Personal Status Logger

Release

Sam Kleinman

March 30, 2013

Contents

1 Contents **ii**
- 1.1 Getting Started with Status Logger . . . ii
  - Overview . . . ii
  - Procedure . . . ii
- 1.2 Status Logger Use and Operation . . . iii
  - Manual Pages . . . iii
  - Getting Started . . . vii
  - General Operation . . . vii
  - `projects` Configuration . . . vii
- 1.3 Internal Details and Implementation . . . viii
  - Overview . . . viii
  - Programs . . . viii
- 1.4 Contribute to the Status Logger Project . . . xv
  - Source Code . . . xv
  - Bugs/Issues . . . xv

2 Legacy Implementation **xv**
- 2.1 Legacy `zsh` Implementation . . . xv
  - Overview . . . xv
  - Pages . . . xv

3 Resources **xxv**
- 3.1 Stats Logger Reference . . . xxv
  - Glossary . . . xxv
  - Additional Reference . . . xxv

4 Overview **xxv**

5 Future Development **xxvi**

Python Module Indexxxvii

1 Contents

1.1 Getting Started with Status Logger

Overview

You have a couple of options for implementing status logger:

1. Use the `simple` method which creates a directory with the Python scripts and symbolic links. Place the content of this directory in your `PATH` and being using stl directly.

   Use this if you want the most hands on experience with stl.

2. Build and install using Python setuptools.

   If you do work in Python and are comfortable with Python tools, this option may make the most sense for you.

In the future stl may be available as a package in the Python Package repository.

For full documentation of all stl components see:

- *Internal Details and Implementation* for documentation of the classes, methods, and implementation of each Python module, including .
- *Status Logger Use and Operation* for documentation of the command line interface of each module, including descriptions of use

Procedure

Regardless of the method you use, at this time you will need to download the repository, using the following command at a system prompt:

```
git clone http://git.cyborginstitute.net/repo/stl.git
```

Then from within the `stl/` directory follow either of the following processes.

Note: You may want to modify your files, particularly `wc_track` before installing stl.

Direct Installation

Issue the following command at the system prompt to “stage” the installation:

```
make simple
```

All required stl programs (and symbolic links,) are now in `stl/build/bin/`. Either copy the content of this directory into a directory in your search path, (g.e. `~/scripts/`) or add this directory to your search path, by adding the following lines to your shell `rc` or `profile` file (e.g. `~/.bashrc`, `~/.zshrc`, or `~/.profile`.)

```
PATH=$PATH:~/scripts
export PATH
```

You may have to reinitialize or source your `rc`/`profile` file for the change to take place. You’ll be able to use any of the stl programs from your system shell.

Install with `setup.py`

Issue the following command to build and install the stl python package:

```
sudo make install
```

This make target simply calls `python setup.py install`. Because this operation installs files in `/usr/bin` or a similar path, this requires root access (i.e. `sudo`.)

1.2 Status Logger Use and Operation

Manual Pages

`lnote` Manual

Synopsis `lnote` provides a simple way to add an arbitrary note to the log and notification system. While most of the information included in the `stl` output log is automatically generated, `lnote` allows you to provide context, to make manual log analysis easier.

Dependencies `lnote` depends on the following components:

- `argparse`, part of the Python standard library as of 2.7, and installable separately for some earlier versions.
- `socket`, part of the Python standard library, used to include the hostname in the log output.
- `datetime`, part of the Python standard library, used to build timestamps for log output.
- *sauron*, to provide an interface to Sauron by way of `emacsclient`.
- *wc_track*, to provide local configuration information.

Options

`-help`, `h`
: Returns a brief help message regarding available options and output.

`-target` `<daemon>`, **`-t`** `<daemon>`
: For users that run multiple named `emacs` daemon instances, this option allows you to send the *Sauron* notification to a specific named instances. Chosen from a list of daemons named in the *sauron* script.

`-message` `<message>`, **`-m`** `<message>`
: A string containing the arbitrary message to add to the notification.

Use A typical invocation of `lnote` resembles the following:

```
lnote -t hud -m 'message here'
```

`wc-track` Manual

Synopsis `wc-track` is a simplified wrapper around *stl* with *stored configuration*.

Options

`-help, -h`
: Returns a brief help message regarding available options and output.

`-force, -f`
: By default *stl* caches a copy of the word count for each project in the `/tmp/stl/` directory. If you pass *`--force`*, `wc-track` will log and notify on all word-count events *even if* the value has not changed since the last time `wc-track` ran.

`-project` `<project-name>`, **`-p`** `<p>`
: The name of the project. You must specify one of the projects defined in the *projects Configuration* by the `<project>` value.

  If you do not specify *`--project`*, `wc-track` will report all projects defined in the *projects structure*.

`projects` Configuration `projects` is a python dict, that holds a number of dicts for each project that you want to report on with *stl*. Consider the following prototypical project configuration:

```
'<project>' : {
    'path' : <path>
    'target' : <string>
    'emacs' : <bool>
    'quiet' : <bool>
    'log' : <path>
    'ext' : <string>
}
```

Consider the following documentation of each of these fields:

`projects`
: Holds the name of the project. Used as *`stl --project`*.

`projects.'path'`
: Holds the path to the top level of the project. Used as *`stl --directory`*.

`projects.'target'`
: Holds the name of the emacs instance to send the Sauron notification. Used as *`sauron --target`*.

`projects.'emacs'`
: A Boolean value. If `False`, this will disable output to emacs and Sauron.

`projects.'quiet'`
: A Boolean value. If true, will suppress all command line output. Used as *`stl --quiet`*.

`projects.'log'`
: Holds the path of the log file to record output data. Used as *`stl --log`*.

`projects.'ext'`
: Holds the extension of all project files, that *stl* will measure. All files with different extensions ignored. Used as *`stl --extension`*.

Use At the system shell, invoke `wc-track` as in one of the following examples.

```
wc-track
wc-track --project rhizome
wc-track -f --project rhizome
```

Often it makes sense to run this program automatically using `cron` or some other scheduling tool. Consider the following `crontab` lines:

```
*/2 9-18 * * 1-5              wc-track > /dev/null 2>&1
*/2 * * * *                   wc-track > /dev/null 2>&1
```

The first operation schedules `wc-track` to run every two minutes, between 9 am and 6 pm, Monday through Friday. The second operation runs `wc-track` every two minutes at all times.

