Sealce API Documentation

Release 0.1

Christopher Patton

November 26, 2013

CONTENTS

SeaIce is an online, open-source, crowd-sourced dictionary for metadata terms. Users log in and contribute terms, vote on others', and leave comments. A reputation-based heuristic is used to estimate community consensus on these terms as a percentage. This **consensus score** is used in combination with a **term stability** metric to classify terms as being **vernacular**, **canonical**, or **deprecated**. The hope is that *SeaIce* will facilitate the evolution of a set of stable, canonical metadata terms, verified in social a ecosystem. We're calling the service a *metadictionary*. Check out the prototype at seaice.herokuapp.com.

SeaIce is written in Python on top of a PostgreSQL database and uses Flask for the web framework. Here you can find the complete documentation of the API.

CHAPTER

ONE

A MODEL FOR CROWD-SOURCED DICTIONARY TERMS

1.1 Scoring

The *SeaIce* metadictionary classifies terms in three categories. *Vernacular* terms are those for which there is no community consensus. The debate may be centered around the term's definition or the usefulness of the term itself. The owner has the oppurtunity to modify the defintion or term itself; therefore, vernacular terms are subject to change as the debate proceeds. *Canonical* terms are those whose definitions stabilize over time and upon which the community largely agrees. Finally, *deprecated* terms are stable terms that the community has largely deemed to not be useful, or for which there is a better alternative. The goal of *SeaIce* is to evolve a set of stable, canonical terms within the context of a reputation-based social ecosystem by promoting vernacular terms to the canon (shorthand meaning the canonical class) and deprecating those deemed not useful. I propose a scoring and classification system that automates this process for the online metadictionary. In this document I refer to community as the body of *SeaIce* users.

The first step is to quantify consensus. The most obvious definition is the percentage of the community that finds the term and its definition useful. We'll call this score. Let u be the number of up-votes and d be the number of down-votes. If every user casts a vote, then a term's score is simply

$$S = \frac{u}{u+d}$$

We could declare a term canonical if the term stabilizes (the owner makes no modifications for some period of time) and the score remains above some threshold (for the same time period). Similarly, when a stable term's score falls below some threshold (deprecation). However, and this is the central observation, we can't assume that every user will vote on every term as the dictionary increases in size. We therfore need to introduce a heuristic for community consensus based on user repuation. This makes it possible for domain-experts to promote useful terms that don't need to be verified by the entire community. At the sametime, it is critical that substantial dissent can prevent a term from entering the canon.

A user's reputation is a positive integer value that is initially seeded in the database. A user gains reputation by participating in the community (voting and commenting on terms) and contributing terms that enter the canon.[1] Let R_i be the reputation acquired by user i and let R be the total reputation of the users who have voted on a particular term. Let $r_i = R_i/R$ for each user i who voted on the term. Let t be the total number of users in the community and v be the number of votes cast such that $d = v - u$. The weight w_i of a user's vote is based on his or her reputation in the following relationship:

$$w_i = 1 + r_i \cdot (t - v)$$

The influence of the user's reputation decreases linearily as the number of voters for the term approaches the total number of users. However, everyone is guarenteed one vote. Now let $U = \{w_1, w_2, \ldots w_u\}$ be the set of weighted

[1] Note that the actual rules for gaining reputation within the community have yet to be defined.

up-votes and $D = \{v_1, v_2, \ldots v_d\}$ the set of weighted down-votes. Substituting these values into the equation for S, we have

$$S = \frac{\sum_U w_i}{\sum_U w_i + \sum_D v_i}$$

Pulling out some terms from the sums and substituting $w_i = 1 + U_i/R \cdot (t - v)$ for each up voter i and $v_i = 1 + D_i/R \cdot (t - v)$ for each down voter i:

$$\begin{aligned} &= \frac{u + (t - v) \cdot \sum_U U_i/R}{u + d + (t - v) \cdot \left(\sum_U U_i/R + \sum_D D_i/R \right)} \\ &= \frac{u + (t - v) \cdot \sum_U U_i/R}{u + d + (t - v) \cdot 1} \\ S &= \frac{u + \sum U_i/R \cdot (t - v)}{t} \end{aligned}$$

This equation has the property that as v approaches t, votes become increasingly fair; i.e., S approaches $u/(u + d)$ Notice as well that we don't use a user's community-wide reputation percentage; this gives every term an equal chance of reaching a stable, canonical state despite the proposer's reputation. At the same time, it allows users with high reputation to debunk bad ones immediately with a single down-vote.

An advantage of this equation is that it's easy to compute. To collect the reputations of users voting on a particular term would normally require a join on the *Users* and *Tracking* tables. However, we need only keep track of u, v, t, R and $\sum U_i$ for each term. Then when a user casts or changes a vote, we update these values and calculate the new score in constant time. The other instance when it's necessary to update a term score is when a user's reputation changes; this can be done in linear time in terms of the number of votes cast.

We can refine this relationship a bit more if a lineear function with respect to v isn't adequate in practice. For instance, it may be useful to scale down reputation exponentially:

$$w_i = 1 + \frac{r_i \cdot t}{v}$$

Howver, we'll have a better idea of how to do this once we get users and a sense of the distribution of reputation values.

