stream$framework$Documentation

Release

Thierry Schellenbach

August 10, 2015

Contents

1 What can you build? 3

2 GetStream.io 5

3 Consultancy 7

4 Using Stream Framework 9

5 Features 11

6 Background Articles 13

7 Documentation 15
- 7.1 Installation . . . 15
- 7.2 Feed setup . . . 15
- 7.3 Adding data . . . 16
- 7.4 Verbs . . . 16
- 7.5 Querying feeds . . . 17
- 7.6 Settings . . . 17
- 7.7 Metrics . . . 19
- 7.8 Testing Stream Framework . . . 19
- 7.9 Support . . . 20
- 7.10 Activity class . . . 20
- 7.11 Choosing a storage layer . . . 21
- 7.12 Background Tasks with celery . . . 22
- 7.13 Tutorial: building a notification feed . . . 23
- 7.14 Stream Framework Design . . . 26
- 7.15 Cassandra storage backend . . . 27

8 API Docs 29
- 8.1 Stream Framework API Docs . . . 29
- 8.2 Indices and tables . . . 50

Python Module Index 51

Note

This project was previously named Feedly. As requested by feedly.com we have now renamed the project to Stream Framework. You can find more details about the name change on the blog.

CHAPTER 1

What can you build?

Stream Framework allows you to build newsfeed and notification systems using Cassandra and/or Redis. Examples of what you can build are the Facebook newsfeed, your Twitter stream or your Pinterest following page. We've built Feedly for Fashiolista where it powers the flat feed, aggregated feed and the notification system. (Feeds are also commonly called: Activity Streams, activity feeds, news streams.)

To quickly make you acquainted with Stream Framework, we've created a Pinterest like example application, you can find it here

CHAPTER 2

GetStream.io

Stream Framework's authors also offer a Saas solution for building feed systems at getstream.io The hosted service is highly optimized and allows you start building your application immediatly. It saves you the hastle of maintaining Cassandra, Redis, Faye, RabbitMQ and Celery workers. Clients are available for Node, Ruby, Python, Java and PHP

CHAPTER 3

Consultancy

For Stream Framework and GetStream.io consultancy please contact thierry at getstream.io

Authors

- Thierry Schellenbach (thierry at getstream.io)
- Tommaso Barbugli (tommaso at getstream.io)
- Guyon Morée

Resources

- Documentation
- Bug Tracker
- Code
- Mailing List
- IRC (irc.freenode.net, #feedly-python)
- Travis CI

Tutorials

- Pinterest style feed example app

CHAPTER 4

Using Stream Framework

This quick example will show you how to publish a Pin to all your followers. So lets create an activity for the item you just pinned.

```
def create_activity(pin):
    from stream_framework.activity import Activity
    activity = Activity(
        pin.user_id,
        PinVerb,
        pin.id,
        pin.influencer_id,
        time=make_naive(pin.created_at, pytz.utc),
        extra_context=dict(item_id=pin.item_id)
    )
    return activity
```

Next up we want to start publishing this activity on several feeds. First of we want to insert it into your personal feed, and secondly into the feeds of all your followers. Lets start first by defining these feeds.

```
# setting up the feeds

from stream_framework.feeds.redis import RedisFeed


class PinFeed(RedisFeed):
    key_format = 'feed:normal:%(user_id)s'

class UserPinFeed(PinFeed):
    key_format = 'feed:user:%(user_id)s'
```

Writing to these feeds is very simple. For instance to write to the feed of user 13 one would do

```
feed = UserPinFeed(13)
feed.add(activity)
```

But we don't want to publish to just one users feed. We want to publish to the feeds of all users which follow you. This action is called a fanout and is abstracted away in the manager class. We need to subclass the Manager class and tell it how we can figure out which user follow us.

```
from stream_framework.feed_managers.base import Manager


class PinManager(Manager):
    feed_classes = dict(
```

```
        normal=PinFeed,
    )
    user_feed_class = UserPinFeed

    def add_pin(self, pin):
        activity = pin.create_activity()
        # add user activity adds it to the user feed, and starts the fanout
        self.add_user_activity(pin.user_id, activity)

    def get_user_follower_ids(self, user_id):
        ids = Follow.objects.filter(target=user_id).values_list('user_id', flat=True)
        return {FanoutPriority.HIGH:ids}

manager = PinManager()
```

Now that the manager class is setup broadcasting a pin becomes as easy as

```
manager.add_pin(pin)
```

Calling this method wil insert the pin into your personal feed and into all the feeds of users which follow you. It does so by spawning many small tasks via Celery. In Django (or any other framework) you can now show the users feed.

```
# django example

@login_required
def feed(request):
    '''
    Items pinned by the people you follow
    '''
    context = RequestContext(request)
    feed = manager.get_feeds(request.user.id)['normal']
    activities = list(feed[:25])
    context['activities'] = activities
    response = render_to_response('core/feed.html', context)
    return response
```

This example only briefly covered how Stream Framework works. The full explanation can be found on read the docs.

CHAPTER 5

Features

Stream Framework uses celery and Redis/Cassandra to build a system with heavy writes and extremely light reads. It features:

- Asynchronous tasks (All the heavy lifting happens in the background, your users don't wait for it)
- Reusable components (You will need to make tradeoffs based on your use cases, Stream Framework doesnt get in your way)
- Full Cassandra and Redis support
- The Cassandra storage uses the new CQL3 and Python-Driver packages, which give you access to the latest Cassandra features.
- Built for the extremely performant Cassandra 2.0

CHAPTER 6

Background Articles

A lot has been written about the best approaches to building feed based systems. Here's a collection on some of the talks:

Twitter 2013 Redis based, database fallback, very similar to Fashiolista's old approach.

Etsy feed scaling (Gearman, separate scoring and aggregation steps, rollups - aggregation part two)

Facebook history

Django project with good naming conventions

Activity stream specification

Quora post on best practises

Quora scaling a social network feed

Redis ruby example

FriendFeed approach

Thoonk setup

Yahoo Research Paper

Twitter's approach

Cassandra at Instagram

CHAPTER 7

Documentation

7.1 Installation

Installation is easy using `pip` both redis and cassandra dependencies are installed by the setup.

```
$ pip install Stream-Framework
```

or get it from source

```
$ git clone https://github.com/tschellenbach/Stream-Framework.git
$ cd Stream-Framework
$ python setup.py install
```

Depending on the backend you are going to use (*Choosing a storage layer*) you will need to have the backend server up and running.

7.2 Feed setup

A feed object contains activities. The example below shows you how to setup two feeds:

```
# implement your feed with redis as storage

from stream_framework.feeds.redis import RedisFeed

class PinFeed(RedisFeed):
    key_format = 'feed:normal:%(user_id)s'

class UserPinFeed(PinFeed):
    key_format = 'feed:user:%(user_id)s'
```

Next up we need to hook up the Feeds to your Manager class. The Manager class knows how to fanout new activities to the feeds of all your followers.

```
from stream_framework.feed_managers.base import Manager


class PinManager(Manager):
    feed_classes = dict(
        normal=PinFeed,
    )
    user_feed_class = UserPinFeed
```

```
    def add_pin(self, pin):
        activity = pin.create_activity()
        # add user activity adds it to the user feed, and starts the fanout
        self.add_user_activity(pin.user_id, activity)

    def get_user_follower_ids(self, user_id):
        ids = Follow.objects.filter(target=user_id).values_list('user_id', flat=True)
        return {FanoutPriority.HIGH:ids}

manager = PinManager()
```

7.3 Adding data

You can add an Activity object to the feed using the add or add_many instructions.

```
feed = UserPinFeed(13)
feed.add(activity)

# add many example
feed.add_many([activity])
```

What's an activity

The activity object is best described using an example. For Pinterest for instance a common activity would look like this:

Thierry added an item to his board Surf Girls.

In terms of the activity object this would translate to:

```
Activity(
        actor=13, # Thierry's user id
        verb=1, # The id associated with the Pin verb
        object=1, # The id of the newly created Pin object
        target=1, # The id of the Surf Girls board
        time=datetime.utcnow(), # The time the activity occured
)
```

The names for these fields are based on the activity stream spec.

7.4 Verbs

7.4.1 Adding new verbs

Registering a new verb is quite easy. Just subclass the Verb class and give it a unique id.

```
from stream_framework.verbs import register
from stream_framework.verbs.base import Verb


class Pin(Verb):
    id = 5
    infinitive = 'pin'
    past_tense = 'pinned'
```

```
register(Pin)
```

See also:

Make sure your verbs are registered before you read data from stream_framework, if you use django you can just define/import them in models.py to make sure they are loaded early

7.4.2 Getting verbs

You can retrieve verbs by calling get_verb_by_id.

```
from stream_framework.verbs import get_verb_by_id

pin_verb = get_verb_by_id(5)
```

7.5 Querying feeds

You can query the feed using Python slicing. In addition you can order and filter the feed on several predefined fields. Examples are shown below

Slicing:

```
feed = RedisFeed(13)
activities = feed[:10]
```

Filtering and Pagination:

```
feed.filter(activity_id__gte=1)[:10]
feed.filter(activity_id__lte=1)[:10]
feed.filter(activity_id__gt=1)[:10]
feed.filter(activity_id__lt=1)[:10]
```

Ordering feeds

New in version 0.10.0.

This is only supported using Cassandra and Redis at the moment.

```
feed.order_by('activity_id')
feed.order_by('-activity_id')
```

7.6 Settings

Note: Settings currently only support Django settings. To add support for Flask or other frameworks simply change stream_framework.settings.py

7.6.1 Redis Settings

STREAM_REDIS_CONFIG

The settings for redis, keep here the list of redis servers you want to use as feed storage

Defaults to

```
STREAM_REDIS_CONFIG = {
    'default': {
        'host': '127.0.0.1',
        'port': 6379,
        'db': 0,
        'password': None
    },
}
```

7.6.2 Cassandra Settings

STREAM_CASSANDRA_HOSTS

The list of nodes that are part of the cassandra cluster.

Note: You dont need to put every node of the cluster, cassandra-driver has built-in node discovery

---

Defaults to `['localhost']`

STREAM_DEFAULT_KEYSPACE

The cassandra keyspace where feed data is stored

Defaults to `stream_framework`

STREAM_CASSANDRA_CONSISTENCY_LEVEL

The consistency level used for both reads and writes to the cassandra cluster.

Defaults to `cassandra.ConsistencyLevel.ONE`

CASSANDRA_DRIVER_KWARGS

Extra keyword arguments sent to cassandra driver (see http://datastax.github.io/python-driver/_modules/cassandra/cluster.html#Cluster)

Defaults to `{}`

7.6.3 Metric Settings

STREAM_METRIC_CLASS

The metric class that will be used to collect feeds metrics.

Note: The default metric class is not collecting any metric and should be used as example for subclasses

---

Defaults to `stream_framework.metrics.base.Metrics`

STREAM_METRICS_OPTIONS

A dictionary with options to send to the metric class at initialisation time.

Defaults to `{}`

7.7 Metrics

Stream Framework collects metrics regarding feed operations. The default behaviour is to ignore collected metrics rather than sending them anywhere.