Typically there is no need to use `wc_track` from other Python scripts, and no interface for that; however, you may want to access the *projects structure* in another script as needed:

```
from wc_track import projects as wc_track_projects
```

`sauron` Manual

Note: The `sauron` program is simply a symbolic link to `sauron.py`, for more "native feeling" command line use.

Synopsis `sauron` is a Python module that provides an interface to Sauron via the command line and `emacsclient`. Sauron is a notification system for emacs. `sauron` also provide a command line interface for sending messages to Sauron for use in other shell scripts.

Options

`-help, -h`
: Returns a brief help message regarding available options and output.

`-priority` `<int>`, **`-p`** `<int>`
: Defines the priority of messages. Typically *Sauron* ignores all notification events with priority values lower than `3`, though users may configure other defaults.

`-target` `<daemon>`, **`-t`** `<daemon>`
: For users that run multiple named `emacs` daemon instances, this option allows you to send the *Sauron* notification to a specific named instances. Chosen from a list of daemons defined in `sauron.py`.

`-source` `<source>`, **`-s`** `<source>`
: A short arbitrary string for use *Sauron's* `Orig` field.

`-message` `<message>`, **`-m`** `<message>`
: A string containing the arbitrary message to add to the notification.

Use A typical invocation from a system shell resembles the following:

```
sauron -t hud -p 3 'This is the text of the message.'
```

To send a Sauron notification from a Python module using `sauron`, ensure that `sauron.py` is in your Python path, and then use code that resembles the following:

```
import sauron

message = sauron.NotificationMessage(source='system',
                                     message='This is the text of the message.',
                                     target='hud')

message.send()
```

See the documentation of the `sauron.NotificationMessage` class for more information about this interface.

`stl` Manual

Synopsis `stl` is a command line word count tool for projects. *wc-track* provides a more consistent interface for this word count functionality, and wraps the operation of `stl`. Use the `stl` module as a starting point for adding additional data collection modules to the `stl` system.

Options

`-help, -h`
Returns a brief help message regarding available options and output.

`-target` `<daemon>`, **`-t`** `<daemon>`
For users that run multiple named `emacs` daemon instances, this option allows you to send the *Sauron* notification to a specific named instances. Chosen from a list of daemons defined in `sauron`.

If you do not modify the value of the `emacs_daemon`, **stl** will assume that your system only has one emacs instance running.

`-project` `<name>`, **`-p`** `<name>`
A simple, human digestible name for the project, used in notifiaction and logging output.

`-directory` `<directory>`, **`-d`** `<directory>`
The top-level path of the project. Used for word counting purposes.

`-extension` `<extension>`, **`-e`** `<extension>`
The extension of the project files. `stl` ignores all files with a different extension in the *`--directory`*. The default value is `txt` unless otherwise specified.

`-quiet, -q`
Suppress output on the command line. Disabled by default.

`-force, -f`
By default `stl` caches a copy of the word count for each project in the `/tmp/stl/` directory. If you pass *`--force`*, `stl` will log and notify on all word-count events *even if* the value has not changed since the last time `stl` ran.

`-logfile` `<path>`, **`-l`** `<path>`
A path to the logfile. By default there is no logfile.

Use On the command line, an invocation of `stl` might resemble the following:

```
stl --project rhizome --directory ~/wikish/rhizome/ --extension mdwn --logfile ~/stl.log
```

Consider the following section from *wc-track*, that wraps `stl`:

```
import stl

stl.generate_events( project='rhizome',
                     directory='~/wikish/rhizome/',
                     target='hud',
                     quiet=False,
                     log='~/stl.log'
                     force=True,
                     extension='mdwn' )
```

See *Internal Details and Implementation* for more information about the implementation of each component.

Getting Started

While `stl` may have more traditional Python packaging in the future, given the current state of development and organization, to begin using STL you should simply add the four constituent files to the *same* folder in you shell's search path, by adding one of the following lines to your `~/.bashrc`, `~/.bash_profile`, or `~/.zshrc` (or similar) file:

```
PATH=$PATH:~/bin
PATH=$PATH:~/scripts
PATH=$PATH:/opt/bin
```

Then, copy the files in the `stl/` directory in the `stl` repository to this directory. You can issue the following sequence of commands at your system prompt to download the files hosted on github:

```
curl http://raw.github.com/cyborginstitute/stl/master/stl.py > stl.py
curl http://raw.github.com/cyborginstitute/stl/master/lnote.py > lnote.py
curl http://raw.github.com/cyborginstitute/stl/master/wc_track.py > wc_track.py
curl http://raw.github.com/cyborginstitute/stl/master/sauron.py > sauron.py
```

Then create symlinks in this directory for easy use without the `.py` extension, with the following commands. Replace `~/scripts` with the path to your personal script directory:

```
cd ~/scripts
ln -s stl stl.py
ln -s lnote lnote.py
ln -s wc-track wc_track.py
ln -s sauron sauron.py
```

Continue reading the *manual pages* pages the *internals section* for more information about the use and implementation of stl.

See

General Operation

`stl` consists of three connected Python modules/scripts that you may use either independently or in conjunction. From the highest level, these programs are:

- *sauron*: A script that provides a wrapper around emacsclient, and ethe emacs notification system *Sauron*. Supports multiple emacs clients running on a single system/user account, and does not require a *dbus* configuration. This provides a programatic interface for sauron-mode.
- *stl*: A script that provides a simple interface to calculate and log word counts for multi-file projects. This script does not store any information regarding projects or configuration.
- *wc-track*: For running regularly as a cronjob, `wc_track` stores the `projects` dict that contains per-project configuration, and several functions for automatically running `stl` for each project.
- *lnote*: A simple interactive script for creating arbitrary notes in the log to provide additional context.

`projects` Configuration

The `projects` dict in the `wc_track` file, provides a way to pre-configure `wc_track` as a wrapper around `stl`. Consider the following basic setup:

```
username = tychoish

projects = {
  'stl' : {
        'path' : '/home/' + username + '/projects/stl/docs/source/',
        'target' : 'projects',
        'emacs' : True,
        'quiet' : True,
        'log' : '/home/' + username + '/projects/stats-' + socket.gethostname() + '.log',
        'ext': 'txt'
   },
}
```

See *wc-track Manual* and *wc_track Internals* for more information.