1.2 Classification

Once a vernacular term stabilizes, we decide whether to include it in the canon. I suggest two conditions for this property: (1) the owner hasn't modifed the term or its defnition for some predefined time interval and (2) the rate at which the term's score changes has dropped below some threshold close to zero for the same time interval.

1.2.1 Stability

Term stability is recalculated each time the consensus score is recalculated. To calculate the rate of change, we store the time at which the consensus score S was computed. Given previous score (S_0, t_0) and current score (S, t), the rate of change for this interval is:

$$\Delta S = \frac{S - S_0}{t - t_0}$$

Let t_S be the time point when the term became stable, or `nil` if the term is unstable. Let ϵ be stability error, or the amount that S can fluctuate and still be considered stable. If $|\Delta S| < \epsilon$ and $t_S =$ `nil`, then set $t_S \leftarrow t$; if $|\Delta S| > \epsilon$, then set $t_S \leftarrow$ `nil`; otherwise, S is stable and t_S remains unchanged.

Now, let T be the interval of time required for stability and t be the time at which we wish to classify a term. The following conditions are perscribed:

1. $t - t_M > T$, where t_M is the time at which the term was modified, and
2. $t_S \neq \texttt{nil}$ and $t - T - S > T$ **or** $t - t_0 > T$.

1.2.2 Promotion and demotion

If a term has stabilized, we use its consensus score S to classify it. We need only to decide on reasonable threshold values. To start, I suggest a stabile term should require 75% consensus to be promoted to the canon. If consnesus drops below 25% after it stabilizes, it should be deprecated. Otherwise, the term should remain in the vernacular.

If a term is canonical or deprecated and becomes unstable, it does not immediately reenter the vernacular. Instead, we wait until it restabilizes and classify it based on the resulting score.

We expect that terms will be deprecated by the community when there is a viable alternative. In this case, the standard practice should be to include a reference to the new term in the old's definition.

1.3 Software components

In Chicago, we were brainstorming about a consistency issue related to voting and the term being modified. The solution we discussed was to require a call-for-votes and a score reset when the owner modifies the term. It was remarked that this is a bit clunky. An alternative approach would be to notify users with a term they're interested in get's updated, promoted, or demoted.

1.3.1 Notifications

First, we will allow users to "star" and "unstar" terms that they're interested in. On the homepage of the SeaIce website, we could have a notification page, much like facebook, which provides updates on terms you own and terms your're tracking:

1. User X has commented on your term A
2. User X has updated term B
3. Term B has been promoted to the canon
4. Term C has been deprecated
5. Term C has reentered the vernacular

Once you see the notifcation, you can respond by commenting, changing your vote, unstarring the term, etc. Because term stabilization depends not only on the owner not editing the term but on the score's rate of change, there is no need to reset the score on modification.

1.3.2 Browsing and search

We have a number of distinct goals in term discovery: most stable (canonical, lots of consensus), most recently contributed, and most frequently discussed to name a few. In the manner of stackoverflow, we'll provide a listing of terms per criterion. Making these available will be very useful for contribution. The default search query ranks pages first by relevance to the query, second by stability and classificaition, and third by consensus score.

CHAPTER

TWO

THE SEAICE PACKAGE

seaice is comprised of tools and datastructures on which the front ends are built. Distributed with *SeaIce* are two top level Python programs with very clever names:

- sea – Command line tool for setting up, tearing down, and other wise interacting with the database in various ways.
- ice – Web UI based on Flask for *SeaIce*. This facilitates user interactions with the database by serving various *GET/POST* requests.

See the section on top-level programs **[ref]** for more details. The *SeaIce* API should be imported into the application namespace with:

```
import seaice
```

Included in seaice.* are user data structures for Flask-login, get_config() which handles local database configuration, and the various output formatters (seaice.pretty.*). The most important object, seaice.SeaIceFlask is used to build a Flask-based web interface for *SeaIce*.

2.1 class SeaIceFlask

class seaice.SeaIceFlask.**SeaIceFlask**(*import_name*, *static_path=None*, *static_url_path=None*, *static_folder='html/static'*, *template_folder='html/templates'*, *instance_path=None*, *instance_relative_config=False*, *db_user=None*, *db_password=None*, *db_name=None*)

Bases: flask.app.Flask

A subclass of the main Flask interface. This includes various live data structures used in the web interface, as well as a pool of database connectors. All features in the SeaIce API that the top-level progams make use of are available as attributes of this class.

Parameters

- **user** (*str*) – Name of DB role (see seaice.SeaIceConnector.SeaIceConnector for default behavior).
- **password** (*str*) – User's password.
- **db** (*str*) – Name of database.

SeaIceFlask.**dbPool**

Type seaice.ConnectorPool.SeaIceConnectorPool

SeaIceFlask.**userIdPool**

Type seaice.IdPool

SeaIceFlask.**termIdPool**

Type seaice.IdPool

SeaIceFlask.**commentIdPool**

Type seaice.IdPool

SeaIceFlask.**SeaIceUsers**

Type seaice.user.User dict

seaice.SeaIceFlask.**MAX_CONNECTIONS = 1**
The number of DB connections that will be instantiated.

2.2 Classes and modules

In this index, you can find documentation of the various thingys that comprise the package.