You can configure the metric class with the `STREAM_METRIC_CLASS` setting and send options as a python dict via `STREAM_METRICS_OPTIONS`

7.7.1 Sending metrics to Statsd

Stream Framework comes with support for StatsD support, both statsd and python-statsd libraries are supported.

If you use statsd you should use this metric class `stream_framework.metrics.statsd.StatsdMetrics` while if you are a user of python-statsd you should use `stream_framework.metrics.python_statsd.StatsdMetrics`.

The two libraries do the same job and both are suitable for production use.

By default this two classes send metrics to `localhost` which is probably not what you want.

In real life you will need something like this

```
STREAM_METRICS_OPTIONS = {
    'host': 'my.statsd.host.tld',
    'port': 8125,
    'prefix': 'stream'
}
```

7.7.2 Custom metric classes

If you need to send metrics to a not supported backend somewhere you only need to create your own subclass of stream_framework.metrics.base.Metrics and configure your application to use it.

7.8 Testing Stream Framework

Warning: We strongly suggest against running tests on a machine that is hosting redis or cassandra production data!

In order to test Stream Framework you need to install its test requirements with

```
python setup.py test
```

or if you want more control on the test run you can use py.test entry point directly (assuming you are in stream_framework dir)

```
py.test stream_framework/tests
```

The test suite connects to Redis on 127.0.0.1:6379 and to a Cassandra node on 127.0.0.1 using the native protocol.

The easiest way to run a cassandra test cluster is using the awesome ccm package

If you are not running a cassandra node on localhost you can specify a different address with the *TEST_CASSANDRA_HOST* environment variable

Every commit is built on Travis CI, you can see the current state and the build history here.

If you intend to contribute we suggest you to install pytest's coverage plugin, this way you can make sure your code changes run during tests.

7.9 Support

If you need help you can try IRC or the mailing list. Issues can be reported on Github.

- IRC (irc.freenode.net, #feedly-python)
- Mailing List
- Bug Tracker

7.10 Activity class

Activity is the core data in Stream Framework; their implementation follows the activitystream schema specification. An activity in Stream Framework is composed by an actor, a verb and an object; for example: "Geraldine posted a photo". The data stored in activities can be extended if necessary; depending on how you use Stream Framework you might want to store some extra information or not. Here is a few good rule of thumbs to follow in case you are not sure wether some information should be stored in Stream Framework:

Good choice:

1. Add a field used to perform aggregation (eg. object category)
2. You want to keep every piece of information needed to work with activities in Stream Framework (eg. avoid database lookups)

Bad choice:

1. The data stored in the activity gets updated
2. The data requires lot of storage

7.10.1 Activity storage strategies

Activities are stored on Stream Framework trying to maximise the benefits of the storage backend used.

When using the redis backend Stream Framework will keep data denormalized; activities are stored in a special storage (activity storage) and user feeds only keeps a reference (activity_id / serialization_id). This allow Stream Framework to keep the (expensive) memory usage as low as possible.

When using Cassandra as storage Stream Framework will denormalize activities; there is not an activity storage but instead every user feed will keep the complete activity. Doing so allow Stream Framework to minimise the amount of Cassandra nodes to query when retrieving data or writing to feeds.

In both storages activities are always stored in feeds sorted by their creation time (aka Activity.serialization_id).

7.10.2 Extend the activity class

New in version 0.10.0.

You can subclass the activity model to add your own methods. After you've created your own activity model you need to hook it up to the feed. An example follows below

```
from stream_framework.activity import Activity

# subclass the activity object
class CustomActivity(Activity):
        def mymethod():
                pass

# hookup the custom activity object to the Redis feed
class CustomFeed(RedisFeed):
        activity_class = CustomActivity
```

For aggregated feeds you can customize both the activity and the aggregated activity object. You can give this a try like this

```
from stream_framework.activity import AggregatedActivity

# define the custom aggregated activity
class CustomAggregated(AggregatedActivity):
    pass

# hook the custom classes up to the feed
class RedisCustomAggregatedFeed(RedisAggregatedFeed):
    activity_class = CustomActivity
    aggregated_activity_class = CustomAggregated
```

7.10.3 Activity serialization

7.10.4 Activity order and uniqueness

7.10.5 Aggregated activities

7.11 Choosing a storage layer

Currently Stream Framework supports both Cassandra and Redis as storage backends.

Summary

Redis is super easy to get started with and works fine for smaller use cases. If you're just getting started use Redis. When your data requirements become larger though it becomes really expensive to store all the data in Redis. For larger use cases we therefor recommend Cassandra.

7.11.1 Redis (2.7 or newer)

PROS:

- Easy to install
- Super reliable
- Easy to maintain
- Very fast

CONS:

- Expensive memory only storage

- Manual sharding

Redis stores its complete dataset in memory. This makes sure that all operations are always fast. It does however mean that you might need a lot of storage.

A common approach is therefor to use Redis storage for some of your feeds and fall back to your database for less frequently requested data.

Twitter currently uses this approach and Fashiolista has used a system like this in the first half of 2013.

The great benefit of using Redis comes in easy of install, reliability and maintainability. Basically it just works and there's little you need to learn to maintain it.

Redis doesn't support any form of cross machine distribution. So if you add a new node to your cluster you need to manual move or recreate the data.

In conclusion I believe Redis is your best bet if you can fallback to the database when needed.

7.11.2 Cassandra (2.0 or newer)

PROS:

- Stores to disk
- Automatic sharding across nodes
- Awesome monitoring tools (opscenter)

CONS:

- Not as easy to setup
- Hard to maintain

Cassandra stores data to both disk and memory. Instagram has recently switched from Redis to Cassandra. Storing data to disk can potentially be a big cost saving.

In addition adding new machines to your Cassandra cluster is a breeze. Cassandra will automatically distribute the data to new machines.

If you are using amazon EC2 we suggest you to try Datastax's easy AMI to get started on AWS.

7.12 Background Tasks with celery

Stream Framework uses celery to do the heavy fanout write operations in the background.

We really suggest you to have a look at celery documentation if you are not familiar with the project.

Fanout

When an activity is added Stream Framework will perform a fanout to all subscribed feeds. The base Stream Framework manager spawns one celery fanout task every 100 feeds. Change the value of *fanout_chunk_size* of your manager if you think this number is too low/high for you.

Few things to keep in mind when doing so:

1. really high values leads to a mix of heavy tasks and light tasks (not good!)
2. publishing and consuming tasks introduce some overhead, dont spawn too many tasks
3. Stream Framework writes data in batches, thats a really good optimization you want to keep
4. huge tasks have more chances to timeout

Note: When developing you can run fanouts without celery by setting *CELERY_ALWAYS_EAGER = True*

7.12.1 Prioritise fanouts

Stream Framework partition fanout tasks in two priority groups. Fanouts with different priorities do exactly the same operations (adding/removing activities from/to a feed) the substantial difference is that they get published to different queues for processing. Going back to our pinterest example app, you can use priorities to associate more resources to fanouts that target active users and send the ones for inactive users to a different cluster of workers. This also make it easier and cheaper to keep active users' feeds updated during activity spikes because you dont need to scale up capacity less often.

Stream Framework manager is the best place to implement your high/low priority fanouts, in fact the *get_follower_ids* method is required to return the feed ids grouped by priority.

eg:

```
class MyStreamManager(Manager):

    def get_user_follower_ids(self, user_id):
        follower_ids = {
                FanoutPriority.HIGH: get_follower_ids(user_id, active=True),
                FanoutPriority.LOW: get_follower_ids(user_id, active=False)
        }
        return follower_ids
```

7.12.2 Celery and Django

If this is the time you use Celery and Django together I suggest you should follow this document's instructions.

It will guide you through the required steps to get Celery background processing up and running.

7.12.3 Using other job queue libraries

As of today background processing is tied to celery.

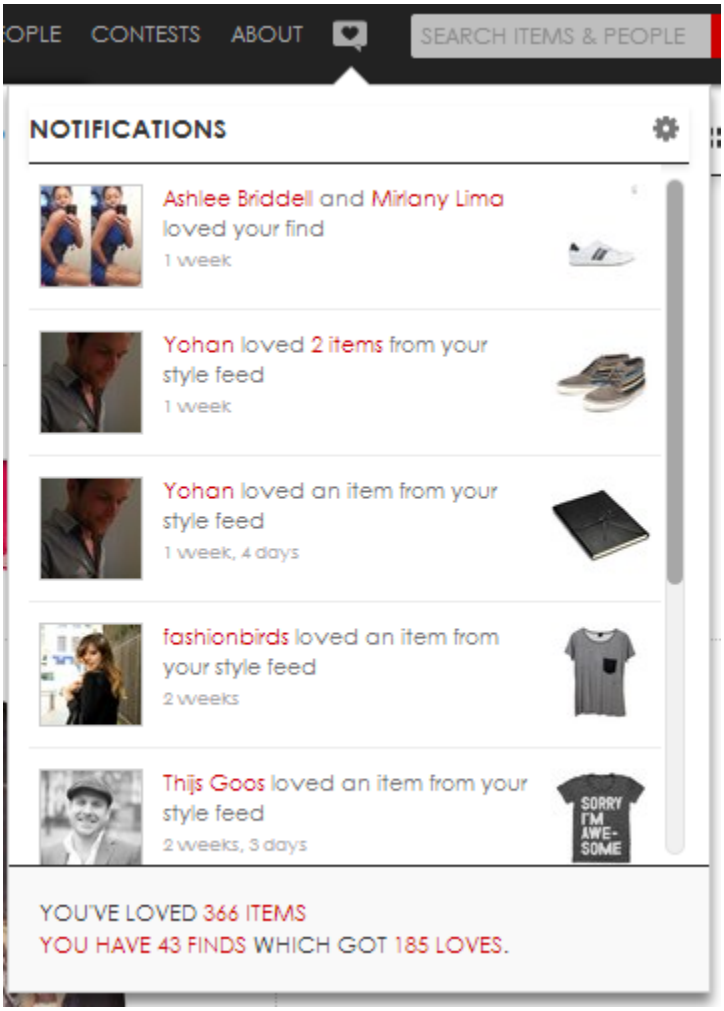
While we are not planning to support different queue jobs libraries in the near future using something different than celery should be quite easy and can be mostly done subclassing the feeds manager.

