main purpose if these labels is to allow contributors with specific skill sets to find issues fitting them.

UX Issue does not deal with functionality itself but with user interaction.

UI Issue deal with visual representation of functionality.

Documentation Issue does not require actual coding or even necessarily python knowledge at all but deals with documenting the project itself. It may be used to indicate improvements to the code's documentation, in which case a basic familiarity with the language and code layout is highly desirable. However, it may also be used for issues dealing with front end user facing documentation that elaborates on how to use the package in a plain natural language.

Packaging Issues related to project package releases.

Meta Issues related to the general project setup.

6.3 Status (#159818)

Labels that indicate the status of an issue. Their provide a quick and easy answer to whether the issue is actionable or not.

Decision needed Tickets that need a design decision are blocked for development until a project leader clarifies the way in which the issue should be approached.

Information needed This label indicates that the issue has not enough information in order to decide on how to go forward. See the documentation about our triage process for more information.

Help needed Designates issues which seem to require a certain skill currently not available to the core developers. Such issues are unlikely to be solved unless a contributor with the required skill-set steps forward to help out. Giving pointers to domain specific resources or best practices may already be enough. This does not necessarily imply that all the actual coding has to be done by the person providing the desired skill.

Ready The issue has been screened by the core devs and may be worked upon at your leisure.

Blocked This issue can only be addressed once another issue has been resolved. The cause may be an internal issue or external dependency.

In progress: Issues currently worked on. If you want to join work on it please coordinate with its assignee to achieve the best possible solution and avoid duplicate work.

Rejected: Issues that are not considered within the general goals of the project. A reference to said previous discussion/issue should be given.

6.4 Other

Labels that did not warrant their own group.

Ready for review Pull Requests that are considered complete. A review by at least one core developer is required prior to merging it.

Good First Bug This label marks tickets that are easy to get started with. The ticket should be ideal for beginners to dive into the code base, indicating low-hanging fruits. These tickets generally should fit the following requirements:

- No comprehensive knowledge of the entire code base needed.
- No particular 3rd party library familiarity required.
- Most likely does not involve long term effort.
- No elaborate design decisions involved.

CHAPTER 7

General

class compact

- Follow PEP 8 and PEP 257.
- Try to stick to 79 chars. When this is not enough you may use up to 99 chars. This is more tolerable for code than for documentation.
- Use double quotes for human readable strings and single quotes for all other strings.
- Private functions and methods are prefixed with a single underscore: `_method` .

7.1 Python 2 and 3 compability

class compact

- Declare encoding in first line: `-*-encoding: utf-8 -*-`
- Use *absolute_import* and *unicode_literals* from the `__future__` package.
- Use *six.text_type* to cast a unicode string under python 2 and 3.

7.2 Code-style

- Readability trumps almost anything. Readable and approachable code carries it's weight as a lower contribution-barrier, less bugs and easier debugging. Having a particular clever, aka dense, alternative is rarely warranted.
- Favour multiple small specialized (local) functions/methods over big all-encompassing ones.
- Use well established and maintained high quality 3rd party libraries over own implementations.
- Use expressive variable names. If you have to trade off verbosity and expressiveness, go for the later.
- Assigning variables even if they are used only once can be preferable if expressions become clearer and less dense.
- Any method / function that is not deliberately considered part of the public API should be considered private. This is to increase the mental threshold for declaring them public as well as making it easier to the occasional reader to figure out which parts of the code are relevant to his/her needs and which are internal details.
- Try to minimize use of return statements with a method/function while using exceptions wherever suitable. While this may not allways improve readability it tends to make debugging easier as it provides one central breaking point.

- Methods should have the following order: special (`__foo__`) > public > private (*_foo'*).

7.3 Imports

class `compact`

- Imports should be grouped in the following order:
  - standard library imports
  - related third party imports
  - local application/library specific imports
- You should put a blank line between each group of imports.
- Always order each group of imports by name.
- You can use isort to sort the imports.
- Remove import statements that are no longer used when you change code.

7.4 Documentation

class `compact`

- Docstrings should be provided for all public and private classes, methods and functions. Simple local functions may go without. They should elaborate the methods signature and use.
- Use google-style docstrings. Sphinx's napoleon extension will make turn this into valid `rst` .
- use block comments to explain implementation

7.5 Committing and commit messages

class `compact`

- Commit one change/feature at a time (you can use tig to select the changes you want to commit).
- Separate bug fixes from feature changes, bugfixes may need to be backported to the stable branch.
- Maximum line length is 50 characters for the first line and 72 for all following lines.
- The first line is a short summary, no trailing period.
- Leave a blank line between the summary and the body of the commit message.
- Explain what you did and add all relevant information to the commit message.
- If an issue exists for your feature/bug/task add it to the end of the commit message.
- Run the test suite **before** pushing your changes.
- **Never commit** passwords, `*.pyc` files, sqlite database files or `pdb` calls.

7.6 Rebasing

class compact

- try to rebase to keep the commit history linear:

```
$ git pull --rebase
```

- If you have uncommitted changes in your working directory use `git stash` to stash the changes while rebasing:

```
$ git stash
$ git pull --rebase
$ git stash pop
```

- **Do not** rebase already published changesets!

7.7 Pull Requests

class compact

The title of a pull request should contain a summary of the issue it is related to, as well as the issue id. An example would look like `Advanced report options (#23)` . This way, a link between the PR and the issue will be created.

Every pull request has to be approved by at least one other developer before merging.

CHAPTER 8

Notes

These notes are just a dumping ground for semanitic information extracted from legacy hamster while dealing with its codebase that is not documented/obvious. Also, some basic troubleshooting heuristics and 'lessons learned' may be documented here for now.