2.2.1 class SeaIceConnector – interface for PostgreSQL DB

This is the interface for the *SeaIce* database. It is assumed that a PostgreSQL database has already been configured. The following queries are implemented:

- Create the DB schema SI and the various table and triggers (createSchema()).
- Drop SI , tables, and triggers (dropSchema()).
- Insert, remove, and update terms, users, and comments.
- Cast vote and track terms.
- Calculate term consensus and stability.
- Inset and delete notifications.
- Import/export the DB.

class seaice.SeaIceConnector.**SeaIceConnector**(*user=None*, *password=None*, *db=None*)
Connection to the PostgreSQL database.

This object is capable of either connecting to a local database (specified by the parameters) or a remote database specified by the environment variable DATABASE_URL if no paramters are given. This is to support Heroku functionality. For local testing with your Heroku-Postgres based database, do

```
export DATABAE_URL=$(heroku config:get DATABASE_URL) && ./ice
--config=heroku
```

Parameters

- **user** (*str*) – Name of DB role.
- **password** (*str*) – User's password.
- **db** (*str*) – Name of database.

Export(*table*, *outf=None*)
Export database in JSON format to *outf*. If no file name provided, dump to standard out.

Parameters

- **table** (*str*) – Name of table.
- **outf** (*str*) – Filename to output table to.

Import(*table*, *inf=None*)
Import database from JSON formated *inf*.

Parameters

- **table** (*str*) – Name of table.
- **inf** (*str*) – File name from which to import.

castVote(*user_id*, *term_id*, *vote*)
Cast or change a user's vote on a term. Return the term's new consensus score.

Parameters

- **user_id** (*int*) – User ID.
- **term_id** (*int*) – Term Id.
- **vote** (*int*) – +1, 0, or -1.

Return type float

checkTermConsistency(*term_id*)
Check that a term's consensus score is consistent by scoring it the hard way. Update if it wasn't.

Return type bool

checkTracking(*user_id*, *term_id*)
Check tracking.

Return type bool

classifyTerm(*term_id*)
Check if term is stable. If so, classify it as being *canonical*, *vernacular*, or *deprecated*. Update term and return class as string.

Returns 'canonical', 'vernacular', or 'deprecated'.

Return type class

commit()
Commit changes to database made while the connection was open. This should be called before the class destructor is called in order to save changes. It should be called freuqently in a mult-threaded environment.

con = None
The PostgreSQL database connector provided by the psycopg2 package.

createSchema()
Create the `SI` schema for SeaIce with the tables `SI.Users`, `SI.Terms`, `SI.Comments`, `SI.Tracking` (vote and star), and various triggers; create `SI_Notify` with `SI_Notify.Notify`.

dropSchema()
Drop `SI` and `SI_Notify` schemas.

getAllNotifications()
Return an iterator over `SI_Notify.Notify`.

Return type dict iterator

getAllTerms(*sortBy=None*)
Return an iterator over `SI.Terms`.

Parameters **sortBy** (*str*) – Column by which sort the results in ascending order.

Return type dict iterator

getAllUsers()
Return an iterator over `SI.Users`.

Return type dict iterator

getByTerm(*term_string*)
Search table by term string and return an iterator over the matches.

Parameters **term_string** (*str*) – The exact term string (case sensitive).

Return type dict or None

getComment(*id*)
Get comment by ID.

Parameters **id** (*int*) – Comment ID.

Return type dict or None

getCommentHistory(*term_id*)
Return a term's comment history, ordered by creation date.

Parameters **term_id** (*int*) – Term ID.

Return type dict iterator

getTerm(*id*)
Get term by ID.

Parameters **id** (*int*) – Term ID.

Return type dict or None

getTermStats()
Return the various parameters used to calculate consensus and stability for each term.

Return type dict iterator

getTermString(*id*)
Get term string by ID.

Parameters **id** (*int*) – Term ID.

Return type str or None

getTermsByTracking(*user_id*)
Return an iterator over terms tracked by a user (terms that the user has starred).

Parameters **user_id** (*int*) – ID of the user.

Return type dict iterator

getTermsByUser(*user_id*)
Return an iterator over terms owned by a user.

Parameters **user_id** (*int*) – ID of the user.

Return type dict iterator

getTime()
Get *T_now* timestamp according to database. This is important when the SeaIce database is deployed to some anonymous server farm.

Return type datetime.datetime

getTrackingByTerm(*term_id*)
Return an iterator over users tracking a term.

Parameters **term_id** – ID of term.

Returns User rows.

Return type dict iterator

getUser(*id*)
Get User by ID.

Parameters **id** (*int*) – User ID.

Return type dict or None

getUserByAuth(*authority*, *auth_id*)
Get user identified by an authentication ID. It's assumed that this ID is unique in the context of a particular authority, such as Google.

TODO: originally I planned to use (authority, auth_id) as the unique constraint on the SI.Users table. My guess is that most, if not all services have an associated email address. The unique constraint is actually the user's email. This method should be replaced.

Parameters

- **authority** (*str*) – Organization providing authentication.
- **auth_id** (*str*) – Authentication ID.

Returns Internal surrogate ID of user.

Return type int or None

getUserNameById(*id*, *full=False*)
Get username by ID.

Parameters

- **id** (*int*) – User ID.
- **full** (*bool*) – Get full name

Returns If *full* is set, then return the first and last name of the user. Otherwise, just return the first name.

Return type str or None

getVote(*user_id*, *term_id*)
Get user's vote for a term.