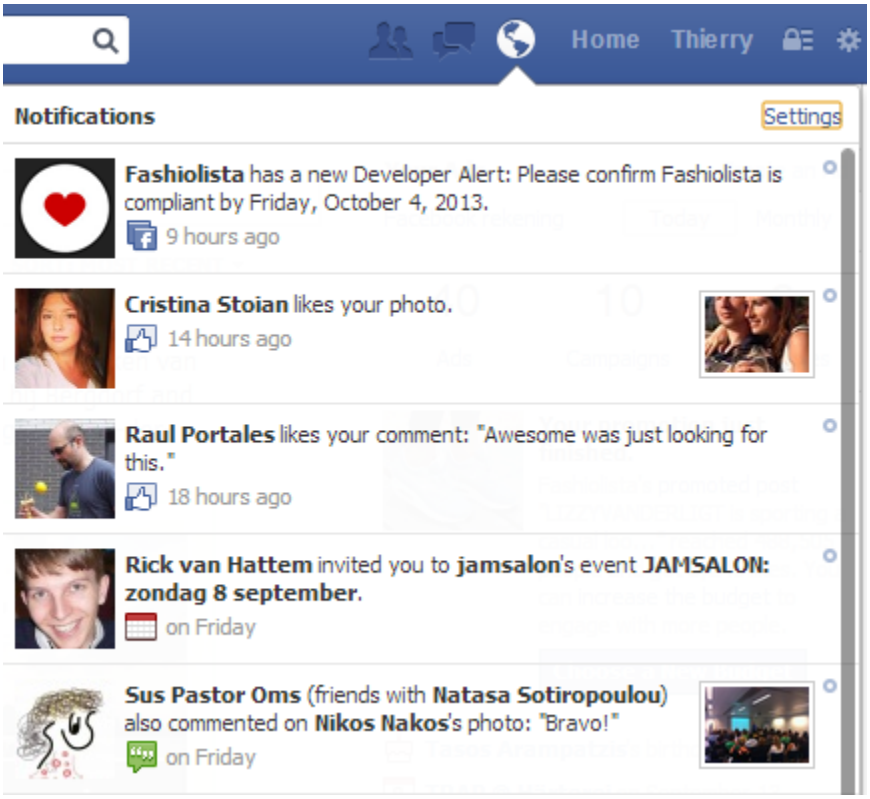
7.13 Tutorial: building a notification feed

Note: We are still improving this tutorial. In its current state it might be a bit hard to follow.

7.13.1 What is a notification system?

Building a scalable notification system is almost entirely identical to building an activity feed. From the user's perspective the functionality is pretty different. A notification system commonly shows activity related to your account. Whereas an activity stream shows activity by the people you follow. Examples of Fashiolista's notification system and Facebook's system are shown below. Fashiolista's system is running on Stream Framework.


OPLE
CONTESTS
ABOUT
SEARCH ITEMS & PEOPLE
NOTIFICATIONS
Ashlee Briddell and Mirlany Lima loved your find
1 week
Yohan loved 2 items from your style feed
1 week
Yohan loved an item from your style feed
1 week, 4 days
fashionbirds loved an item from your style feed
2 weeks
Thijs Goos loved an item from your style feed
2 weeks, 3 days
SORRY I'M AWE-SOME
YOU'VE LOVED 366 ITEMS
YOU HAVE 43 FINDS WHICH GOT 185 LOVES.



It looks very different from an activity stream, but the technical implementation is almost identical. Only the Feed manager class is different since the notification system has no fanouts.

Note: Remember, Fanout is the process which pushes a little bit of data to all of your followers in many small and asynchronous tasks.

7.13.2 Tutorial

For this tutorial we'll show you how to customize and setup your own notification system.

Step 1 - Subclass NotificationFeed

As a first step we'll subclass NotificationFeed and customize the storage location and the aggregator.

```
from stream_framework.feeds.aggregated_feed.notification_feed import RedisNotificationFeed

class MyNotificationFeed(RedisNotificationFeed):
    # : they key format determines where the data gets stored
    key_format = 'feed:notification:%(user_id)s'

    # : the aggregator controls how the activities get aggregated
    aggregator_class = MyAggregator
```

Step 2 - Subclass the aggregator

Secondly we want to customize how activities get grouped together. Most notification systems need to aggregate activities. In this case we'll aggregate on verb and date. So the aggregations will show something like (thierry, peter and two other people liked your photo).

```
class MyAggregator(BaseAggregator):
    '''
    Aggregates based on the same verb and same time period
    '''
    def get_group(self, activity):
        '''
        Returns a group based on the day and verb
        '''
        verb = activity.verb.id
        date = activity.time.date()
        group = '%s-%s' % (verb, date)
        return group
```

Step 3 - Test adding data

The aggregated feed uses the same API as the flat feed. You can simply add items by calling feed.add or feed.add_many. An example for inserting data is shown below:

```
feed = MyNotificationFeed(user_id)
activity = Activity(
    user_id, LoveVerb, object_id, influencer_id, time=created_at,
    extra_context=dict(entity_id=self.entity_id)
)
feed.add(activity)
print feed[:5]
```

Step 4 - Implement manager functionality

To keep our code clean we'll implement a very simple manager class to abstract away the above code.

```
class MyNotification(object):
    '''
    Abstract the access to the notification feed
    '''
    def add_love(self, love):
        feed = MyNotificationFeed(user_id)
        activity = Activity(
            love.user_id, LoveVerb, love.id, love.influencer_id,
            time=love.created_at, extra_context=dict(entity_id=self.entity_id)
        )
        feed.add(activity)
```

7.14 Stream Framework Design

The first approach

A first feed solution usually looks something like this:

```
SELECT * FROM tweets
JOIN follow ON (follow.target_id = tweet.user_id)
WHERE follow.user_id = 13
```

This works in the beginning, and with a well tuned database will keep on working nicely for quite some time. However at some point the load becomes too much and this approach falls apart. Unfortunately it's very hard to split up the tweets in a meaningfull way. You could split it up by date or user, but every query will still hit many of your shards. Eventually this system collapses, read more about this in Facebook's presentation.

Push or Push/Pull

In general there are two similar solutions to this problem.

In the push approach you publish your activity (ie a tweet on twitter) to all of your followers. So basically you create a small list per user to which you insert the activities created by the people they follow. This involves a huge number of writes, but reads are really fast they can easily be sharded.

For the push/pull approach you implement the push based systems for a subset of your users. At Fashiolista for instance we used to have a push based approach for active users. For inactive users we only kept a small feed and eventually used a fallback to the database when we ran out of results.

Stream Framework

Stream Framework allows you to easily use Cassndra/Redis and Celery (an awesome task broker) to build infinitely scalable feeds. The high level functionality is located in 4 classes.

- Activities
- Feeds
- Feed managers
- Aggregators

Activities are the blocks of content which are stored in a feed. It follows the nomenclatura from the [activity stream spec] [astream] [astream]: http://activitystrea.ms/specs/atom/1.0/#activity.summary Every activity therefor stores at least:

- Time (the time of the activity)
- Verb (the action, ie loved, liked, followed)
- Actor (the user id doing the action)
- Object (the object the action is related to)
- Extra context (Used for whatever else you need to store at the activity level)

Optionally you can also add a target (which is best explained in the activity docs)

Feeds are sorted containers of activities. You can easily add and remove activities from them.

Stream Framework classes (feed managers) handle the logic used in addressing the feed objects. They handle the complex bits of fanning out to all your followers when you create a new object (such as a tweet).

In addition there are several utility classes which you will encounter

- Serializers (classes handling serialization of Activity objects)
- Aggregators (utility classes for creating smart/computed feeds based on algorithms)
- Timeline Storage (cassandra or redis specific storage functions for sorted storage)
- Activity Storage (cassandra or redis specific storage for hash/dict based storage)

7.15 Cassandra storage backend

This document is specific to the Cassandra backend.

7.15.1 Create keyspace and columnfamilies

Keyspace and columnfamilies for your feeds can be created via cqlengine's sync_table.

```
from myapp.feeds import MyCassandraFeed
from cqlengine.management import sync_table

timeline = MyCassandraFeed.get_timeline_storage()
sync_table(timeline.model)
```

sync_table can also create missing columns but it will never delete removed columns.

7.15.2 Use a custom activity model

Since the Cassandra backend is using CQL3 column families, activities have a predefined schema. Cqlengine is used to read/write data from and to Cassandra.

```
from stream_framework.storage.cassandra import models


class MyCustomActivity(models.Activity)
    actor = columns.Bytes(required=False)


class MySuperAwesomeFeed(CassandraFeed):
    timeline_model = MyCustomActivity
```

Remember to resync your column family when you add new columns (see above).

CHAPTER 8

API Docs

8.1 Stream Framework API Docs

8.1.1 stream_framework Package

8.1.2 activity Module

class stream_framework.activity.**Activity**(*actor*, *verb*, *object*, *target=None*, *time=None*, *extra_context=None*)

Bases: *stream_framework.activity.BaseActivity*

Wrapper class for storing activities Note

actor_id target_id and object_id are always present

actor, target and object are lazy by default

get_dehydrated()

returns the dehydrated version of the current activity

serialization_id

serialization_id is used to keep items locally sorted and unique (eg. used redis sorted sets' score or cassandra column names)

serialization_id is also used to select random activities from the feed (eg. remove activities from feeds must be fast operation) for this reason the serialization_id should be unique and not change over time

eg: activity.serialization_id = 13732667550000000000042008 1373266755000 activity creation time as epoch with millisecond resolution 0000000000042 activity left padded object_id (10 digits) 008 left padded activity verb id (3 digits)

Returns int –the serialization id

class stream_framework.activity.**AggregatedActivity**(*group*, *activities=None*, *created_at=None*, *updated_at=None*)

Bases: *stream_framework.activity.BaseActivity*

Object to store aggregated activities

activity_count

Returns the number of activities

activity_ids

Returns a list of activity ids

actor_count
Returns a count of the number of actors When dealing with large lists only approximate the number of actors

actor_ids

append(*activity*)

contains(*activity*)
Checks if activity is present in this aggregated

get_dehydrated()
returns the dehydrated version of the current activity

get_hydrated(*activities*)
expects activities to be a dict like this {'activity_id': Activity}

is_read()
Returns if the activity should be considered as seen at this moment

is_seen()
Returns if the activity should be considered as seen at this moment

last_activities

last_activity

max_aggregated_activities_length = 15

object_ids

other_actor_count

remove(*activity*)

remove_many(*activities*)

serialization_id
serialization_id is used to keep items locally sorted and unique (eg. used redis sorted sets' score or cassandra column names)

serialization_id is also used to select random activities from the feed (eg. remove activities from feeds must be fast operation) for this reason the serialization_id should be unique and not change over time

eg: activity.serialization_id = 13732667550000000000042008 1373266755000 activity creation time as epoch with millisecond resolution 0000000000042 activity left padded object_id (10 digits) 008 left padded activity verb id (3 digits)

Returns int –the serialization id

update_read_at()
A hook method that updates the read_at to current date

update_seen_at()
A hook method that updates the seen_at to current date

verb

verbs

class stream_framework.activity.**BaseActivity**
Bases: object

Common parent class for Activity and Aggregated Activity Check for this if you want to see if something is an activity

class stream_framework.activity.**DehydratedActivity**(*serialization_id*)
Bases: *stream_framework.activity.BaseActivity*

The dehydrated verions of an *Activity*. the only data stored is serialization_id of the original

Serializers can store this instead of the full activity Feed classes

get_hydrated(*activities*)
returns the full hydrated Activity from activities

Parameters **a dict {'activity_id'** (*activities*) – Activity}

class stream_framework.activity.**NotificationActivity**(**args*, ***kwargs*)
Bases: *stream_framework.activity.AggregatedActivity*

8.1.3 default_settings Module

8.1.4 exceptions Module

exception stream_framework.exceptions.**ActivityNotFound**
Bases: exceptions.Exception

Raised when the activity is not present in the aggregated Activity

exception stream_framework.exceptions.**DuplicateActivityException**
Bases: exceptions.Exception

Raised when someone sticks a duplicate activity in the aggregated activity

exception stream_framework.exceptions.**SerializationException**
Bases: exceptions.Exception