1.3 Internal Details and Implementation

Overview

After experiencing trouble with the integrated approach of the *zsh implementation*, the current implementation takes a more modular approach.

The two core components, *stl* and *sauron*, provide core functionality collecting data and creating notifications respectively. The additional components, *lnote* and *wc_track* wrap *stl* and *sauron* to provide more useful automatic operation. In the case of *lnote*, to provide a way of creating arbitrary log messages to annotate a status log: and in the case of *wc_track*, to provide a way of automatically collecting word count data from a number of pre-configured projects.

The documentation listed below provides an overview of the internal implementation of each Python module, including code samples, with particular attention toward enhancement and future extension. For usage information consider the *Status Logger Use and Operation* documentation.

Programs

sauron Internals

Overview *saroun* provides a thin wrapper around Saruon so that programs running outside of emacs can send notifications via the shell, either remotely or locally, without needing to use dbus or other methods. `sauron` requires that your system runs Emacs as a daemon, and contains support for multiple emacs daemons running on the same system or under the same user account.

Although you can run the command directly, as described in *sauron*, you will typically use `sauron` by way of the `NotificationMessage()` class, or in another shell script.

Dependencies

- `argparse`
- `os`
- `subprocess`
- `socket`
- `datetime`

Implementation The implementation of the `sauron` has the following components:

Data

sauron.**work_emacs_daemons**
A list of "work" emacs daemons.

sauron.**personal_emacs_daemons**
A list of "personal" emacs daemons.

sauron.**emacs_daemons**
A new list consisting of all the elements of `work_emacs_daemons` and `personal_emacs_daemons`.

The first instance in this list is always the "default" emacs used in other scripts.

The distinction between "work" and personal allows you to maintain different log files. Coordinate these variables with the paths to the log files in `wc_track`.

Methods

sauron.**parse_message**(*message*)
Processes messages, to normalize formats for messages submitted as strings or via the command line for use by Sauron.

Interfaces and Classes

class sauron.**NotificationMessage**(*source=<hostname>*, *target=<emacs_daemon>*, *priority=<3>*, *message=<None>*)
`NotificationMessage()` is the primary interface for `sauron` and the other modules in the *stl* suite as well as by the `main()` method. When creating `NotificationMessage()` objects, you only *need* to pass the `message` argument. Read the documentation of the following default instance objects in the `NotificationMessage()` class for information about the parameters:

target
Defaults to the first item in `emacs_daemons` array. Used to determine to which emacs instance `send()` will deliver the notification.

priority
Defaults to `3`, which is the lowest priority of Sauron messages that are conveyed to emacs users by default. Sauron hides lower priorities unless users configure a different threshold.

source
A string, passed to sauron for the `Orig` field of the Sauron display. Use this to describe or specify the process or script that sends the notification. Defaults to the system hostname.

message
The test of the massage. You must specify a value for this variable.

NotificationMessage.**send**()
Call the `send()` to send a message to Sauron as configured, as in the following invocation:

```
n = NotificationMessage( message="this is a test message.")
n.send()
```

NotificationMessage.**log**()
`log()` is equivalent to `send()` except that it write `message` to a log file. This method will attempt to import the `wc_track.work_log` and `wc_track.personal_log` values from the `wc_track` module, and will output the log message to standard out if there are no log files specified.

Consider the following invocation:

```
n = NotificationMessage( message="this is a test message.")
n.log()
```

`sauron.`**`cli`**`()`

Collects input from the command line using `argparse`. See *sauron Manual* for more information about the command line interface.

The return value of `cli()` is the output of `argparse.ArgumentParser.parse_args()`.

`sauron.`**`main`**`()`

The entrance point for `sauron` when called from the command line. Collects output from `cli()`, creates a `NotificationMessage` object, and then calls `send()` on the object.

Extension and Development In many ways, the entire *stl suite* is a wrapper and extension of the `sauron` module. Future development of `sauron` will focus on more flexible logging, options to provide more structured logs, and increased capacity with other emacs configurations. The command line interface might benefit from some additional work or other changes.

`NotificationMessage` encapsulates all functionality, and makes it easy to wrap and send notifications from other scripts.

stl Internals

Overview `stl` performs the core operations of the stl suite:

- collects word count data from project files.
- caches and maintains word count data (in `/tmp`) to prevent overreacting of unchanged data.
- integrates with `sauron` to provide logging and notification.

Dependencies `stl` depends on the following system tools:

- `wc`
- `grep`
- `sed`
- `find`

Additionally, it uses the following python modules:

Internal Modules `sauron`

Standard Library Modules

- `os.path`
- `subprocess`
- `argparse`

Implementation

`stl.`**`shell_word_count`**`(`*directory*, *extension='txt'*`)`

Parameters

- **directory** (*string*) – The path of the top level directory that contains the project's documents. Pass in as a string.

- **extension** (*string*) – The file extension, without the preceding period (i.e. `.`) of all project files. All project files *must* have an extension. `stl` only counts the words in files that have this extension.

Returns The word count for the project files in the specified directory.

`stl.`**`wc_message_builder`**(*project*, *directory*, *force=None*, *extension=None*)

Parameters

- **project** (*string*) – The name of the project. `stl` uses this string to report and track the word count for a specific project, to avoid over-reporting, and to annotate logs.
- **directory** (*string*) – The path of the top level directory that contains the project's documents. Pass in as a string.
- **force** (*bool*) – When true, always return the word count, even if the word count value is the same as the cached value.
- **extension** – The file extension, without the preceding period (i.e. `.`) of all project files. All project files *must* have an extension. `stl` only counts the words in files that have this extension.

`wc_message_builder()` is the main point of integration between the parts of the script that collect and process the word count data (i.e. `wc_message()` and `shell_word_count()`) and `sauron()` that handles and produces the notification and logging.

Not intended for direct use.

`stl.`**`wc_message`**(*word_count*, *project*)

Parameters

- **word_count** (*string*) – The word count value as a string.
- **project** (*string*) – The name of the project.

Returns A string with a formed word count message used by `wc_message_builder()`.