Parameters

- **user_id** (*int*) – User ID.
- **term_id** (*int*) – Term ID.

Returns +1, 0, or -1.

Return type int or None

insertComment(*comment*)
Insert a new comment into the database and return ID.

Parameters **comment** (*dict*) – New comment as dictionary. Default values will be used for ommitted columns.

Return type int

insertNotification(*user_id*, *notif*)
Insert a notification.

Parameters **notif** (*seaice.notify.BaseNotification*) – Notification.

insertTerm(*term*)
Add a term to the database and return the term's ID.

Parameters **term** (*dict*) – Term row to be inserted. Default values will be used for omitted columns.

Returns ID of inserted row. If term['id'] isn't assigned, the next available ID in the sequence is given.

Return type int or None

insertTracking(*tracking*)
Insert a tracking row, skipping if (term_id, user_id) pair exists.

Parameters **tracking** (*dictionary*) – Expect dictionary with keys 'user_id', 'term_id', and 'vote'.

Returns (term_id, user_id)

Return type (int, int) or None

insertUser(*user*)
Insert a new user into the table and return the new ID.

Parameters **user** (*dict*) – Default values are used for any omitted columns.

Return type int or None

postScore(*term_id*, *U*, *D*)
Postscore term. Input the reputations of up voters and down voters and compute the consensus score. Update the term row as a side-affect.

Parameters

- **U** (*int –> int*) – Up voters.
- **D** (*int –> int*) – Down voters.

preScore(*term_id*)
Prescore term. Returns a tuple of dictionaries (User.Id -> User.Reputation) of up voters and down voters (*U, D*).

Parameters **term_id** (*int*) – Term ID.

Returns (*U, D*)

Return type ((*int –> int*), (*int –> int*))

removeComment(*id*)
Remove comment and return ID.

Parameters **id** (*int*) – Comment ID.

Return type int or None

removeNotification(*user_id*, *notif*)
Remove a notification.

Parameters **notif** (*seaice.notify.BaseNotification*) – Notification.

removeTerm(*id*)
Remove term row from the database.

Parameters **id** (*int*) – Term Id.

Returns ID of removed term.

Return type int or None

search(*string*)
Search table by term_string, definition and examples. Rank results by relevance to query, consensus, and classificaiton.

Parameters **string** (*str*) – Search query.

Return type dict list

trackTerm(*user_id*, *term_id*)
User has starred a term. Return True if a row was inserted into the Tracking table, False if not.

Return type bool

untrackTerm(*user_id*, *term_id*)
Untrack term.

updateComment(*id*, *comment*)
Update term comment. Note that user authentication is handled up stream!

Parameters

- **id** (*dict*) – Comment ID.
- **comment** – Expects at least the key 'comment_string' with string value.

updateTerm(*id*, *term*)
Modify a term's term string, deifnition and examples. Note: term ownership authenticated upstream!

Parameters

- **id** (*int*) – Term ID.
- **term** (*dict*) – Dictionary containing atleast the keys 'term_string', 'definition', and 'examples' with string values.

updateUser(*id*, *first*, *last*)
Update user's name.

Parameters

- **id** (*int*) – User ID.
- **first** (*str*) – First name.
- **last** (*str*) – Last name.

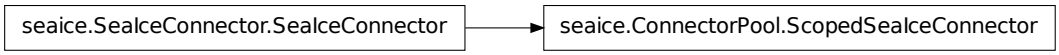
updateUserReputation(*id*, *rep*)
Set reputation of user. This triggers an update of the consensus score and term stability. Commit updates immediately.

Parameters

- **id** (*int*) – User ID.

- **rep** (*int*) – New reputation score.

class ScopedSeaIceConnector



class seaice.ConnectorPool.**ScopedSeaIceConnector**(*pool*, *db_con*)
Bases: seaice.SeaIceConnector.SeaIceConnector

A SeaIce DB Connector which is released to the pool it from whence it came when it goes out of scope. This type of connector is produced by seaice.ConnectorPool.SeaIceConnectorPool.getScoped() and should not be used directly.

Parameters

- **pool** (*seaice.ConnectorPool.SeaIceConnectorPool*) – The pool from which this connector originates. When the destructor is called, the connection is enqueued int to the pool.
- **db_con** (*seaice.SeaIceConnector.SeaIceConnector*) – The connector.

2.2.2 class ConnectorPool

This class implements a thread-safe DB connector pool in the typical way. ConnectorPool is the generic base class for SeaIceConnectorPool which is should be used in practice.

class seaice.ConnectorPool.**ConnectorPool**(*Connector*, *count=20*, *user=None*, *password=None*, *db=None*)

A thread-safe connection pool.

TODO: Make this an actual queue, not a stack. Nomenclature is imporant sometimes.

dequeue()
Get connector.

Return type seaice.SeaIceConnector.SeaIceConnector

enqueue(*db_con*)
Release connector.

Parameters **db_con** (*seaice.SeaIceConnector.SeaIceConnector*) – The connector.

class seaice.ConnectorPool.**SeaIceConnectorPool**(*count=20*, *user=None*, *password=None*, *db=None*)
Bases: seaice.ConnectorPool.ConnectorPool

A thread-safe connection pool which can produce scoped SeaIce connectors.