Raised when encountering invalid data for serialization

8.1.5 settings Module

stream_framework.settings.**import_global_module**(*module*, *current_locals*, *current_globals*, *exceptions=None*)
Import the requested module into the global scope Warning! This will import your module into the global scope

Example: from django.conf import settings import_global_module(settings, locals(), globals())

Parameters

- **module** – the module which to import into global scope
- **current_locals** – the local globals
- **current_globals** – the current globals
- **exceptions** – the exceptions which to ignore while importing

8.1.6 tasks Module

8.1.7 utils Module

class stream_framework.utils.**LRUCache**(*capacity*)

get(*key*)

set(*key*, *value*)

stream_framework.utils.**chunks**(*iterable*, *n=10000*)

stream_framework.utils.**datetime_to_epoch**(*dt*)
Convert datetime object to epoch with millisecond accuracy

stream_framework.utils.**epoch_to_datetime**(*time_*)

stream_framework.utils.**get_class_from_string**(*path*, *default=None*)
Return the class specified by the string.

stream_framework.utils.**get_metrics_instance**()
Returns an instance of the metric class as defined in stream_framework settings.

stream_framework.utils.**make_list_unique**(*sequence*, *marker_function=None*)
Makes items in a list unique Performance based on this blog post: http://www.peterbe.com/plog/uniqifiers-benchmark

class stream_framework.utils.**memoized**(*func*)
Bases: object

Decorator. Caches a function's return value each time it is called. If called later with the same arguments, the cached value is returned (not reevaluated).

stream_framework.utils.**warn_on_duplicate**(*f*)

stream_framework.utils.**warn_on_error**(*f*, *exceptions*)

8.1.8 Subpackages

aggregators Package

base Module

class stream_framework.aggregators.base.**BaseAggregator**(*aggregated_activity_class=None*, *activity_class=None*)
Bases: object

Aggregators implement the combining of multiple activities into aggregated activities.

The two most important methods are aggregate and merge

Aggregate takes a list of activities and turns it into a list of aggregated activities

Merge takes two lists of aggregated activities and returns a list of new and changed aggregated activities

activity_class
alias of Activity

aggregate(*activities*)

Parameters **activties** – A list of activities

Returns list A list of aggregated activities

Runs the group activities (using get group) Ranks them using the giving ranking function And returns the sorted activities

Example

```
        aggregator = ModulusAggregator()
        activities = [Activity(1), Activity(2)]
        aggregated_activities = aggregator.aggregate(activities)
```

aggregated_activity_class
alias of AggregatedActivity

get_group(*activity*)
Returns a group to stick this activity in

group_activities(*activities*)
Groups the activities based on their group Found by running get_group(actvity on them)

merge(*aggregated*, *activities*)

Parameters

- **aggregated** – A list of aggregated activities
- **activities** – A list of the new activities

Returns tuple Returns new, changed

Merges two lists of aggregated activities and returns the new aggregated activities and a from, to mapping of the changed aggregated activities

Example

```
        aggregator = ModulusAggregator()
        activities = [Activity(1), Activity(2)]
        aggregated_activities = aggregator.aggregate(activities)
        activities = [Activity(3), Activity(4)]
        new, changed = aggregator.merge(aggregated_activities, activities)
        for activity in new:
            print activity

        for from, to in changed:
            print 'changed from %s to %s' % (from, to)
```

rank(*aggregated_activities*)
The ranking logic, for sorting aggregated activities

class stream_framework.aggregators.base.**NotificationAggregator**(*aggregated_activity_class=None*, *activity_class=None*)
Bases: *stream_framework.aggregators.base.RecentRankMixin*, *stream_framework.aggregators.base.BaseAggregator*

Aggregates based on the same verb, object and day

get_group(*activity*)
Returns a group based on the verb, object and day

class stream_framework.aggregators.base.**RecentRankMixin**
Bases: object

Most recently updated aggregated activities are ranked first.

rank(*aggregated_activities*)
The ranking logic, for sorting aggregated activities

class stream_framework.aggregators.base.**RecentVerbAggregator**(*aggregated_activity_class=None*, *activity_class=None*)

Bases: *stream_framework.aggregators.base.RecentRankMixin*, *stream_framework.aggregators.base.BaseAggregator*

Aggregates based on the same verb and same time period

get_group(*activity*)

Returns a group based on the day and verb

feed_managers Package

base Module

feeds Package

base Module

class stream_framework.feeds.base.**BaseFeed**(*user_id*)

Bases: object

The feed class allows you to add and remove activities from a feed. Please find below a quick usage example.

Usage Example:

```
feed = BaseFeed(user_id)
# start by adding some existing activities to a feed
feed.add_many([activities])
# querying results
results = feed[:10]
# removing activities
feed.remove_many([activities])
# counting the number of items in the feed
count = feed.count()
feed.delete()
```

The feed is easy to subclass. Commonly you'll want to change the max_length and the key_format.

Subclassing:

```
class MyFeed(BaseFeed):
    key_format = 'user_feed:%(user_id)s'
    max_length = 1000
```

Filtering and Pagination:

```
feed.filter(activity_id__gte=1)[:10]
feed.filter(activity_id__lte=1)[:10]
feed.filter(activity_id__gt=1)[:10]
feed.filter(activity_id__lt=1)[:10]
```

Activity storage and Timeline storage

To keep reduce timelines memory utilization the BaseFeed supports normalization of activity data.

The full activity data is stored only in the activity_storage while the timeline only keeps a activity references (refered as activity_id in the code)

For this reason when an activity is created it must be stored in the activity_storage before other timelines can refer to it

eg.

```
feed = BaseFeed(user_id)
feed.insert_activity(activity)
follower_feed = BaseFeed(follower_user_id)
feed.add(activity)
```

It is also possible to store the full data in the timeline storage

The strategy used by the BaseFeed depends on the serializer utilized by the timeline_storage

When activities are stored as dehydrated (just references) the BaseFeed will query the activity_storage to return full activities

eg.

```
feed = BaseFeed(user_id)
feed[:10]
```

gets the first 10 activities from the timeline_storage, if the results are not complete activities then the BaseFeed will hydrate them via the activity_storage

activity_class
alias of `Activity`

activity_serializer
alias of `BaseSerializer`

activity_storage_class
alias of `BaseActivityStorage`

add(*activity*, **args*, ***kwargs*)

add_many(*activities*, *batch_interface=None*, *trim=True*, **args*, ***kwargs*)
Add many activities

Parameters

- **activities** – a list of activities
- **batch_interface** – the batch interface

count()
Count the number of items in the feed

delete()
Delete the entire feed

filter(***kwargs*)
Filter based on the kwargs given, uses django orm like syntax

Example :: # filter between 100 and 200 feed = feed.filter(activity_id__gte=100) feed = feed.filter(activity_id__lte=200) # the same statement but in one step feed = feed.filter(activity_id__gte=100, activity_id__lte=200)

filtering_supported = False

classmethod flush()

get_activity_slice(*start=None*, *stop=None*, *rehydrate=True*)
Gets activity_ids from timeline_storage and then loads the actual data querying the activity_storage

classmethod get_activity_storage()
Returns an instance of the activity storage

classmethod **get_timeline_batch_interface**()

classmethod **get_timeline_storage**()
Returns an instance of the timeline storage

classmethod **get_timeline_storage_options**()
Returns the options for the timeline storage

hydrate_activities(*activities*)
hydrates the activities using the activity_storage

index_of(*activity_id*)
Returns the index of the activity id

Parameters **activity_id** – the activity id

classmethod **insert_activities**(*activities*, ***kwargs*)
Inserts an activity to the activity storage

Parameters **activity** – the activity class

classmethod **insert_activity**(*activity*, ***kwargs*)
Inserts an activity to the activity storage

Parameters **activity** – the activity class

key_format = 'feed_%(user_id)s'

max_length = 100

needs_hydration(*activities*)
checks if the activities are dehydrated

on_update_feed(*new*, *deleted*)
A hook called when activities area created or removed from the feed

order_by(**ordering_args*)
Change default ordering

ordering_supported = False

remove(*activity_id*, **args*, ***kwargs*)

classmethod **remove_activity**(*activity*, ***kwargs*)
Removes an activity from the activity storage

Parameters **activity** – the activity class or an activity id

remove_many(*activity_ids*, *batch_interface=None*, *trim=True*, **args*, ***kwargs*)
Remove many activities

Parameters **activity_ids** – a list of activities or activity ids

timeline_serializer
alias of `SimpleTimelineSerializer`

timeline_storage_class
alias of `BaseTimelineStorage`

trim(*length=None*)
Trims the feed to the length specified

Parameters **length** – the length to which to trim the feed, defaults to self.max_length

trim_chance = 0.01

class stream_framework.feeds.base.**UserBaseFeed**(*user_id*)
Bases: *stream_framework.feeds.base.BaseFeed*

Implementation of the base feed with a different Key format and a really large max_length

key_format = 'user_feed:%(user_id)s'

max_length = 1000000

cassandra Module

memory Module

class stream_framework.feeds.memory.**Feed**(*user_id*)
Bases: *stream_framework.feeds.base.BaseFeed*

activity_storage_class
alias of InMemoryActivityStorage

timeline_storage_class
alias of InMemoryTimelineStorage

redis Module

class stream_framework.feeds.redis.**RedisFeed**(*user_id*)
Bases: *stream_framework.feeds.base.BaseFeed*

activity_serializer
alias of ActivitySerializer

activity_storage_class
alias of RedisActivityStorage

filtering_supported = True

classmethod get_timeline_storage_options()
Returns the options for the timeline storage

ordering_supported = True

redis_server = 'default'

timeline_storage_class
alias of RedisTimelineStorage

Subpackages

aggregated_feed Package

aggregated_feed Package

base Module

class stream_framework.feeds.aggregated_feed.base.**AggregatedFeed**(*user_id*)

Bases: *stream_framework.feeds.base.BaseFeed*

Aggregated feeds are an extension of the basic feed. They turn activities into aggregated activities by using an aggregator class.

See *BaseAggregator*

You can use aggregated feeds to built smart feeds, such as Facebook's newsfeed. Alternatively you can also use smart feeds for building complex notification systems.

Have a look at fashiolista.com for the possibilities.

Note: Aggregated feeds do more work in the fanout phase. Remember that for every user activity the number of fanouts is equal to their number of followers. So with a 1000 user activities, with an average of 500 followers per user, you already end up running 500.000 fanout operations

Since the fanout operation happens so often, you should make sure not to do any queries in the fanout phase or any other resource intensive operations.

Aggregated feeds differ from feeds in a few ways:

- Aggregator classes aggregate activities into aggregated activities
- We need to update aggregated activities instead of only appending
- Serialization is different

add_many(*activities*, *trim=True*, *current_activities=None*, **args*, ***kwargs*)

Adds many activities to the feed

Unfortunately we can't support the batch interface. The writes depend on the reads.

Also subsequent writes will depend on these writes. So no batching is possible at all.