`stl.`**`generate_events`**(*project*, *directory*, *target*, *quiet=False*, *log=False*, *emacs=True*, *force=False*, *extension=None*)

Parameters

- **project** (*string*) – The name of the project. Used to store the cached word count values.
- **directory** (*string*) – The path of the directory that holds the project files.
- **target** (*string*) – The name of the emacs daemon to send the notification to. If `None`, does not send notifications.
- **quiet** (*bool*) – If `True`, suppresses output (on standard output.) Defaults to `False`.
- **log** (*bool*) – If `True` write output to the log. Uses the `sauron.NotificationMessage.log()` method. Defaults to `False`.
- **emacs** (*bool*) – If `True` send the Sauron notification. Defaults to `True`.
- **force** (*bool*) – If `True` passes `force=True` to `wc_message_builder()`, which impels reporting, even when the current value is the same as the cached value
- **extension** (*string*) – The extension of the project files, to limit reporting to only relevant project files. If `None`, the default, `generate_events()` does not pass the `extension` parameter to `shell_word_count()`.

stl.**cli**()

Defines and describes the command line interface provided by `argparse`. See *stl Manual* for full documentation of this interface.

stl.**main**()

Logically the flow of data through the methods in `stl` is:

1. data from `cli()` passes in from the user into the `generate_events()`.
2. `generate_events()` calls `wc_message_builder()`, and if `wc_message_builder()` returns data, then `generate_events()` creates and distributes notifications.
3. `wc_message_builder()` calls `shell_word_count()` to return the word count, and then uses `wc_message()` to format and return the word count message to `generate_events()` where all of the main action is.

Extension and Improvement

- Data generated by `stl` could be pre-aggregated in some way (running totals, daily progress, etc.), or collected in some system more easily analyzed.
- For projects stored in git, some information about commit, update (i.e. pull) events or branch context, might provide some basic insight when reviewing `stl` logs. While each system logs to its own log file, some sort of centralization or synchronizing of log data may be useful.

lnote Internals

Overview *lnote* is a basic example of a *sauron* and *stl*, wrapper. It allows users to insert messages into the message

Practically being able to insert arbitrary messages into the log is great for personal logging because it allows you to provide a little bit of context. At only 45 lines, the module also demonstrates the small account of code required to extend and collect data for the personal log.

Dependencies *lnote* depends on the following Python modules:

Internal Modules

- *sauron*
- *wc_track*

Standard Library Components

- `argparse`
- `datetime`

Methods

lnote.**cli**()

Defines the basic basic command-line interaction. If the `sauron.emacs_daemons` list is empty then it suppresses Sauron notification.

Takes no arguments and returns the `argparse` parsed arguments.

lnote.**send_message**(*note*, *target*)

Parameters

- **note** (*string*) – The message text. Passed in `lnote.main()` from the command line output returned by `lnote.cli()`.
- **target** (*string*) – The name of the emacs daemon to send the Sauron notification to. Must be in the `sauron.emacs_daemons` structure.

Creates and object of the `sauron.NotificationMessage` class and sends the message. Also writes the message to a log. If you need to modify the loggoing output, edit this part of the module.

`lnote.`**`main`**`()`

The primary entry point into *`lnote`*. Simply calls the `cli()` method and pass its data into `send_message()`.

Extension In general `lnote` is a skeleton `stl` tool. To create new tools:

- copy `lnote`,
- create a new function that returns a different message, and
- optionally change the `source` parameter in the creation of the `sauron.NotificationMessage` object, within `send_message()`.
- customize the logging behavior of `send_message()` as needed.

Everything else is standard.

wc_track Internals

Overview `wc_track` is a wrapper around `sauron` and `stl` that provides an initial implementation of per-project tracking and operation. See *wc-track Manual* for more information about use.

In most cases, you will not need to use `wc_track` from another Python module, although some of the data in the `projects` dictionary may be useful in other scripts. In the future the configuration of projects may be distinct from `wc_track`, but at the moment you will need to edit `wc_track.py` before installing and using *wc-track*.

Dependencies

Internal `stl`

Standard Library

- `socket`
- `os`
- `argparse`

Implementation

Variables and Data

`username`

Stores the current username. By default, uses `os.getusername` to get the username of the user that owns its process. Override if needed, and with caution.

personal_log

Stores the path of a "personal" projects log file. Defaults to: `~/.stl/personal-stats-<hostname>.log`, where `<hostname>` is the output of `python.socket.gethostname()`. Override as needed.

work_log

Stores the path of a "work" projects log file. Defaults to: `~/.stl/work-stats-<hostname>.log`, where `<hostname>` is the output of `noindex:python:socket.gethostname()`. Override as needed.

projects

A data structure (python dictionary) that stores configuration information for projects that you will use `wc_track` to collect data on regularly. for tracking.

Each project has a key in `projects`, which holds a dictionary. The keys of that dictionary map to the arguments to `stl.generate_events()`/*stl*. The `projects` the following keys:

projects.<project>.path

The path to the project's top level directory. `wc_track` passes this value as `directory` to `stl.generate_events()`.

projects.<project>.target

The name of the emacs daemon to send notifications about this project. `wc_track` passes this value as `target` to `stl.generate_events()`.

projects.<project>.emacs

A Boolean value. If `True`, send a notification to emacs via `sauron`. `wc_track` passes this value as `emacs` to `stl.generate_events()`.

projects.<project>.quiet

A Boolean value. If `True`, suppress all output on the console. `wc_track` passes this value as `quiet` to `stl.generate_events()`.

projects.<project>.log

A filesystem path to the log. Typically `personal_log` or `work_log`.

projects.<project>.ext

The extension of the project files. `wc_track` passes this value as `quiet` to `stl.generate_events()`.

set_quiet()

Parameters

- **force** (*bool*) – Pass `True` to override the value of `projects.<project>.quiet`.
- **project_quiet** – A value, likely a bool, which `set_quiet()` will return if `force` is `False`.

A helper method that takes two arguments. If the first is true, the method will return `True` (and override the `projects.<project>.quiet`,) otherwise `set_quiet()` returns the second value.

main()

The core operation of the program. Includes the following operations:

- Read input and user from the command line.
- If users do not specify and project, iterate through all projects in `projects` and report all statistics if different from the cached values. (*Unless passed forced on the command line.*)
- If users *do* specify a project, report only thhose projects, and only if they've not changed in the since (*Unless passed forced on the command line.*)

See *wc-track Manual* for more information.