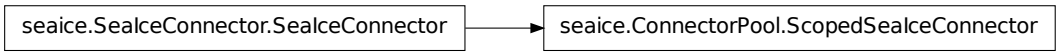
Parameters

- **count** (*int*) – Size of the pool.
- **user** (*str*) – Name of DB role (see seaice.SeaIceConnector.SeaIceConnector for default behavior).
- **password** (*str*) – User's password.

- **db** (*str*) – Name of database.

getScoped()
Return a scoped connector from the pool.

Return type seaice.SeaIceConnector.SeaIceConnector



class seaice.ConnectorPool.**ScopedSeaIceConnector**(*pool*, *db_con*)
Bases: seaice.SeaIceConnector.SeaIceConnector

A SeaIce DB Connector which is released to the pool it from whence it came when it goes out of scope. This type of connector is produced by seaice.ConnectorPool.SeaIceConnectorPool.getScoped() and should not be used directly.

Parameters

- **pool** (*seaice.ConnectorPool.SeaIceConnectorPool*) – The pool from which this connector originates. When the destructor is called, the connection is enqueued int to the pool.
- **db_con** (*seaice.SeaIceConnector.SeaIceConnector*) – The connector.

2.2.3 class IdPool

This class implements a surrogate identifier pool for the DB tables SI.Terms, SI.Users, and SI.Users.

class seaice.IdPool.**IdPool**(*db_con*, *table*)

A thread-safe object for producing and consuming table row IDs within a particluar context, i.e. SI.Terms, SI.Users, and SI.Comments. When initialized, an instance queries the table for all assigned IDs in ascending order. Continuous regions of unassigned IDs are found and added to the pool. The highest assigned ID is noticed so that when the pool is empty, the producer function returns the next highest avaialble.

TODO This could be made more space-efficient by combining contiguous free IDs into ranges.

Parameters

- **db_con** (*seaice.SeaIceConnector.SeaIceConnector*) – Connection to the SeaIce database.
- **table** (*str*) – Name of the table for which a pool will be created. The table should have a column of surrogate ID scalled "id".

ConsumeId()
Consume the next available ID.

Return type int

GetNextId()
Get the next ID without consuming it (look ahead).

Return type int

ReleaseId(*id*)
Release and ID back into the pool (produce)

Parameters **id** (*int*) – Surrogate ID.

2.2.4 the user module

This module implements the live data structures needed for Flask as well as notifications. See related topic:

The auth module

This module contains the various data structures used for authenticating sessions. *SeaIce* doesn't handle user accounts directly; instead, it utilizes third-party authentication services to make it easy for users log on with existing accounts. So far, only Google account authentication is implemented. To add other authenticator services – Facebook, OpenID, and StackOverflow to name a few – it will be necessary to restructure this code a bit (noted below). In addition, this will require changing `ice` a fair amount.

seaice.auth.**accessible_by_group_or_world**(*file*)
Verify the permissions of configuration file. *Contributed by Nassib Nassar.*

Parameters **file** (*str*) – File name.

Return type bool

seaice.auth.**get_config**(*config_file='/home/christopher/.seaice'*)
Get local db configuration. *Contributed by Nassib Nassar.*

Structure with DB connection parameters for particular roles. See the top-level program *ice* for example usage.

Parameters **config_file** (*str*) – File Name.

Return type dict

seaice.auth.**REDIRECT_URI = '/authorized'**
Variable prescribed by the Google OAuth API. **TODO:** To accomadate other authentication services, change this to '/authorized/google' (also on code.google.com/apis/console).

seaice.auth.**GOOGLE_CLIENT_ID = 'MISSING'**
Google OAuth credentials, client ID. These values are generated from code.google.com/apis. Note that the actual values for the online deployment of SeaIce aren't published.

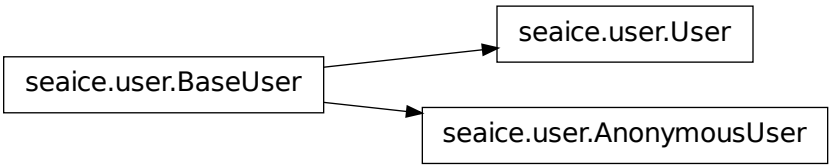
seaice.auth.**GOOGLE_CLIENT_SECRET = 'MISSING'**
Google OAuth credentials, client secret.

seaice.auth.**google = <flask_oauth.OAuthRemoteApp object at 0x388f190>**
Represents a remote application.

Parameters

- **oauth** – the associated `OAuth` object.
- **name** – then name of the remote application
- **request_token_url** – the URL for requesting new tokens
- **access_token_url** – the URL for token exchange
- **authorize_url** – the URL for authorization
- **consumer_key** – the application specific consumer key

- **consumer_secret** – the application specific consumer secret
- **request_token_params** – an optional dictionary of parameters to forward to the request token URL or authorize URL depending on oauth version.
- **access_token_params** – an option diction of parameters to forward to the access token URL
- **access_token_method** – the HTTP method that should be used for the access_token_url. Defaults to `'GET'`.



class `seaice.user.`**`BaseUser`**(*id*, *name*)

Base class for users. Users are used in Flask fo storing information about active and authenticated user sessions. This implements. BaseUser implements the basic routines needed for the Flask login manager. See the Flask-Login documenttation for details.