- Why all this buisiness with `search_names` , which are lowercase versions of proper names? Is it because cases insesitive matching was not available? or due to performence considerations?
- It looks like the dbus client assume PKs to be > 0 and uses 0 as marker for failure. Would be great if we can change that on the frontend instead of working around that.
- Whilst we do support networked/distributed storage acces we do not at all can provide multiple simultanious connections. Our backend uses the fact that an Object has a PK to decide if it updates or inserts. If any other "client" has manipulated the data stored under this key between this clients retrieval and its save call, those changes will simply be overwritten.
- force_flush on SQLAlchemy-factories helped with:

  > sqlalchemy.exc.IntegrityError: (raised as a result of Query-invoked autoflush; consider using a session.no_autoflush block if this flush is occurring prematurely) (sqlite3.IntegrityError) UNIQUE constraint failed: categories.name [SQL: 'INSERT INTO categories (id, name) VALUES (?, ?)'] [parameters: ((19, 'vero'), (20, 'officiis'), (21, 'aliquid'), (22, 'vero'), (23, 'dolorum'))]

  after we started using factory-instance fixtures 'alchemy_category' vs 'alchemy_category_factory'
- Integrity Error (SQLAlchemy) If we end up with 'constraint' or 'Integrety' erros although we should not have commited anything to the db, it may be that one of our unsuspicious queries nearby triggered an autoflush/commit. Popular candidates are lookup- and count queries. One way to get around this using instances of classes not tracked/mapped by SQLAlchemy.

8.1 Not supported legacy 'functionality'

Not now:

- tags
- `search_name`
- indexing
- ical export
- autocomplete
- `resurrect` /`temporary` mechanic

- `get facts` can inverse search_terms
- trophies
- migration from old database aka `run_fixtures` .

Opted against:

- `__solve_overlaps`
- `__squeeze_in`
- `__touch_fact)`

8.2 Legacy Storage API notes

- `get_tag_ids` seems to create tags that have been passed if they do not exist * activities flagged as `temporary` dont get ressurected (`__add_fact`).
- seperate `storage.__get_activities` is dedicated to autocomplete. we summerized its usecase under the regular one so far. The difference seems to be that autocomplete reasonable needs a way to retrieve *all* activity names, irrespective of category association. This should be coverable by adding a `categories=False` flag to our default method. Worth noting: considers only non-deleted activities. Activities are returned ordered by their corresponding facts start time with the 'latests' beeing first. Maybe it is actually cleaner to add a dedicated method like this once we get to autocomplete.

Dismissed:

- resurrect/temporary for `add_fact` is about checking for preexisting activities by using `__get_activity_by_name` . If True we will consider 'deleted' activities and stick this to our new fact.
  - We don't do temporary facts.
- if an activity is created with `temporary=True` it will be marked as `deleted=True` . why not set the attribute directly? Whats the role of a temporary activity?
  - This is only used when creating *temporary facts* in order to prevent proper activities beeing created for them. We don't do temporary facts, so we can ommit this.

8.3 Things we try to improve

- python >=2.7, >=3.4 support
- full unicode support
- full pep8 and 257 complience
- >=95% test coverage
- strict and honest *seperation of concerns*. We provide just the backend, but that we do proper. * cleaner, more object oriented pythonic code
- 'one exit point' strategy for method return values. Reduce the spagettiness.
- modular architecture.
- focus on solid core functionality and only expand features once existing code meets our standart.
- better project layout including waffle.io, codeship.com and requirements.io

- fully integrated and focused on PyPi distribution. All you need for production, test or dev comes out of the box with regular python tools.

CHAPTER 9

Credits

9.1 Development Lead

- Eric Goller <eric.goller@ninjaduck.solutions>

9.2 Contributors

9.3 Code taken from 'legacy hamster'

- tbaugis: * `parse_time_info` from hamster-cli * `XMLWriter` from hamster

CHAPTER 10

History

10.1 0.11.0 (2016-07-06)

- Renamed this package to `hamster-lib` as it now an offical part of projecthamster. It was previously named and distributed as hamsterlib
- Add documentation checker `pep257` to testsuite.
- Fixed docstrings.
- Removed `hamster_lib.objects.Fact.serialized_name`.
- Improved test factories
- Made `hamster_lib.objects.*` *hashable*.
- Provide consistent and improved `__repr__` methods for `hamster_lib.objects` classes.
- `FactManager._get_all` can now return facts completely*or* partially within the timeframe. As a consequence, we removed `FactManager._timeframe_is_free`.
- Added a set of helper function to ease common configuration related tasks. In particular they make it easy to store a given config at its canonical file system location.
- Improved `ActivityManager.get_all` to enable it to return *all* activities.

10.2 0.10.0 (2016-04-20)

- Add `ical` export facilities. Brand new writer using the `icalendar` library.
- Add `xml` export facilities.
- Switch to semantic versioning
- Added GPL3 boilerplate
- Provide documentation on packaging and `requirements.txt`.
- Add support for editorconfig
- Introduce fine grained, storage backend dependent config options.

10.3 0.0.3 (2016-04-08)

- fact managers `save` method now enforces new `fact_min_delta` setting.
- Fixed broken packing in `setup.py`.
- Storage manager methods now use extensive logging.
- Documentation moved to 'alabaster' theme and content extended.
- Remove usage of `future.builtins.str`.
- Adjusted `release` make target.

10.4 0.0.2 (2016-04-07)

- First release on PyPi
- Improved documentation
- Support for *ongoing facts*.
- Updated requirements

10.5 0.0.1 (2016-04-03)

- First release on github

CHAPTER 11

Indices and tables

- genindex
- modindex
- search

C