Extension In most cases, there are few possible modifications or extensions to `wc_track`, but it serves as a good example for the kinds of modification and interfaces that you could provide to the kinds of personal statistic monitoring and recording as part of STL and future related tools.

1.4 Contribute to the Status Logger Project

Source Code

`stl` source code is available from the following git hosting providers:

- stl git repository (Cyborg Institute.)
- stl on Github

Feel free to clone or fork at your leisure. Issue a pull request on GitHub or send me an email/IM/IRC message if you want to send a patch, or would like me to pull from another repository changes back into "maineline." I will accept patches of all sizes against both the source and documentation.

Bugs/Issues

For the moment, I will track issues using the Cyborg Institute Listserv. Just write an email to the list with your issue or bug, or send me a message directly and I'll start a new thread.

2 Legacy Implementation

The following documentation reflects the *prior* implementation of `stl`, and remains for historical and archival reasons.

2.1 Legacy `zsh` Implementation

Overview

Initially I implemented `stl` as a zsh script. These pages document that program and it's use and operation.

Pages

`zsh stl` Manual Page (Legacy)

Note: This documentation reflects the legacy implementation of `stl`. See *Status Logger Use and Operation* for the current implementation.

Synopsis This document provides an overview of `stl` from the perspective of the user concerning both the script itself as it exists in "stock" format, and how many will choose to customize the script. See "*zsh stl Internals (Legacy)*" for more information on the underlying functions, operation, and the code paths of the script.

Basic Usage This section describes the invocation and purpose of various `stl` commands, ignoring most of the internals of the script.

Note: While you may want to set up interfaces for calling `stl` directly, in most cases `stl` will run fairly regularly as a cron job.

`stl` commands take the following basic form:

```
stl [domain] [worker] [project] [options]
```

The elements of this command are:

`[domain]`
You may omit this layer in some cases, but is useful if you need to maintain two separate log files, with two separate sets of projects.

If this term does not match one of the defined commands, then the program exits with help text.

`[worker]`
In the default implementation this is either "`make`", "`stats`", `report`", or "`output`" and defines the major fork in the behavior of the program. "`make`" will build a project, while "`stats`" provides access to word count and latest build times, "`report`" displays the output of the last build, while "`output`" is responsible for modifying the default output style.

The default worker option is "`stats`".

`[project]`
A keyword that defines each project. `stl` requires this option for successful output. The `make` worker only accepts one argument, while `stats` can handle multiple projects in some cases.

`[options]`
Some workers, accept additional arguments or messages.

The `stats` worker is the default and returns statistics about your projects. The options are:

- `wc`, generates a word count for the project
- `build`, returns information about the last build generated by the `make` worker.
- `force`, force `stl` to generate output even if the value of the output has not changed since the last output.

The `entry` worker provides the capacity for recording arbitrary message to the log. The options are:

- `start`, append a note to mark the beginning of a period of work on a project.
- `stop`, appends a note to mark the end of a period of work on a specific projec.t
- `note`, appends the remainder of the command line arguments to a note that `stl` writes to the log.

The `make` worker runs a specific build routine for a project, you will configure options by default when setting up the project. For `sphinx` projects, the `make` worker provides the following options:

- `clean`
- `html`
- `latex`
- `epub`
- `sffms`

These correspond directly to a target in the default `Makefile` that `sphinx-quickstart` generates. You may specify multiple options to generate multiple outputs. The `ikiwiki` builder uses the remainder of the command line argument as string that becomes the commit message for the wiki's git repository.

The `report` worker provides an interface to view the build reports generated by previous runs of the `make` worker. `report` always displays *only* the last build for whatever project you specify. You must specify one of the following log viewers.

- `less` opens the build report file using the **less** command.
- `more` opens the build report file using the **more** command.
- `cat` outputs the build report file using **cat**.
- `emacs-new` opens the log file in a new graphical **emacsclient** window.
- `emacs` opens the log file in an existing **emacsclient** instance.
- `emacs-term` opens the log file in a terminal **instance** emacsclient.
- `term` opens the build report in a new terminal window (i.e. **urxvtcd**) using the **less** command.

Customizing `stl` The example `stl` included here is reasonably generic, but all users will need to customize the code at least a little. All user customizable code resides at the bottom of the file. Continue for more detail on these customizations.

At the very end of the file the following "`main`" function, which is the user's entry into the code, which resembles the following:

```
main(){
   ARG=($@)

   case $ARG[2] in
        ( make ) ACTION=make ;;
        ( stat* ) ACTION=stats ;;
        ( entry* ) ACTION=entry ;;
        ( report ) ACTION=report ;;
        ( * ) ACTION=stats ;;
   esac

   domain=$ARG[1]
   ARG[1]=()

   case $domain in
      ( tycho ) tycho-worker $ARG; exit 0 ;;
      ( job ) job-worker $ARG; exit 0 ;;
      ( * ) echo "help text"; exit 1 ;;
   esac
}
main $@
```

The first `case` statement sets a variable that the `action-handler` function uses. The second `case` statement selects the `domain`.

If you modify the first statement, add corresponding code to the `action-handler` function. `action-handler` calls the functions that *do something* (i.e. "actions.") The second case statement simply passes arguments to the next user customizeable function, which is the "domain-selector."

For the the first case statement, it's important to set a good default (i.e. `stats`) as most invocations of the program will be "`stats`" operations, and the action function itself can handle errors more clearly. For the second operation, it makes sense to produce an error, because if one there is no domain, there is no way to proceed.

See “`tycho-selector`”, which is an example “domain-selector” function:

```
tycho-selector(){
    PROJECT=projects
    LOG_TAG=tycho

    for argument in $ARG; do
        case "$argument" in
            ( ae ) queue=($queue al-edarian); shift ;;
            ( mars ) queue=($queue knowing-mars); shift ;;
            ( admin ) queue=($queue cyborg-admin); shift ;;
            ( gmg|mg ) queue=($queue mediagoblin);
                   WC_PATH=~/projects/mediagoblin/docs/source
                   shift ;;
            ( rhizome ) queue=($queue rhizome); shift
                       PROJECT_PATH=~/assemblage/rhizome/
                       WC_PATH=$PROJECT_PATH
                       BUILD_TYPE=wiki; EXTENSION=mdwn
            ;;
            ( assemblage|ass ) queue=($queue assemblage); shift
                              PROJECT_PATH=~/assemblage/
                              WC_PATH=$PROJECT_PATH
                              BUILD_TYPE=wiki; EXTENSION=mdwn
            ;;
            ( wikish|wiki )   queue=($queue wikish); shift
                             PROJECT_PATH=~/wikish/
                             WC_PATH=$PROJECT_PATH
                             BUILD_TYPE=wiki; EXTENSION=mdwn
            ;;
            ( * ) # continue silently ;;
        esac
    done

    action-handler $@;
}
```

The “domain-selector” functions set variables that describe the sub-projects in the domain that the “actions” use. The main reason to have separate projects is to be able to log statistics into separate files.