Parameters

- **id** (*int*) – User's surrogate ID in the database.
- **name** (*str*) – First name of user (for display purposes).

`get_id`()
Required by Flask-Login.

Return type unicode str

`is_active`()
Required by Flask-Login.

`is_anonymous`()
Required by Flask-Login.

`is_authenticated`()
Required by Flask-Login.

class `seaice.user.`**`User`**(*id*, *name*)
Bases: `seaice.user.BaseUser`

Handler for authenticated sessions.

`getNotificationsAsHTML`(*db_con*)
Get notifications as HTML.

Create a link next each one which, when clicked, calls `User.remove()`.

Parameters **db_con** (*seaice.SeaIceConnector.SeaIceConnector*) – DB connection.

Returns HTML-formatted string.

notify(*notif*, *db_con=None*)
Receive notification from another user.

If *db_con* is specified, insert the notifiation into the databse. The reason for having this option is that not all deployments of SeaIce will need to have persistent notifications. This method is thread-safe, as it's possible to receive many notifications simultaneously.

Parameters

- **notif** (*seaice.notify.BaseNotification*) – Notification instance.
- **db_con** (*seaice.SeaIceConnector.SeaIceConnector*) – DB connection.

remove(*i*, *db_con=None*)
Remove notification at index *i* from the list.

If *db_con* is specified, then remove notification from the database. This method is thread-safe. **NOTE:** If the order of notifications changed sometime between this point and the last time the HTML page was generated, the wrong notification will be removed.

Parameters

- **i** (*int*) – Index of notification.
- **db_con** (*seaice.SeaIceConnector.SeaIceConnector*) – DB connection.

class seaice.user.**AnonymousUser**
Bases: seaice.user.BaseUser

Handler for non-authenticated sessions.

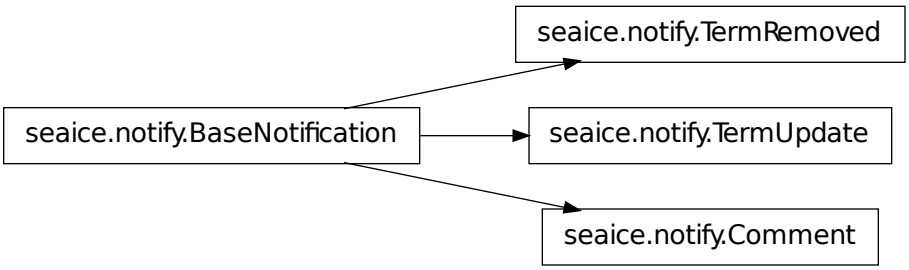
id = None
When polled, return None for ID, as prescribed by Flask-Login.

logged_in = None
When asked if you are logged in, always respond with **no**.

2.2.5 The notify module

Notifications are generated when important changes to the *SeaIce* dictionary are made and relevant users neeed to be informed. For example, when the owner of a term modifies the definition, users who have voted on the term must be notified so that they can recast their vote if necessary. Another useful application of notifications is when users comment on your term. BaseNotification comprises the basis and requires at a minimum the time when the event occured and the surrogate ID of the term it pertains to. A few basic notifications are implemented here; undoubtedly more will prove to be useful.

Notifications are handled by the function seaice.user.User.notify(). There is a table and schema (SI_Notify.Notfiy) in the *SeaIce* database for persistent storage of notifications. insertNotifciation() and removeNotification() are called in seaice.user.User.notify() for handling insertion and deletion respectively. Note that this is meant to be optional, however; omitting the *db_con* parameter will skip this.



class seaice.notify.**BaseNotification**(*term_id*, *T_notify*)
Base class for notifications in the SeaIce web interface Each sub class should implement __init__(), __str__(), and getAsHTML(db_con) This data structure only stores surrogate keys for users, terms, and comments. so that a notification is never inconsistent. As a result, getAsHTML causes a query to the DB.

Parameters

- **term_id** (*int*) – Term ID.
- **T_notify** (*datetime.datetime*) – The time at which the notification was produced.

getAsHTML(*db_con*)
Return an HTML-formatted notification string. To avoid dereferencing something that has been deleted (term, comment, user, etc.), return None if the database has no results.

Parameters **db_con** (*seaice.SeaIceConnector.SeaIceConnector*) – Connection to database.

Return type str or None

class seaice.notify.**Comment**(*term_id*, *user_id*, *T_notify*)
Bases: seaice.notify.BaseNotification

Notification object for comments.

Parameters

- **term_id** (*int*) – Term ID.
- **user_id** (*int*) – ID of the user has commented on a term.
- **T_notify** (*datetime.datetime*) – The time at which the notification was produced.

class seaice.notify.**TermUpdate**(*term_id*, *user_id*, *T_notify*)
Bases: seaice.notify.BaseNotification

Notification object for term updates.

Parameters

- **term_id** (*int*) – Term ID.
- **user_id** (*int*) – ID of the user who has updated the term.
- **T_notify** (*datetime.datetime*) – The time at which the notification was produced.

class seaice.notify.**TermRemoved**(*user_id*, *term_string*, *T_notify*)
Bases: seaice.notify.BaseNotification

Notification object for term removals.

Parameters

- **user_id** (*int*) – ID of the user who has updated the term.
- **term_string** (*str*) – Term string before the term was deleted. We can't use the ID since it has been removed from the database.
- **T_notify** (*datetime.datetime*) – The time at which the notification was produced.