Parameters **activities** – the list of activities

add_many_aggregated(*aggregated*, **args*, ***kwargs*)

Adds the list of aggregated activities

Parameters **aggregated** – the list of aggregated activities to add

aggregated_activity_class

alias of AggregatedActivity

aggregator_class

alias of RecentVerbAggregator

contains(*activity*)

Checks if the activity is present in any of the aggregated activities

Parameters **activity** – the activity to search for

get_aggregator()

Returns the class used for aggregation

classmethod **get_timeline_storage_options**()

Returns the options for the timeline storage

merge_max_length = 20

remove_many(*activities*, *batch_interface=None*, *trim=True*, **args*, ***kwargs*)

Removes many activities from the feed

Parameters **activities** – the list of activities to remove

remove_many_aggregated(*aggregated*, **args*, ***kwargs*)
Removes the list of aggregated activities

Parameters **aggregated** – the list of aggregated activities to remove

timeline_serializer
alias of AggregatedActivitySerializer

cassandra Module

redis Module

class stream_framework.feeds.aggregated_feed.redis.**RedisAggregatedFeed**(*user_id*)
Bases: *stream_framework.feeds.aggregated_feed.base.AggregatedFeed*

activity_serializer
alias of ActivitySerializer

activity_storage_class
alias of RedisActivityStorage

timeline_serializer
alias of AggregatedActivitySerializer

timeline_storage_class
alias of RedisTimelineStorage

notification_feed Module

class stream_framework.feeds.aggregated_feed.notification_feed.**NotificationFeed**(*user_id*, ***kwargs*)
Bases: *stream_framework.feeds.aggregated_feed.base.AggregatedFeed*

Similar to an aggregated feed, but: - doesnt use the activity storage (serializes everything into the timeline storage) - features denormalized counts - pubsub signals which you can subscribe to For now this is entirely tied to Redis

activity_serializer = None

activity_storage_class = None

add_many(*activities*, ***kwargs*)
Similar to the AggregatedActivity.add_many The only difference is that it denormalizes a count of unseen activities

count_format = 'notification_feed:1:user:%(user_id)s:count'
the format we use to denormalize the count

count_unseen(*aggregated_activities=None*)
Counts the number of aggregated activities which are unseen

Parameters **aggregated_activities** – allows you to specify the aggregated activities for improved performance

denormalize_count()
Denormalize the number of unseen aggregated activities to the key defined in self.count_key

get_denormalized_count()
Returns the denormalized count stored in self.count_key

key_format = 'notification_feed:1:user:%(user_id)s'

lock_format = 'notification_feed:1:user:%s:lock'
the key used for locking

mark_all(*seen=True*, *read=None*)
Mark all the entries as seen or read

Parameters

- **seen** – set seen_at
- **read** – set read_at

max_length = 99
notification feeds only need a small max length

publish_count(*count*)
Published the count via pubsub

Parameters **count** – the count to publish

pubsub_main_channel = 'juggernaut'
the main channel to publish

set_denormalized_count(*count*)
Updates the denormalized count to count

Parameters **count** – the count to update to

timeline_serializer
alias of NotificationSerializer

class stream_framework.feeds.aggregated_feed.notification_feed.**RedisNotificationFeed**(*user_id*, ***kwargs*)

Bases: *stream_framework.feeds.aggregated_feed.notification_feed.NotificationFeed*

timeline_storage_class
alias of RedisTimelineStorage

storage Package

base Module

class stream_framework.storage.base.**BaseActivityStorage**(*serializer_class=None*, *activity_class=None*, ***options*)

Bases: *stream_framework.storage.base.BaseStorage*

The Activity storage globally stores a key value mapping. This is used to store the mapping between an activity_id and the actual activity object.

Example:

```
storage = BaseActivityStorage()
storage.add_many(activities)
storage.get_many(activity_ids)
```

The storage specific functions are located in

•add_to_storage

•get_from_storage

•remove_from_storage

add(*activity*, **args*, ***kwargs*)

add_many(*activities*, **args*, ***kwargs*)
Adds many activities and serializes them before forwarding this to add_to_storage

Parameters **activities** – the list of activities

add_to_storage(*serialized_activities*, **args*, ***kwargs*)
Adds the serialized activities to the storage layer

Parameters **serialized_activities** – a dictionary with {id: serialized_activity}

get(*activity_id*, **args*, ***kwargs*)

get_from_storage(*activity_ids*, **args*, ***kwargs*)
Retrieves the given activities from the storage layer

Parameters **activity_ids** – the list of activity ids

Returns **dict** a dictionary mapping activity ids to activities

get_many(*activity_ids*, **args*, ***kwargs*)
Gets many activities and deserializes them

Parameters **activity_ids** – the list of activity ids

remove(*activity*, **args*, ***kwargs*)

remove_from_storage(*activity_ids*, **args*, ***kwargs*)
Removes the specified activities

Parameters **activity_ids** – the list of activity ids

remove_many(*activities*, **args*, ***kwargs*)
Figures out the ids of the given activities and forwards The removal to the remove_from_storage function

Parameters **activities** – the list of activities

class stream_framework.storage.base.**BaseStorage**(*serializer_class=None*, *activity_class=None*, ***options*)

Bases: object

The feed uses two storage classes, the - Activity Storage and the - Timeline Storage

The process works as follows:

```
feed = BaseFeed()
# the activity storage is used to store the activity and mapped to an id
feed.insert_activity(activity)
# now the id is inserted into the timeline storage
feed.add(activity)
```

Currently there are two activity storage classes ready for production:

•Cassandra

•Redis

The storage classes always receive a full activity object. The serializer class subsequently determines how to transform the activity into something the database can store.

activities_to_ids(*activities_or_ids*)
Utility function for lower levels to chose either serialize

activity_class
alias of Activity

activity_to_id(*activity*)

aggregated_activity_class
alias of `AggregatedActivity`

default_serializer_class
The default serializer class to use

alias of `DummySerializer`

deserialize_activities(*serialized_activities*)
Serializes the list of activities

Parameters

- **serialized_activities** – the list of activities
- **serialized_activities** – a dictionary with activity ids and activities

flush()
Flushes the entire storage

metrics = <stream_framework.metrics.base.Metrics object>

serialize_activities(*activities*)
Serializes the list of activities

Parameters **activities** – the list of activities

serialize_activity(*activity*)
Serialize the activity and returns the serialized activity

Returns str the serialized activity

serializer
Returns an instance of the serializer class

The serializer needs to know about the activity and aggregated activity classes we're using

class `stream_framework.storage.base.`**BaseTimelineStorage**(*serializer_class=None*, *activity_class=None*, ***options*)
Bases: *stream_framework.storage.base.BaseStorage*

The Timeline storage class handles the feed/timeline sorted part of storing a feed.

Example:

```
storage = BaseTimelineStorage()
storage.add_many(key, activities)
# get a sorted slice of the feed
storage.get_slice(key, start, stop)
storage.remove_many(key, activities)
```

The storage specific functions are located in

add(*key*, *activity*, **args*, ***kwargs*)

add_many(*key*, *activities*, **args*, ***kwargs*)
Adds the activities to the feed on the given key (The serialization is done by the serializer class)

Parameters

- **key** – the key at which the feed is stored
- **activities** – the activities which to store

count(*key*, **args*, ***kwargs*)

default_serializer_class
alias of SimpleTimelineSerializer

delete(*key*, **args*, ***kwargs*)

get_batch_interface()
Returns a context manager which ensure all subsequent operations Happen via a batch interface

An example is redis.map

get_index_of(*key*, *activity_id*)

get_slice(*key*, *start*, *stop*, *filter_kwargs=None*, *ordering_args=None*)
Returns a sorted slice from the storage

Parameters **key** – the key at which the feed is stored

get_slice_from_storage(*key*, *start*, *stop*, *filter_kwargs=None*, *ordering_args=None*)

Parameters

- **key** – the key at which the feed is stored
- **start** – start
- **stop** – stop

Returns list Returns a list with tuples of key,value pairs

index_of(*key*, *activity_or_id*)
Returns activity's index within a feed or raises ValueError if not present

Parameters

- **key** – the key at which the feed is stored
- **activity_id** – the activity's id to search

remove(*key*, *activity*, **args*, ***kwargs*)

remove_from_storage(*key*, *serialized_activities*)

remove_many(*key*, *activities*, **args*, ***kwargs*)
Removes the activities from the feed on the given key (The serialization is done by the serializer class)

Parameters

- **key** – the key at which the feed is stored
- **activities** – the activities which to remove

trim(*key*, *length*)
Trims the feed to the given length

Parameters

- **key** – the key location
- **length** – the length to which to trim

memory Module

class stream_framework.storage.memory.**InMemoryActivityStorage**(*serializer_class=None*, *activity_class=None*, ***options*)

Bases: *stream_framework.storage.base.BaseActivityStorage*

add_to_storage(*activities*, **args*, ***kwargs*)

flush()

get_from_storage(*activity_ids*, **args*, ***kwargs*)

remove_from_storage(*activity_ids*, **args*, ***kwargs*)

class stream_framework.storage.memory.**InMemoryTimelineStorage**(*serializer_class=None*, *activity_class=None*, ***options*)

Bases: *stream_framework.storage.base.BaseTimelineStorage*

add_to_storage(*key*, *activities*, **args*, ***kwargs*)

contains(*key*, *activity_id*)

count(*key*, **args*, ***kwargs*)

delete(*key*, **args*, ***kwargs*)

classmethod get_batch_interface()

get_index_of(*key*, *activity_id*)

get_slice_from_storage(*key*, *start*, *stop*, *filter_kwargs=None*, *ordering_args=None*)

remove_from_storage(*key*, *activities*, **args*, ***kwargs*)

trim(*key*, *length*)

stream_framework.storage.memory.**reverse_bisect_left**(*a*, *x*, *lo=0*, *hi=None*)

same as python bisect.bisect_left but for lists with reversed order

Subpackages

cassandra Package

cassandra Package

connection Module

stream_framework.storage.cassandra.connection.**setup_connection**()

redis Package

activity_storage Module

class stream_framework.storage.redis.activity_storage.**ActivityCache**(*key*, *redis=None*)

Bases: *stream_framework.storage.redis.structures.hash.ShardedHashCache*

key_format = 'activity:cache:%s'

class stream_framework.storage.redis.activity_storage.**RedisActivityStorage**(*serializer_class=None*, *activity_class=None*, ***options*)

Bases: *stream_framework.storage.base.BaseActivityStorage*

add_to_storage(*serialized_activities*, **args*, ***kwargs*)

default_serializer_class
alias of ActivitySerializer

flush()

get_cache()

get_from_storage(*activity_ids*, **args*, ***kwargs*)

get_key()

remove_from_storage(*activity_ids*, **args*, ***kwargs*)

connection Module

stream_framework.storage.redis.connection.**get_redis_connection**(*server_name='default'*)
Gets the specified redis connection

stream_framework.storage.redis.connection.**setup_redis**()
Starts the connection pool for all configured redis servers

timeline_storage Module

class stream_framework.storage.redis.timeline_storage.**RedisTimelineStorage**(*serializer_class=None*, *activity_class=None*, ***options*)

Bases: *stream_framework.storage.base.BaseTimelineStorage*

add_to_storage(*key*, *activities*, *batch_interface=None*)

contains(*key*, *activity_id*)

count(*key*)

delete(*key*)

get_batch_interface()

get_cache(*key*)

get_index_of(*key*, *activity_id*)

get_slice_from_storage(*key*, *start*, *stop*, *filter_kwargs=None*, *ordering_args=None*)
Returns a slice from the storage :param key: the redis key at which the sorted set is located :param start: the start :param stop: the stop :param filter_kwargs: a dict of filter kwargs :param ordering_args: a list of fields used for sorting