There are two constants set at the beginning of the “domain-selector.” Consider them and their purpose:

- `$PROJECT` describes the domain and unless overridden the directory in which all sub-projects reside.
- `$LOG_TAG` describes the string that prefixes log items when sending the log via XMPP.

The initial version of the script assumed each “domain” would refer to a group of projects that were sub-directories of a single “domain” folder. This is why the “`ae`”, “`mars`”, and “`admin`” projects only set the `$queue` variable. However, it’s not practical to force projects into such a rigid hierarchy, and as a result, these defaults can be overridden, which is what happens in the other sub-projects.

The key variables here are:

- `$queue` is an array that holds lists of sub-projects. For many workers, as long as you don’t mix Sphinx and Ikiwiki builds, you can specify multiple projects and, and `stl` will report or act on all of them.

  Note: While script uses this basic “queue” structure in a number of places, given the way that the shell (and I) have set up the variables, means that this functionality is not as robust as it ought to be. In the interests of reliability over correctness, the script should always “*do the right thing*” if you only specify one project in an invocation.

- `$WC_PATH` is the location of the source files. By default, the script will look for source files in "`$PROJECT/source/$queue.item`" (where `$queue.item` is an element in the `$queue` array.) Set `WC_PATH` to override this.
- `$PROJECT_PATH` is the path of the project files. Unless set this defaults to "`~/project`".
- `$BUILD_TYPE` specifies which "build method" to use. Current options are `wiki` for Ikiwiki instances (stored using git), and `sphinx`. The default is `sphinx`.
- `$EXTENSION` specifies the file extension of the files. Defaults to `rst`. Used to ensure that the word counts do not include extraneous files.

Set any or all of these variables in each case statement. You may now begin using `stl` to track the stats of your task.

`zsh stl` Internals (Legacy)

Note: This documentation reflects the legacy implementation of `stl`. See *Internal Details and Implementation* for the current implementation.

Synopsis This document provides an overview of the logic of `stl`, and discusses the code on a per-function basis, to provide a very fine grained idea of how the script operates. This information should be helpful if you wish to extend and add features to `stl`, or if you are having a problem and want to know about the underlying operation of the script.

Code Paths Beginning in the `main()` function (at the end of the file,) the first argument specifies the "domain," and the second argument specifies one of the program's of "workers." The worker sets the global `ACTION` variable, and the "domain" setting calls one of the "domain-selector" functions.

Each domain selector sets the default `PROJECT` and optionally `LOG_TAG` variables. Then, the selector loops over the remaining elements in the argument string to extract and set variables for each sub-project that you want to track with `stl`. While there are some global variables, the main operation of the domain-selector function adds the sub-projects to a `queue` variable that holds an array. When the selectors have set the all required variables for the project, and then call the `action-handler` function.

`action-handler` calls the appropriate worker function based on the value of the `ACTION` variable, and passes each worker the full argument string.

Abstractly, worker functions:

- begin by calling the `notify-init` function. `notify-init` is the first function in the file.
- Sets required variables if they are heretofore unset.
- Sets additional values from the argument string, if needed.
- Performs the required work.
- Sends output by way of the `notify()` function provided by the `notify-init` function.
- Exits.

`stl` contains the following worker functions with operations described below.

`compile-project`

`compile-project` provides the procedure to build a project.

The function begins by running `notify-init` and setting three variables if they are unset:

- `BUILD_REPORTS_LOC` which specifies the directory where the script writes the output of the build.

- `PROJECT_PATH` which sets the path of the project to "`~/$PROJECT`" unless overridden. Overrides of this variable typically affect the ability of `stl` to run multiple stats in one invocation.
- `BUILD_PATH` specifies either "`sphinx`" or "`wiki`", with `sphinx` being the default. The `wiki` option, works with Ikiwiki instances running in git.

Then, there are two embeded functions (described below) for building both types of project, followed by a `for` loop that builds all projects specified in the `queue`.

The loop begins by declaring and then creating the file for the `BUILD_REPORT` which consists of: the path from `BUILD_REPORTS_LOC`, a 32-bit UNIX timestamp, the name of the project, and a .txt extension. The loop also contains a `case` statement that calls an embeded function with the required arguments to build the project. When `stl` reaches the end of the `queue`, the program exits.

Note: Multiple project building does not work as efficiently as you'd expect: if you override `PROJECT_PATH`, for instance, the behavior is erratic.

Ideally, the domain selectors should declare a configuration array rather than a simple variable so that the builders and other operations can itterate over entire configuration objects rather than a list of sub-project names.

`build-sphinx-project`

This function assumes that your Sphinx projects use the default Makefile provided by `sphinx-quickinstall` or similar.

The main body of this function provides a for-loop around a case statement for each build type to call `make` as many times as necessary. When a build complete, the function calls the `notify` function to log the completion of the new build.

The function itself expects that its enclosing function will loop over it several time for each project, and is simple as a result.

`build-wiki`

`build-wiki` works for any Ikiwiki that use git as the storage system; however, it's general enough to use as the basis for any system that controls the build in a "post-commit" or "post-update" hook. The procedure is:

- Change directory to the `PROJECT_PATH`
- In git's staging area, remove all files had existed in the repository, but have been removed from the file system since the last commit. Then, add all uncommitted changes to the repository's staging area.
- Commit all changes, using the remainder of the argument string as the commit message.
- Pull in new changes using the "`--rebase`" option from the default remotes.
- Push all changes to the default remote repositories.

When the procedure is complete, the function calls the `notify` function to log the completion of the build.