2.2.6 The `pretty` module

This module implements various output-formatters that I've have been found useful for creating web pages and console output.

`seaice.pretty.`**`printPrettyDate`**(*T*)
Format output of a timestamp.

If a small amount of time has elapsed between *T_now* and *T*, then return the interval. **TODO:** This should be localized based on the HTTP request.

Parameters **T** (*datetime.datetime*) – Timestamp.

Return type str

`seaice.pretty.`**`processTags`**(*db_con*, *string*)
Process tags in DB text entries into HTML.

Parameters

- **db_con** (*seaice.SeaIceConnector.SeaIceConnector*) – DB connection.
- **string** – The input string.

Returns HTML-formatted string.

Plain text

`seaice.pretty.`**`printAsJSObject`**(*rows*, *fd=<open file '<stdout>', mode 'w' at 0x2ad26eee01e0>*)
Print table rows as JSON-formatted object.

Parameters

- **rows** (*dict iterator*) – Table rows.
- **fd** (*file*) – File descriptor to which to output the result (default is sys.stdout).

`seaice.pretty.`**`printParagraph`**(*db_con*, *text*, *leftMargin=8*, *width=60*)
Format some text into a nice paragraph for displaying in the terminal. Output the result directly to sys.stdout.

Parameters

- **db_con** (*seaice.SeaIceConnector.SeaIceConnector*) – DB connection.
- **text** (*str*) – The paragraph.
- **leftMargin** (*int*) – Number of spaces to print before the start of each line.
- **width** – Number of characters to print per line.

`seaice.pretty.`**`printTermsPretty`**(*db_con*, *rows*)
Print term rows to terminal.

Parameters

- **db_con** (*seaice.SeaIceConnector.SeaIceConnector*) – DB connection.
- **rows** (*dict iterator*) – Table rows.

HTML

seaice.pretty.**printTermAsHTML**(*db_con*, *row*, *user_id=0*)
Format a term for the term page, e.g. this.

This is the main page where you can look at a term. It includes a term definition, examples, a voting form, ownership, and other stuff.

Parameters

- **db_con** (*seaice.SeaIceConnector.SeaIceConnector*) – DB connection.
- **row** (*dict*) – Term row.
- **user_id** (*int*) – Surrogate ID of user requesting the page. Defaults to 0 if session is unauthenticated.

Returns HTML-formatted string.

seaice.pretty.**printTermsAsHTML**(*db_con*, *rows*, *user_id=0*)
Format search results for display on the web page.

Parameters

- **db_con** (*seaice.SeaIceConnector.SeaIceConnector*) – DB connection.
- **row** (*dict iterator*) – Term rows.
- **user_id** (*int*) – Surrogate ID of user requesting the page. Defaults to 0 if session is unauthenticated.

Returns HTML-formatted string.

seaice.pretty.**printTermsAsBriefHTML**(*db_con*, *rows*, *user_id=0*)
Format table rows as abbreviated HTML table, e.g. this.

Parameters

- **db_con** (*seaice.SeaIceConnector.SeaIceConnector*) – DB connection.
- **row** (*dict iterator*) – Term rows.
- **user_id** (*int*) – Surrogate ID of user requesting the page. Defaults to 0 if session is unauthenticated.

Returns HTML-formatted string.

seaice.pretty.**printTermsAsLinks**(*rows*)
Print terms as a link list (pun intended).

Parameters **rows** (*dict iterator*) – Table rows.

Returns HTML-formatted string.

seaice.pretty.**printCommentsAsHTML**(*db_con*, *rows*, *user_id=0*)
Format comments for display on the term page.

Parameters

- **db_con** (*seaice.SeaIceConnector.SeaIceConnector*) – DB connection.
- **row** (*dict iterator*) – Comment rows.
- **user_id** (*int*) – Surrogate ID of user requesting the page. Defaults to 0 if session is unauthenticated.

Returns HTML-formatted string.

CHAPTER

THREE

TERM SCORING AND CLASSIFICATION

seaice.SeaIceConnector.**calculateConsensus**(*u*, *d*, *t*, *U_sum*, *D_sum*)

Calcluate consensus score. This is a heuristic for the percentage of the community who finds a term useful. Based on the observation that not every user will vote on a given term, user reptuation is used to estimate consensus. As the number of voters approaches the number of users, the votes become more equitable. (See doc/Scoring.pdf for details.)

Parameters

- **u** – Number of up voters.
- **d** – Number of donw voters.
- **t** – Number of total users.
- **U_sum** – Sum of up-voter reputation.
- **D_sum** – Sum of down-voter reputation.

seaice.SeaIceConnector.**calculateStability**(*S*, *p_S*, *T_now*, *T_last*, *T_stable*)

Calculate term stability, returning the time point when the term become stable (as a datetime.datetime) or None if it's not stable. This is based on the rate of change of the consensus score:

```
delta_S = (S - P_s) / (T_now - T_last)
```

Parameters

- **T_now** (*datetime.datetime*) – Current time.
- **T_last** (*datetime.datetime or None*) – Time of last consensus score calculation.
- **T_stable** – Time since term stabilized.
- **S** (*float*) – Consensus score at T_now.
- **p_S** (*float*) – Consensus score at T_last.

seaice.SeaIceConnector.**stabilityError = 0.1**

The maximum varation in consensus allowed for score to be considered stable (S/hour).

seaice.SeaIceConnector.**stabilityFactor = 3600**

Convert seconds (datetime.timedelta.seconds) to hours.

seaice.SeaIceConnector.**stabilityInterval = 4**

Interval (in hours) for which a term must be stable in order to be classified.

seaice.SeaIceConnector.**stabilityConsensusIntervalHigh = 0.75**

Classify stable term as canonical.

seaice.SeaIceConnector.**stabilityConsensusIntervalLow = 0.25**
Classify stable term as deprecated.