Example:: get_slice_from_storage('feed:13', 0, 10, {activity_id__lte=10})

remove_from_storage(*key*, *activities*, *batch_interface=None*)

trim(*key*, *length*, *batch_interface=None*)

class stream_framework.storage.redis.timeline_storage.**TimelineCache**(*key*, *redis=None*)

Bases: *stream_framework.storage.redis.structures.sorted_set.RedisSortedSetCache*

sort_asc = False

Subpackages

structures Package

base Module

class stream_framework.storage.redis.structures.base.**RedisCache**(*key*, *redis=None*)

Bases: object

The base for all redis data structures

delete()

get_key()

get_redis()
Only load the redis connection if we use it

key_format = 'redis:cache:%s'

redis
Only load the redis connection if we use it

set_redis(*value*)
Sets the redis connection

hash Module

class stream_framework.storage.redis.structures.hash.**BaseRedisHashCache**(*key*, *redis=None*)

Bases: *stream_framework.storage.redis.structures.base.RedisCache*

key_format = 'redis:base_hash_cache:%s'

class stream_framework.storage.redis.structures.hash.**FallbackHashCache**(*key*, *redis=None*)

Bases: *stream_framework.storage.redis.structures.hash.RedisHashCache*

Redis structure with fallback to the database

get_many(*fields*, *database_fallback=True*)

get_many_from_fallback(*missing_keys*)
Return a dictionary with the serialized values for the missing keys

key_format = 'redis:db_hash_cache:%s'

class stream_framework.storage.redis.structures.hash.**RedisHashCache**(*key*, *redis=None*)

Bases: *stream_framework.storage.redis.structures.hash.BaseRedisHashCache*

contains(*field*)
Uses hexists to see if the given field is present

count()
Returns the number of elements in the sorted set

delete_many(*fields*)

get(*field*)

get_key(**args*, ***kwargs*)

get_many(*fields*)

key_format = 'redis:hash_cache:%s'

keys()

set(*key*, *value*)

set_many(*key_value_pairs*)

class stream_framework.storage.redis.structures.hash.**ShardedDatabaseFallbackHashCache**(*key*, *redis=None*)

Bases: *stream_framework.storage.redis.structures.hash.ShardedHashCache*, *stream_framework.storage.redis.structures.hash.FallbackHashCache*

class stream_framework.storage.redis.structures.hash.**ShardedHashCache**(*key*, *redis=None*)

Bases: *stream_framework.storage.redis.structures.hash.RedisHashCache*

Use multiple keys instead of one so its easier to shard across redis machines

contains(*field*)

count()
Returns the number of elements in the sorted set

delete()
Delete all the base variations of the key

delete_many(*fields*)

get_key(*field*)
Takes something like field="3,79159750" and returns 7 as the index

get_keys()
Returns all possible keys

get_many(*fields*)

keys()
list all the keys, very slow, don't use too often

number_of_keys = 10

list Module

class stream_framework.storage.redis.structures.list.**BaseRedisListCache**(*key*, *redis=None*)

Bases: *stream_framework.storage.redis.structures.base.RedisCache*

Generic list functionality used for both the sorted set and list implementations

Retrieve the sorted list/sorted set by using python slicing

get_results(*start*, *stop*)

key_format = 'redis:base_list_cache:%s'

max_length = 100

class stream_framework.storage.redis.structures.list.**FallbackRedisListCache**(*key*, *redis=None*)

Bases: *stream_framework.storage.redis.structures.list.RedisListCache*

Redis list cache which after retrieving all items from redis falls back to a main data source (like the database)

cache(*fallback_results*)
Hook to write the results from the fallback to redis

get_fallback_results(*start*, *stop*)

get_redis_results (*start*, *stop*)
Returns the results from redis

Parameters

- **start** – the beginning
- **stop** – the end

get_results (*start*, *stop*)
Retrieves results from redis and the fallback datasource

key_format = 'redis:db_list_cache:%s'

overwrite (*fallback_results*)
Clear the cache and write the results from the fallback

class stream_framework.storage.redis.structures.list.**RedisListCache** (*key*, *redis=None*)
Bases: *stream_framework.storage.redis.structures.list.BaseRedisListCache*

append (*value*)

append_many (*values*)

count ()

get_results (*start*, *stop*)

key_format = 'redis:list_cache:%s'

max_items = 1000
the maximum number of items the list stores

remove (*value*)

remove_many (*values*)

trim ()
Removes the old items in the list

sorted_set Module

class stream_framework.storage.redis.structures.sorted_set.**RedisSortedSetCache** (*key*, *redis=None*)
Bases: *stream_framework.storage.redis.structures.list.BaseRedisListCache*, *stream_framework.storage.redis.structures.hash.BaseRedisHashCache*

add (*score*, *key*)

add_many (*score_value_pairs*)
StrictRedis so it expects score1, name1

contains (*value*)
Uses zscore to see if the given activity is present in our sorted set

count ()
Returns the number of elements in the sorted set

get_results (*start=None*, *stop=None*, *min_score=None*, *max_score=None*)
Retrieve results from redis using zrevrange O(log(N)+M) with N being the number of elements in the sorted set and M the number of elements returned.

index_of (*value*)
Returns the index of the given value

remove_by_scores(*scores*)

remove_many(*values*)
values

sort_asc = False

trim(*max_length=None*)
Trim the sorted set to max length zremrangebyscore

verbs Package

verbs Package

stream_framework.verbs.**get_verb_by_id**(*verb_id*)

stream_framework.verbs.**get_verb_storage**()

stream_framework.verbs.**register**(*verb*)
Registers the given verb class

base Module

class stream_framework.verbs.base.**Add**
Bases: *stream_framework.verbs.base.Verb*

id = 4

infinitive = 'add'

past_tense = 'added'

class stream_framework.verbs.base.**Comment**
Bases: *stream_framework.verbs.base.Verb*

id = 2

infinitive = 'comment'

past_tense = 'commented'

class stream_framework.verbs.base.**Follow**
Bases: *stream_framework.verbs.base.Verb*

id = 1

infinitive = 'follow'

past_tense = 'followed'

class stream_framework.verbs.base.**Love**
Bases: *stream_framework.verbs.base.Verb*

id = 3

infinitive = 'love'

past_tense = 'loved'

class stream_framework.verbs.base.**Verb**
Bases: object

Every activity has a verb and an object. Nomenclatura is loosly based on http://activitystrea.ms/specs/atom/1.0/#activity.summary

id = 0

serialize()

8.2 Indices and tables

- genindex
- modindex
- search

S

stream_framework, 29
stream_framework.activity, 29
stream_framework.aggregators.base, 32
stream_framework.default_settings, 31
stream_framework.exceptions, 31
stream_framework.feeds.aggregated_feed, 37
stream_framework.feeds.aggregated_feed.base, 38
stream_framework.feeds.aggregated_feed.notification_feed, 39
stream_framework.feeds.aggregated_feed.redis, 39
stream_framework.feeds.base, 34
stream_framework.feeds.memory, 37
stream_framework.feeds.redis, 37
stream_framework.settings, 31
stream_framework.storage.base, 40
stream_framework.storage.cassandra, 44
stream_framework.storage.cassandra.connection, 44
stream_framework.storage.memory, 43
stream_framework.storage.redis.activity_storage, 44
stream_framework.storage.redis.connection, 45
stream_framework.storage.redis.structures.base, 46
stream_framework.storage.redis.structures.hash, 46
stream_framework.storage.redis.structures.list, 47
stream_framework.storage.redis.structures.sorted_set, 48
stream_framework.storage.redis.timeline_storage, 45
stream_framework.utils, 31
stream_framework.verbs, 49
stream_framework.verbs.base, 49

A

activities_to_ids() (stream_framework.storage.base.BaseStorage method), 41
Activity (class in stream_framework.activity), 29
activity_class (stream_framework.aggregators.base.BaseAggregator attribute), 32
activity_class (stream_framework.feeds.base.BaseFeed attribute), 35
activity_class (stream_framework.storage.base.BaseStorage attribute), 41
activity_count (stream_framework.activity.AggregatedActivity attribute), 29
activity_ids (stream_framework.activity.AggregatedActivity attribute), 29
activity_serializer (stream_framework.feeds.aggregated_feed.notification_feed.NotificationFeed attribute), 39
activity_serializer (stream_framework.feeds.aggregated_feed.redis.RedisAggregatedFeed attribute), 39
activity_serializer (stream_framework.feeds.base.BaseFeed attribute), 35
activity_serializer (stream_framework.feeds.redis.RedisFeed attribute), 37
activity_storage_class (stream_framework.feeds.aggregated_feed.notification_feed.NotificationFeed attribute), 39
activity_storage_class (stream_framework.feeds.aggregated_feed.redis.RedisAggregatedFeed attribute), 39
activity_storage_class (stream_framework.feeds.base.BaseFeed attribute), 35
activity_storage_class (stream_framework.feeds.memory.Feed attribute), 37
activity_storage_class (stream_framework.feeds.redis.RedisFeed attribute), 37
activity_to_id() (stream_framework.storage.base.BaseStorage method), 41
ActivityCache (class in stream_framework.storage.redis.activity_storage), 44
ActivityNotFound, 31
actor_count (stream_framework.activity.AggregatedActivity attribute), 29
actor_ids (stream_framework.activity.AggregatedActivity attribute), 30
Add (class in stream_framework.verbs.base), 49
add() (stream_framework.feeds.base.BaseFeed method), 35
add() (stream_framework.storage.base.BaseActivityStorage method), 40
add() (stream_framework.storage.base.BaseTimelineStorage method), 42
add() (stream_framework.storage.redis.structures.sorted_set.RedisSortedSet method), 48
add_many() (stream_framework.feeds.aggregated_feed.base.AggregatedFeed method), 38
add_many() (stream_framework.feeds.aggregated_feed.notification_feed.NotificationFeed method), 39
add_many() (stream_framework.feeds.base.BaseFeed method), 35
add_many() (stream_framework.storage.base.BaseActivityStorage method), 41
add_many() (stream_framework.storage.base.BaseTimelineStorage method), 42
add_many() (stream_framework.storage.redis.structures.sorted_set.RedisSortedSet method), 48
add_many_aggregated() (stream_framework.feeds.aggregated_feed.base.AggregatedFeed method), 38
add_to_storage() (stream_framework.storage.base.BaseActivityStorage method), 41
add_to_storage() (stream_framework.storage.memory.InMemoryActivityStorage method), 43
add_to_storage() (stream_framework.storage.memory.InMemoryTimelineStorage method), 44
add_to_storage() (stream_framework.storage.redis.activity_storage.RedisActivityStorage method), 44
add_to_storage() (stream_framework.storage.redis.timeline_storage.RedisTimelineStorage method), 45
aggregate() (stream_framework.aggregators.base.BaseAggregator method), 32
aggregated_activity_class (stream_framework.aggregators.base.BaseAggregator attribute), 33

aggregated_activity_class (stream_framework.feeds.aggregated_feed.base.AggregatedFeed attribute), 38
aggregated_activity_class (stream_framework.storage.base.BaseStorage attribute), 42
AggregatedActivity (class in stream_framework.activity), 29
AggregatedFeed (class in stream_framework.feeds.aggregated_feed.base), 38
aggregator_class (stream_framework.feeds.aggregated_feed.base.AggregatedFeed attribute), 38
append() (stream_framework.activity.AggregatedActivity method), 30
append() (stream_framework.storage.redis.structures.list.RedisListCache method), 48
append_many() (stream_framework.storage.redis.structures.list.RedisListCache method), 48