`stats-base`

`stats-base` begins by calling the `notify-init` function, and setting four variables if they are unset:

- `BUILD_REPORTS_LOC` which specifies the directory where the script writes the output of the build.
- `PROJECT_PATH` which sets the path of the project to "`~/$PROJECT`" unless overridden. Overrides of this variable typically affect the ability of `stl` to run multiple stats in one invocation.
- `DATE_OUTPUT_FORMAT` Specifies a date output string used in the log messages when reporting the last build time. Translates the UNIX-timestamp into something readable. The default value is:

```
%A %B %d, %Y (%I:%m %p)
```

- EXTENSION which sets the file extension of the source files. The default value is `rst`.

Then a `for` iterates over the remaining arguments in the function, and adds values to an `outputs` array in a `case` statement. Possible settings here, are:

- `wc`, `word`, or `words`, adds a word count to the output queue.
- `build` or `builds` adds a report of the last completed build for which a build report exists.
- `force` which sets the "`FORCE`" environment variable. In the default operation `stats-base` will not output any data unless it has changed from the last time the operation ran. `force` overrides this.

There is one embeded function at this point ("`reporter`" documented below that provides a simple way for the main work of the function to pass information to the `notify` function when (and only when) there is something to report.

The main work of the function occurs in a nested `for` loop. The outer loop, iterates over the items in the `queue` array. It begins by setting the `WC_PATH` variable if it isn't already set (the default value, which works great for Sphinx projects is `~/$PROJECT_PATH/$item/source/` where `$item``is the iterated member of the ``queue`.

Note: Again there are limitations to this method, when overloading the "`WC_PATH` variable with running through the `queue` loop more than once. Although it's a bit more flexible than the `compile-function` behavior the implementation is still flawed.

Ideally, the domain selectors should declare a configuration array rather so that the builder and other operations can itterate over entire configuration objects rather than a list of sub-project names.

The second loop, iterates over the contents of the `outputs` variable, and contains a case statement. At the present time, the only stats are "word counts" (`wc`) that provide a count of the words in the project and "build reports" (`build`) that provide a note regarding the latest recorded build of the project.

By adding casses to the statement here and at the beginning of the `stats` function it's relatively easy to add different type of statistics reporting to `stl`.

The cases in this inner loop, sets two variables:

- `query` is effectively a "lambda" function, stored in a variable, enclosed in backtics (i.e. " `"), that calculates the total word count or the date of the relevant build.
- `message` which constructs the message that is ultimately passed to `notify` and sent to the log.

Finally, each inner-loop case calls the `reporter` function with four arguments. Continue to read the documentation of the reporter function and its use.

`report`

The `reporter` function ensures that `stl` logs only if the value of the statistic has changed since the last time the function ran, or the last time the function ran with a new value. Because it caches changed values in `/tmp` the stl will always report all statistics once following system reboot.

The function begins by setting more readable variable names for the four arguments:

- `type` holds to the kind of build (`outputs` from the `stats-base` function.) `type` identifies the statistic in the cache.
- `project` holds to the sub project, and also identifies the statistic in the cache. Do not confuse `project` with the global `PROJECT` variable, which is also used here.
- `data` holds the value of the statistic that `stats-base` reports.

- message holds the message that will passed to notify. In most cases, this actually overwrites the message variable which already exists with the same content but the reassignment adds clarity.

The function begins by making the directory /tmp/$PROJECT-stats/ if it doesn't already exists. stl stores its cache here, which allows independent caches for each domain. The cache is a directory of files named "$project-$type.

The main work of this function is in a 3-part if statement.

- When FORCE equals 1, the first part, passes the message variable to the notify function .
- When the cache for this static (type) doesn't exist for this project, the second part writes the value of data to the appropriate location in the cache.
- When the value of data is different from the value in the cache, the final part:
  - Removes the existing value from the cache,
  - creates a new cached value, and
  - passes the message to the notify function which logs the changed statistic.

There is no else statement, which would cover the case where the value in the cache is equal to the most recent value measured. This is likely the most common case. In this case stl outputs nothing, and continues running.

Note: While it may be possible to make the entire process more efficient by checking the cached value, earlier in the code path, the savings are minimal because stl would still have to run all of the same expensive operations (checking the new word count, etc.) the same number of times to ensure that the value hasn't changed.

---

stats-log

This simple function allows users to add arbitrary messages to their log files (by way of notify). The function begins by calling notify-init, setting the "type" variable, and cleaning up the array held by the ARG variable.

Next, an if statement detects an error condition if stats-log is running while notify is not in "log-file" mode.

Finally, a case statement passes formatted messages to notify depending on the value of type. Current types include: start for "clocking in," stop for "clocking out," and note for inserting arbitrary messages into the log.

This function returns an error if called with an unknown type value.

build-report

The build report function opens the most recent saved record of a build (as created by all invocations of the compile-project function, and displays them in the specified format.

The function begins by calling notify-init and setting the standard "BUILD_REPORTS_LOC variable if it not already set, and also setting the interface variable, which controls how the value is output.

The main operation of this function occurs within a for loop that iterates over members of the queue. The loop begins by declaring the LAST_LOG variable, which identifies the relevant build report for the current member of the queue. Then a case statement, selects the interface and passes LAST_LOG to this interface. If an interface case does not exist, the case statement produces an error and exits.

There are no known limitations to the ability of this function to handle multiple projects in one invocation, beyond the limitations created by interfaces themselves, provided that you only use one interface.

Additionally the notify-init function, which appears throughout stl, has the following operation:

`notify-init`

Many functions within `stl` call `notify-init`. The main purpose of this function is to determine when to log messages to the log file, and when to log messages by way of the xmpp bot. It accomplishes this by creating the `notify` according to the configuration and the current environment in which `stl` runs.

The function begins with a `case` statement. If the first argument to `notify-init` (accessed by way of the `output` action handler,) is "`xmpp`", then this statement removes the "`/tmp/$PROJECT-stats/log`" file. If the first argument to `notify-init` is "`logfile`" then this statement creates this file. If neither "`xmpp`" nor "`logfile`" is the first argument then the function continues.

Next, `notify-init` sets "`LOG_TAG`" to the value of `PROJECT` if `LOG_TAG` is empty. Then, it creates the "`log-file-notify`" function, which defines the log-file behavior, documented below.