CHAPTER

FOUR

TOP LEVEL PROGRAMS `sea` AND `ice`

Two top-level Python programs that make use of the *SeaIce* API are available in the root directory of the source distribution. Both share two parameters in comon:

- `--config` – File that specifies the postgres role for a local database, typically `$HOME/.seaice`. (Syntax of this file given below.) If `heroku` is given instead of a filename, then a remote database specified by the environment variable `DATABSE_URL` will be used. See `seaice.SeaIceConnector` for details.
- `--role` – Role to use for connection to a local database. The parameter value must appear in the DB config file.

4.1 `sea`

`sea` is the command line UI for *SeaIce* and provides administrative functionality. It allows you to initialize and drop the database schema, import and export individual tables, score and classify terms in the database, and seed a user's reputation. Use `--help` for a full list of options. Following are some important example usages.

Schedule term classification. This process is not inherently periodic in nature, but is dramatically simplified by scheduling it at regular intervals. Currently this occurs hourly on the Heroku deployment. In `crontab` for example:

```
0 * * * * /usr/bin/python /directory/of/seaice/source/sea --classify-terms
```

Remove a term. A useful feature to implement for *SeaIce* would be the ability to flag abusive or spam terms for deletion. In the meantime, it's posible for the administrator to delete a term manually with `--remove=ID`, where `ID` is the term's surrogate ID.

Reset the DB. When modifying the DB schema, it may be necessary to reload the contents.

```
$ ./sea --export=Userss >users.json
$ ./sea --export=Terms >terms.json
$ ./sea --export=Comments >comments.json
$ ./sea --export=Tracking >tracking.json
$ ./sea --drop-db --init-db -q
```

Now add the necessary modifications directly to the `*.json` exports. When importing, it's important to do so in the correct order since the tables use surrogate keys to reference each other.

```
$ ./sea --import=Userss <users.json
$ ./sea --import=Terms <terms.json
$ ./sea --import=Comments <comments.json
$ ./sea --import=Tracking <tracking.json
```

Seed reputation. Use `--set-reputation=N` with `--user=ID`, where `N` is the new reputation value and `ID` is the surrogate ID of the user. Say you want to seed the reputation of the notorious üwe Nordberger to 400. Find his ID by exporting the table to standard output:

```
$ ./sea --export=Users
[
      ...
  {
    "auth_id": "something secret",
    "authority": "google",
    "email": "fella@guy.de",
    "first_name": "\u00dcwe",
    "id": 1032,
    "last_name": "Nordberger",
    "reputation": 1
  }
      ...
]
$ ./sea --set-retpuation=400 --user=1032
```

Score terms manually. When a vote is cast, the new consensus score of a term is calculated immediately in constant time. However, using `--score-terms` will cause each term in the database to be scored once "the hard way". For each term, the reputations for all up voters and down voters of each term are collected and used to compute the score. This is quite inefficient, roughly *O(mn)* for *m* users and *n* votes. In addition, it causes a join on the *User* and *Tracking* tables. In spite of this, I found it useful for verifying the more complex functions `SeaIceConnector.castVote` and `SeaIceConnector.updateUserReputaiton` in development.

4.2 `ice`

This program utilizes the entire *SeaIce* API functionality to implement a Flask-based web framework. The main object, `seaice.SeaIceFlask` creates a DB connection pool (all inherit the `--config` configuration), surrogate ID pools for tables, and data structures for authenticated user sessions and notifications. The code in `ice` defines the various *GET* and *POST* requests that are served. In addition, Flask's login manager (`Flask-login`) is imported to handle authentication of sessions and anonymous users. Finally, Flask provides a simple way to run the framework for local testing. In deployment, the run code is ommitted and `ice` is run with a standalone web server. (`gunicorn ice.py` on Heroku.)

4.3 DB configuration file

Three parameters are specified for each view: **user**, **password**, and **dbname**. Views are given in squar brackets. E.g.:

```
[default]
dbname = seaice
user = postgres
password = SECRET
[contributor]
dbname = seaice
user = contributor
password = SECRET
```

This version of *SeaIce* is meant as a proof-of-concept and is missing some desirable features. Currently, reputation of users is seeded in the database and there is no way to gain reputation by contributing to the metadictionary. Other missing features include:

- Contextual IDs for terms. The ability to reference terms on *SeaIce* elswehere.
- Flag irrelevant/abusive terms and comments. This is standard on other crowd-sourced services.
- More notifications.

The prototype is deployed on Heroku. The source code for the project is distributed under the terms of the BSD license and is published on github.

SeaIce was originally developed by Christopher Patton as an internship for DataONE. CSS and JavaScript templates were contributed by Karthik Ram.

PYTHON MODULE INDEX

S