B

BaseActivity (class in stream_framework.activity), 30
BaseActivityStorage (class in stream_framework.storage.base), 40
BaseAggregator (class in stream_framework.aggregators.base), 32
BaseFeed (class in stream_framework.feeds.base), 34
BaseRedisHashCache (class in stream_framework.storage.redis.structures.hash), 46
BaseRedisListCache (class in stream_framework.storage.redis.structures.list), 47
BaseStorage (class in stream_framework.storage.base), 41
BaseTimelineStorage (class in stream_framework.storage.base), 42

C

cache() (stream_framework.storage.redis.structures.list.FallbackRedisListCache method), 47
chunks() (in module stream_framework.utils), 32
Comment (class in stream_framework.verbs.base), 49
contains() (stream_framework.activity.AggregatedActivity method), 30
contains() (stream_framework.feeds.aggregated_feed.base.AggregatedFeed method), 38
contains() (stream_framework.storage.memory.InMemoryTimelineStorage method), 44
contains() (stream_framework.storage.redis.structures.hash.RedisHashCache method), 46
contains() (stream_framework.storage.redis.structures.hash.ShardedHashCache method), 47
contains() (stream_framework.storage.redis.structures.sorted_set.RedisSorte method), 48
contains() (stream_framework.storage.redis.timeline_storage.RedisTimeline method), 45
count() (stream_framework.feeds.base.BaseFeed method), 35
count() (stream_framework.storage.base.BaseTimelineStorage method), 42
count() (stream_framework.storage.memory.InMemoryTimelineStorage method), 44
count() (stream_framework.storage.redis.structures.hash.RedisHashCache method), 46
count() (stream_framework.storage.redis.structures.hash.ShardedHashCache method), 47
count() (stream_framework.storage.redis.structures.list.RedisListCache method), 48
count() (stream_framework.storage.redis.structures.sorted_set.RedisSortedS method), 48
count() (stream_framework.storage.redis.timeline_storage.RedisTimelineSto method), 45
count_format (stream_framework.feeds.aggregated_feed.notification_feed.N attribute), 39
count_unseen() (stream_framework.feeds.aggregated_feed.notification_feed method), 39

D

datetime_to_epoch() (in module stream_framework.utils), 32
default_serializer_class (stream_framework.storage.base.BaseStorage attribute), 42
default_serializer_class (stream_framework.storage.base.BaseTimelineStora attribute), 42
default_serializer_class (stream_framework.storage.redis.activity_storage.R attribute), 45
DehydratedActivity (class in stream_framework.activity), 30
delete() (stream_framework.feeds.base.BaseFeed method), 35
delete() (stream_framework.storage.base.BaseTimelineStorage method), 43
delete() (stream_framework.storage.memory.InMemoryTimelineStorage method), 44
delete() (stream_framework.storage.redis.structures.base.RedisCache method), 46
delete() (stream_framework.storage.redis.structures.hash.ShardedHashCach method), 47
delete() (stream_framework.storage.redis.timeline_storage.RedisTimelineSt method), 45
delete_many() (stream_framework.storage.redis.structures.hash.RedisHashC method), 46
delete_many() (stream_framework.storage.redis.structures.hash.ShardedHas method), 47
denormalize_count() (stream_framework.feeds.aggregated_feed.notification method), 39

deserialize_activities() (stream_framework.storage.base.BaseStorage method), 42
DuplicateActivityException, 31

E

epoch_to_datetime() (in module stream_framework.utils), 32

F

FallbackHashCache (class in stream_framework.storage.redis.structures.hash), 46
FallbackRedisListCache (class in stream_framework.storage.redis.structures.list), 47
Feed (class in stream_framework.feeds.memory), 37
filter() (stream_framework.feeds.base.BaseFeed method), 35
filtering_supported (stream_framework.feeds.base.BaseFeed attribute), 35
filtering_supported (stream_framework.feeds.redis.RedisFeed attribute), 37
flush() (stream_framework.feeds.base.BaseFeed class method), 35
flush() (stream_framework.storage.base.BaseStorage method), 42
flush() (stream_framework.storage.memory.InMemoryActivityStorage method), 44
flush() (stream_framework.storage.redis.activity_storage.RedisActivityStorage method), 45
Follow (class in stream_framework.verbs.base), 49

G

get() (stream_framework.storage.base.BaseActivityStorage method), 41
get() (stream_framework.storage.redis.structures.hash.RedisHashCache method), 46
get() (stream_framework.utils.LRUCache method), 31
get_activity_slice() (stream_framework.feeds.base.BaseFeed method), 35
get_activity_storage() (stream_framework.feeds.base.BaseFeed class method), 35
get_aggregator() (stream_framework.feeds.aggregated_feed.base.AggregatedFeed method), 38
get_batch_interface() (stream_framework.storage.base.BaseTimelineStorage method), 43
get_batch_interface() (stream_framework.storage.memory.InMemoryTimelineStorage class method), 44
get_batch_interface() (stream_framework.storage.redis.timeline_storage.RedisTimelineStorage method), 45
get_cache() (stream_framework.storage.redis.activity_storage.RedisActivityStorage method), 45
get_cache() (stream_framework.storage.redis.timeline_storage.RedisTimelineStorage method), 45
get_class_from_string() (in module stream_framework.utils), 32
get_dehydrated() (stream_framework.activity.Activity method), 29
get_dehydrated() (stream_framework.activity.AggregatedActivity method), 30
get_denormalized_count() (stream_framework.feeds.aggregated_feed.notification_feed.Notif method), 39
get_fallback_results() (stream_framework.storage.redis.structures.list.Fallba method), 47
get_from_storage() (stream_framework.storage.base.BaseActivityStorage method), 41
get_from_storage() (stream_framework.storage.memory.InMemoryActivity method), 44
get_from_storage() (stream_framework.storage.redis.activity_storage.Redis method), 45
get_group() (stream_framework.aggregators.base.BaseAggregator method), 33
get_group() (stream_framework.aggregators.base.NotificationAggregator method), 33
get_group() (stream_framework.aggregators.base.RecentVerbAggregator method), 34
get_hydrated() (stream_framework.activity.AggregatedActivity method), 30
get_hydrated() (stream_framework.activity.DehydratedActivity method), 31
get_index_of() (stream_framework.storage.base.BaseTimelineStorage method), 43
get_index_of() (stream_framework.storage.memory.InMemoryTimelineStor method), 44
get_index_of() (stream_framework.storage.redis.timeline_storage.RedisTim method), 45
get_key() (stream_framework.storage.redis.activity_storage.RedisActivityS method), 45
get_key() (stream_framework.storage.redis.structures.base.RedisCache method), 46
get_key() (stream_framework.storage.redis.structures.hash.RedisHashCache method), 46
get_key() (stream_framework.storage.redis.structures.hash.ShardedHashCa method), 47
get_keys() (stream_framework.storage.redis.structures.hash.ShardedHashCa method), 47
get_many() (stream_framework.storage.base.BaseActivityStorage method), 41
get_many() (stream_framework.storage.redis.structures.hash.FallbackHashC method), 46
get_many() (stream_framework.storage.redis.structures.hash.RedisHashCac method), 46
get_many() (stream_framework.storage.redis.structures.hash.ShardedHashC method), 47
get_many_from_fallback() (stream_framework.storage.redis.structures.hash.FallbackHashCa method), 46

get_metrics_instance() (in module stream_framework.utils), 32
get_redis() (stream_framework.storage.redis.structures.base.RedisCache method), 46
get_redis_connection() (in module stream_framework.storage.redis.connection), 45
get_redis_results() (stream_framework.storage.redis.structures.list.FallbackRedisListCache method), 47
get_results() (stream_framework.storage.redis.structures.list.BaseRedisListCache method), 47
get_results() (stream_framework.storage.redis.structures.list.FallbackRedisListCache method), 48
get_results() (stream_framework.storage.redis.structures.list.RedisListCache method), 48
get_results() (stream_framework.storage.redis.structures.sorted_set.RedisSortedSetCache method), 48
get_slice() (stream_framework.storage.base.BaseTimelineStorage method), 43
get_slice_from_storage() (stream_framework.storage.base.BaseTimelineStorage method), 43
get_slice_from_storage() (stream_framework.storage.memory.InMemoryTimelineStorage method), 44
get_slice_from_storage() (stream_framework.storage.redis.timeline_storage.RedisTimelineStorage method), 45
get_timeline_batch_interface() (stream_framework.feeds.base.BaseFeed class method), 35
get_timeline_storage() (stream_framework.feeds.base.BaseFeed class method), 36
get_timeline_storage_options() (stream_framework.feeds.aggregated_feed.base.AggregatedFeed class method), 38
get_timeline_storage_options() (stream_framework.feeds.base.BaseFeed class method), 36
get_timeline_storage_options() (stream_framework.feeds.redis.RedisFeed class method), 37
get_verb_by_id() (in module stream_framework.verbs), 49
get_verb_storage() (in module stream_framework.verbs), 49
group_activities() (stream_framework.aggregators.base.BaseAggregator method), 33

H

hydrate_activities() (stream_framework.feeds.base.BaseFeed method), 36

I

id (stream_framework.verbs.base.Add attribute), 49
id (stream_framework.verbs.base.Comment attribute), 49
id (stream_framework.verbs.base.Follow attribute), 49
id (stream_framework.verbs.base.Love attribute), 49
id (stream_framework.verbs.base.Verb attribute), 50
import_global_module() (in module stream_framework.settings), 31
index_of() (stream_framework.feeds.base.BaseFeed method), 36
index_of() (stream_framework.storage.base.BaseTimelineStorage method), 43
index_of() (stream_framework.storage.redis.structures.sorted_set.RedisSort method), 48
infinitive (stream_framework.verbs.base.Add attribute), 49
infinitive (stream_framework.verbs.base.Comment attribute), 49
infinitive (stream_framework.verbs.base.Follow attribute), 49
infinitive (stream_framework.verbs.base.Love attribute), 49
InMemoryActivityStorage (class in stream_framework.storage.memory), 43
InMemoryTimelineStorage (class in stream_framework.storage.memory), 44
insert_activities() (stream_framework.feeds.base.BaseFeed class method), 36
insert_activity() (stream_framework.feeds.base.BaseFeed class method), 36
is_read() (stream_framework.activity.AggregatedActivity method), 30
is_seen() (stream_framework.activity.AggregatedActivity method), 30