Then a 3-part `if` statement defines the `notify` function. This sub function is then used throughout `stl`. The conditions are:

- When the Wireless (i.e. `wlan0`) interface does not exist, *and* the `eth0` interface does not have an IP address, define `notify` as a function calls `log-file-notify` passing `log-file-notify` all arguments.
- If the `log` file in the cache (i.e. "`/tmp/$PROJECT-stats/log`") exists, then `notify` is a function that calls `log-file-notify` passing `log-file-notify` all of the arguments that `notify` was called with.
- In all other cases, the `notify` function sends notifications as an instant message to the XMPP interface. It uses the "`xmpp-notify`" script included in the distribution with `stl`.

  `xmpp-notify` is a simple Perl script that sends it's argument string to a default XMPP address, using account credentials declared in that file.

Finally the function ends, with a conditional that sends a notification if the notification type has changed or been updated as a result of the initial case statement.

`log-file-notify`

The log-file-notify script performs the following operations:

- touches the `~/$PROJECT/stats.log` files and
- outputs an "`[HH:MM]`" time stamp followed by the message to the log file.

`zsh stl` Code and Organization (Legacy)

Note: This documentation reflects the legacy implementation of `stl`. See *Internal Details and Implementation* for the current implementation.

Note: This document explains my rationale for writing the zsh version of `stl` in the way that I did. While this document may be useful for you as a legend for understanding this script, the truth is that the zsh `stl` is only useful as a prototype, and needs to be rewritten in a more modular and maintainable system. – tychoish

Coding Goals The goals in writing this script were:

- To have something that separated the "worker" functions, from the configuration or settings. In the past I'd used a lot of environment variables set early in a script that I could use to customize the script. While this approach is functional, it's not very flexible, and it means that in order to use the script, you have to mangle a lot of variables *just right*. Putting all of this configuration into command-line options is similarly insuffcent for making usable or generalizable tools.

- To use `case` statements, and `for` loops, when possible to solve all of the repetitious code problems I was having in a previous version, without resorting to long, complex, and finicky conditionals.
- To set good defaults and use them when possible. This both makes the script easier to set up, and also makes it easier to add new projects. The assumption is that you have a "`~/projects`" directory beneath which are a collection of projects published/managed by Sphinx, with `rst` (or .txt) extension. the script can do other things, but that takes a (*bit*) more work.
- To run quickly. While there's a lot of crude elements and inefficiencies (particularly around blunt use of loops and not exiting if the options are verifiable invalid,) it rarely takes very long to run, and is able to finish quickly. There are minimal external dependencies, and most of the script stays in ZSH. There's some use of the **date** command, and a call to **wc** to return word counts, and two "`ifconfig | grep`" calls to see if networking is up to configure the notification.

File Organization `stl` begins with a number of reusable generic functions, and ends with a number of deployment specific functions that you'll need to customize so that the script knows where to find your files and work. See the *previous section* for more information on how to customize the script for personal usage, and the *code paths* section for more information on how the interpreter processes the code.

This section provides an inventory, as they appear in the file, of each function and it's general purpose.

- `notify-init` configures the notification system, and creates a `notify` function that either logs the output of script to a logfile (i.e. "`~/$PROJECT/stats.log`") or sends an xmpp message using an `xmpp-notify` script that is also included in the distribution.

  The script will use the log file if:

  – There is no network connection.

  – The file "`/tmp/$PROJECT-stats/log`" exists.

  Otherwise the script will send log messages to XMPP. Use a command in the following form to toggle XMPP/log file logging.

  ```
  stl [domain] output xmpp
  stl [domain] output logfile
  ```

  Remember that you *must* configure your domain before running this command.

- `action-handler` is a simple function that holds a case statement that calls another functions that does the actual work of the script.
- `stats-log` allows users to create a number of entries in the log with arbitrary messages, to provide the build reports and word counts with context. `stats-logs` requires that you specify a `start`, `stop`, or `note`, followed by a message.

  Output follows the system configuration for "logfile" or "xmpp" notifications.

- `build-report` opens or outputs the contents of the last build of the specified project or projects. Use these options to check for build errors.
- `stats-base` is the main worker function of `stl`, and it checks and returns word counts for projects primarily. Unless you include the "`force`" argument, this function will only return data *if* the value is *different* from the value when the script was last run. Every time the script runs, it checks against the last new value written in the "`/tmp/$PROJECTS-stats/` folder.
- `compile-project` triggers a rebuild or build of the project. It supports Sphinx (including sffms,) and Ikiwiki using git. The function writes the output of the build process to files in the "`/tmp/$PROJECT-stats`" folder, and logs completion via the `notify` function.

- `domain-selector` is a function that sets sub-project specific variables before calling the "`action-handler`" function.
- `main` the primary function of the code, and the only function that the main body of the script calls directly.

3 Resources

3.1 Stats Logger Reference

Glossary

sauron A notification system for emacs. See the Sauron EmacWiki Page for more information. The *sauron* script, included as a part of stl, provides a python and command line wrapper for sending Sauron notifications by way of `emacsclient`.

Additional Reference

- stl git repository
- stl on Github
- stl issue tracker

The latest version of this manual is also available for download in ePub and PDF formats:

- stl Manual, ePub
- stl Manual, PDF

4 Overview

`stl` (i.e. "status logger") is a tool for managing, maintaining, and logging work for writers. With `stl` you can track the aggregate word count of multi-file writing projects. Furthermore, `stl` include tools to build a more complete and more automated log of personal work and activity.

The initial implementation (circa Fall/Winter 2011) of `stl` was an over-complex shell script that was neither easy to maintain or particularly robust, that code, is still available in the git repository. The second implementation, in Python, is more modular, and significantly more flexible.

`stl` may be exactly the tool you need to:

- Record, manage, and view the output of your build tools (i.e. build reports.)
- Track word counts automatically to provide an overview of your progress both as you work and as you're attempting to track daily and hourly progress.
- Wrap your build tools to provide a more consistent interface.

This site documents both the use and the internal operation of `stl`, including the initial implementation for posterity and the current implementation.

5 Future Development

- Better validation of inputs.
- More clear interface for configuring outputs.
- Distribution in the Python Package Index.
- A daemon mode.
- More granular caching, potentially caching word counts on a per-file basis rather than forcing repeated `wc`. Potentially using `make` or `ninja`.
- Further decouple and create a more modular notification system. Currently you can use `stl` to send notifications to the command-line, to log files, and/or to Emacs, via sauron-mode.

Python Module Index

l

lnote, xii

s

sauron, viii
stl, x