K

key_format (stream_framework.feeds.aggregated_feed.notification_feed.No attribute), 39
key_format (stream_framework.feeds.base.BaseFeed attribute), 36
key_format (stream_framework.feeds.base.UserBaseFeed attribute), 37
key_format (stream_framework.storage.redis.activity_storage.ActivityCache attribute), 44
key_format (stream_framework.storage.redis.structures.base.RedisCache attribute), 46
key_format (stream_framework.storage.redis.structures.hash.BaseRedisHas attribute), 46
key_format (stream_framework.storage.redis.structures.hash.FallbackHashC attribute), 46
key_format (stream_framework.storage.redis.structures.hash.RedisHashCac attribute), 46
key_format (stream_framework.storage.redis.structures.list.BaseRedisListC attribute), 47

key_format (stream_framework.storage.redis.structures.list.FallbackRedisList attribute), 48
key_format (stream_framework.storage.redis.structures.list.RedisListCache attribute), 48
keys() (stream_framework.storage.redis.structures.hash.RedisHashCache method), 46
keys() (stream_framework.storage.redis.structures.hash.ShardedHashCache method), 47

L

last_activities (stream_framework.activity.AggregatedActivity attribute), 30
last_activity (stream_framework.activity.AggregatedActivity attribute), 30
lock_format (stream_framework.feeds.aggregated_feed.notification_feed.NotificationFeed attribute), 40
Love (class in stream_framework.verbs.base), 49
LRUCache (class in stream_framework.utils), 31

M

make_list_unique() (in module stream_framework.utils), 32
mark_all() (stream_framework.feeds.aggregated_feed.notification_feed.NotificationFeed method), 40
max_aggregated_activities_length (stream_framework.activity.AggregatedActivity attribute), 30
max_items (stream_framework.storage.redis.structures.list.RedisListCache attribute), 48
max_length (stream_framework.feeds.aggregated_feed.notification_feed.NotificationFeed attribute), 40
max_length (stream_framework.feeds.base.BaseFeed attribute), 36
max_length (stream_framework.feeds.base.UserBaseFeed attribute), 37
max_length (stream_framework.storage.redis.structures.list.BaseRedisListCache attribute), 47
memoized (class in stream_framework.utils), 32
merge() (stream_framework.aggregators.base.BaseAggregator method), 33
merge_max_length (stream_framework.feeds.aggregated_feed.base.AggregatedFeed attribute), 38
metrics (stream_framework.storage.base.BaseStorage attribute), 42

N

needs_hydration() (stream_framework.feeds.base.BaseFeed method), 36
NotificationActivity (class in stream_framework.activity), 31
NotificationAggregator (class in stream_framework.aggregators.base), 33
NotificationFeed (class in stream_framework.feeds.aggregated_feed.notification_feed), 39
number_of_keys (stream_framework.storage.redis.structures.hash.ShardedHashCache attribute), 47

O

object_ids (stream_framework.activity.AggregatedActivity attribute), 30
on_update_feed() (stream_framework.feeds.base.BaseFeed method), 36
order_by() (stream_framework.feeds.base.BaseFeed method), 36
ordering_supported (stream_framework.feeds.base.BaseFeed attribute), 36
ordering_supported (stream_framework.feeds.redis.RedisFeed attribute), 37
other_actor_count (stream_framework.activity.AggregatedActivity attribute), 30
overwrite() (stream_framework.storage.redis.structures.list.FallbackRedisList method), 48

P

past_tense (stream_framework.verbs.base.Add attribute), 49
past_tense (stream_framework.verbs.base.Comment attribute), 49
past_tense (stream_framework.verbs.base.Follow attribute), 49
past_tense (stream_framework.verbs.base.Love attribute), 49
publish_count() (stream_framework.feeds.aggregated_feed.notification_feed.NotificationFeed method), 40
pubsub_main_channel (stream_framework.feeds.aggregated_feed.notification_feed.NotificationFeed attribute), 40

R

rank() (stream_framework.aggregators.base.BaseAggregator method), 33
rank() (stream_framework.aggregators.base.RecentRankMixin method), 33
RecentRankMixin (class in stream_framework.aggregators.base), 33
RecentVerbAggregator (class in stream_framework.aggregators.base), 33
redis (stream_framework.storage.redis.structures.base.RedisCache attribute), 46
redis_server (stream_framework.feeds.redis.RedisFeed attribute), 37
RedisActivityStorage (class in stream_framework.storage.redis.activity_storage), 44

RedisAggregatedFeed (class in stream_framework.feeds.aggregated_feed.redis), 39
RedisCache (class in stream_framework.storage.redis.structures.base), 46
RedisFeed (class in stream_framework.feeds.redis), 37
RedisHashCache (class in stream_framework.storage.redis.structures.hash), 46
RedisListCache (class in stream_framework.storage.redis.structures.list), 48
RedisNotificationFeed (class in stream_framework.feeds.aggregated_feed.notification_feed), 40
RedisSortedSetCache (class in stream_framework.storage.redis.structures.sorted_set), 48
RedisTimelineStorage (class in stream_framework.storage.redis.timeline_storage), 45
register() (in module stream_framework.verbs), 49
remove() (stream_framework.activity.AggregatedActivity method), 30
remove() (stream_framework.feeds.base.BaseFeed method), 36
remove() (stream_framework.storage.base.BaseActivityStorage method), 41
remove() (stream_framework.storage.base.BaseTimelineStorage method), 43
remove() (stream_framework.storage.redis.structures.list.RedisListCache method), 48
remove_activity() (stream_framework.feeds.base.BaseFeed class method), 36
remove_by_scores() (stream_framework.storage.redis.structures.sorted_set.RedisSortedSetCache method), 48
remove_from_storage() (stream_framework.storage.base.BaseActivityStorage method), 41
remove_from_storage() (stream_framework.storage.base.BaseTimelineStorage method), 43
remove_from_storage() (stream_framework.storage.memory.InMemoryActivityStorage method), 44
remove_from_storage() (stream_framework.storage.memory.InMemoryTimelineStorage method), 44
remove_from_storage() (stream_framework.storage.redis.activity_storage.RedisActivityStorage method), 45
remove_from_storage() (stream_framework.storage.redis.timeline_storage.RedisTimelineStorage method), 45
remove_many() (stream_framework.activity.AggregatedActivity method), 30
remove_many() (stream_framework.feeds.aggregated_feed.base.AggregatedFeed method), 38
remove_many() (stream_framework.feeds.base.BaseFeed method), 36
remove_many() (stream_framework.storage.base.BaseActivityStorage method), 41
remove_many() (stream_framework.storage.base.BaseTimelineStorage method), 43
remove_many() (stream_framework.storage.redis.structures.list.RedisListCache method), 48
remove_many() (stream_framework.storage.redis.structures.sorted_set.RedisSortedSetCache method), 49
remove_many_aggregated() (stream_framework.feeds.aggregated_feed.base.AggregatedFeed method), 39
reverse_bisect_left() (in module stream_framework.storage.memory), 44

S

serialization_id (stream_framework.activity.Activity attribute), 29
serialization_id (stream_framework.activity.AggregatedActivity attribute), 30
SerializationException, 31
serialize() (stream_framework.verbs.base.Verb method), 50
serialize_activities() (stream_framework.storage.base.BaseStorage method), 42
serialize_activity() (stream_framework.storage.base.BaseStorage method), 42
serializer (stream_framework.storage.base.BaseStorage attribute), 42
set() (stream_framework.storage.redis.structures.hash.RedisHashCache method), 46
set() (stream_framework.utils.LRUCache method), 32
set_denormalized_count() (stream_framework.feeds.aggregated_feed.notification_feed.NotificationFeed method), 40
set_many() (stream_framework.storage.redis.structures.hash.RedisHashCache method), 47
set_redis() (stream_framework.storage.redis.structures.base.RedisCache method), 46
setup_connection() (in module stream_framework.storage.cassandra.connection), 44
setup_redis() (in module stream_framework.storage.redis.connection), 45
ShardedDatabaseFallbackHashCache (class in stream_framework.storage.redis.structures.hash), 46
ShardedHashCache (class in stream_framework.storage.redis.structures.hash), 47
sort_asc (stream_framework.storage.redis.structures.sorted_set.RedisSortedSetCache attribute), 49
sort_asc (stream_framework.storage.redis.timeline_storage.TimelineCache attribute), 45

stream_framework (module), 29
stream_framework.activity (module), 29
stream_framework.aggregators.base (module), 32
stream_framework.default_settings (module), 31
stream_framework.exceptions (module), 31
stream_framework.feeds.aggregated_feed (module), 37
stream_framework.feeds.aggregated_feed.base (module), 38
stream_framework.feeds.aggregated_feed.notification_feed (module), 39
stream_framework.feeds.aggregated_feed.redis (module), 39
stream_framework.feeds.base (module), 34
stream_framework.feeds.memory (module), 37
stream_framework.feeds.redis (module), 37
stream_framework.settings (module), 31
stream_framework.storage.base (module), 40
stream_framework.storage.cassandra (module), 44
stream_framework.storage.cassandra.connection (module), 44
stream_framework.storage.memory (module), 43
stream_framework.storage.redis.activity_storage (module), 44
stream_framework.storage.redis.connection (module), 45
stream_framework.storage.redis.structures.base (module), 46
stream_framework.storage.redis.structures.hash (module), 46
stream_framework.storage.redis.structures.list (module), 47
stream_framework.storage.redis.structures.sorted_set (module), 48
stream_framework.storage.redis.timeline_storage (module), 45
stream_framework.utils (module), 31
stream_framework.verbs (module), 49
stream_framework.verbs.base (module), 49

T

timeline_serializer (stream_framework.feeds.aggregated_feed.base.AggregatedFeed attribute), 39
timeline_serializer (stream_framework.feeds.aggregated_feed.notification_feed.NotificationFeed attribute), 40
timeline_serializer (stream_framework.feeds.aggregated_feed.redis.RedisAggregatedFeed attribute), 39
timeline_serializer (stream_framework.feeds.base.BaseFeed attribute), 36
timeline_storage_class (stream_framework.feeds.aggregated_feed.notification_feed.RedisNotificationFeed attribute), 40
timeline_storage_class (stream_framework.feeds.aggregated_feed.redis.RedisAggregatedFeed attribute), 39
timeline_storage_class (stream_framework.feeds.base.BaseFeed attribute), 36
timeline_storage_class (stream_framework.feeds.memory.Feed attribute), 37
timeline_storage_class (stream_framework.feeds.redis.RedisFeed attribute), 37
TimelineCache (class in stream_framework.storage.redis.timeline_storage), 45
trim() (stream_framework.feeds.base.BaseFeed method), 36
trim() (stream_framework.storage.base.BaseTimelineStorage method), 43
trim() (stream_framework.storage.memory.InMemoryTimelineStorage method), 44
trim() (stream_framework.storage.redis.structures.list.RedisListCache method), 48
trim() (stream_framework.storage.redis.structures.sorted_set.RedisSortedSe method), 49
trim() (stream_framework.storage.redis.timeline_storage.RedisTimelineSto method), 45
trim_chance (stream_framework.feeds.base.BaseFeed attribute), 36

U

update_read_at() (stream_framework.activity.AggregatedActivity method), 30
update_seen_at() (stream_framework.activity.AggregatedActivity method), 30
UserBaseFeed (class in stream_framework.feeds.base), 36

V

Verb (class in stream_framework.verbs.base), 49
verb (stream_framework.activity.AggregatedActivity attribute), 30
verbs (stream_framework.activity.AggregatedActivity attribute), 30

W

warn_on_duplicate() (in module stream_framework.utils), 32
warn_on_error() (in module stream_framework.utils), 32