

---

# **stream***frameworkDocumentation*

## ***Release***

**Thierry Schellenbach**

**Mar 03, 2017**



---

## Contents

---

<b>1</b>	<b>What can you build?</b>	<b>3</b>
<b>2</b>	<b>GetStream.io</b>	<b>5</b>
<b>3</b>	<b>Consultancy</b>	<b>7</b>
<b>4</b>	<b>Using Stream Framework</b>	<b>9</b>
<b>5</b>	<b>Features</b>	<b>11</b>
<b>6</b>	<b>Background Articles</b>	<b>13</b>
<b>7</b>	<b>Documentation</b>	<b>15</b>
7.1	Installation . . . . .	15
7.2	Feed setup . . . . .	15
7.3	Adding data . . . . .	16
7.4	Verbs . . . . .	16
7.4.1	Adding new verbs . . . . .	16
7.4.2	Getting verbs . . . . .	17
7.5	Querying feeds . . . . .	17
7.6	Settings . . . . .	17
7.6.1	Redis Settings . . . . .	18
7.6.2	Cassandra Settings . . . . .	18
7.6.3	Metric Settings . . . . .	18
7.7	Metrics . . . . .	19
7.7.1	Sending metrics to Statsd . . . . .	19
7.7.2	Custom metric classes . . . . .	19
7.8	Testing Stream Framework . . . . .	19
7.9	Support . . . . .	20
7.10	Activity class . . . . .	20
7.10.1	Activity storage strategies . . . . .	20
7.10.2	Extend the activity class . . . . .	21
7.10.3	Activity serialization . . . . .	21
7.10.4	Activity order and uniqueness . . . . .	21
7.10.5	Aggregated activities . . . . .	21
7.11	Choosing a storage layer . . . . .	21
7.11.1	Redis (2.7 or newer) . . . . .	22

7.11.2	Cassandra (2.0 or newer)	22
7.12	Background Tasks with celery	22
7.12.1	Prioritise fanouts	23
7.12.2	Celery and Django	23
7.12.3	Using other job queue libraries	23
7.13	Tutorial: building a notification feed	24
7.13.1	What is a notification system?	24
7.13.2	Tutorial	25
7.14	Stream Framework Design	26
7.15	Cassandra storage backend	27
7.15.1	Create keyspace and columnfamilies	28
7.15.2	Use a custom activity model	28
<b>8</b>	<b>API Docs</b>	<b>29</b>
8.1	Stream Framework API Docs	29
8.1.1	stream_framework Package	29
8.1.2	activity Module	29
8.1.3	default_settings Module	31
8.1.4	exceptions Module	31
8.1.5	settings Module	31
8.1.6	tasks Module	32
8.1.7	utils Module	32
8.1.8	Subpackages	32
8.1.8.1	aggregators Package	32
8.1.8.2	feed_managers Package	34
8.1.8.3	feeds Package	34
8.1.8.4	storage Package	39
8.1.8.5	verbs Package	43
8.2	Indices and tables	44
	<b>Python Module Index</b>	<b>45</b>

## 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](https://getstream.io). The hosted service is highly optimized and allows you to start building your application immediately. It saves you the hassle 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](http://getstream.io)

### Authors

- Thierry Schellenbach ([thierry at getstream.io](mailto:thierry@getstream.io))
- Tommaso Barbugli ([tommaso at getstream.io](mailto:tommaso@getstream.io))
- Guyon Morée

### Resources

- [Documentation](#)
- [Bug Tracker](#)
- [Code](#)
- [Mailing List](#)
- IRC ([irc.freenode.net](http://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.

---

### 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 doesn't 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](#)



### 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.

### 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 streamframework.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()
```

## 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 occurred
)
```

The names for these fields are based on the [activity stream spec](#).

## Verbs

### 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

## 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)

```

## 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')

```

## Settings

---

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

---

## 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
    },
}
```

## 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](http://datastax.github.io/python-driver/_modules/cassandra/cluster.html#Cluster))

Defaults to { }

## 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 `{}`

## 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`

### 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'
}
```

### 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.

## 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.

## 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](#)

## 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

## 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).

## 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 streamframework.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 streamframework.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
```

## Activity serialization

## Activity order and uniqueness

## Aggregated activities

## 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.

## 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 therefore 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 manually move or recreate the data.

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

## 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.

## 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*

---

## 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
```

## 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.

## Using other job queue libraries

As of today background processing is tied to celery.

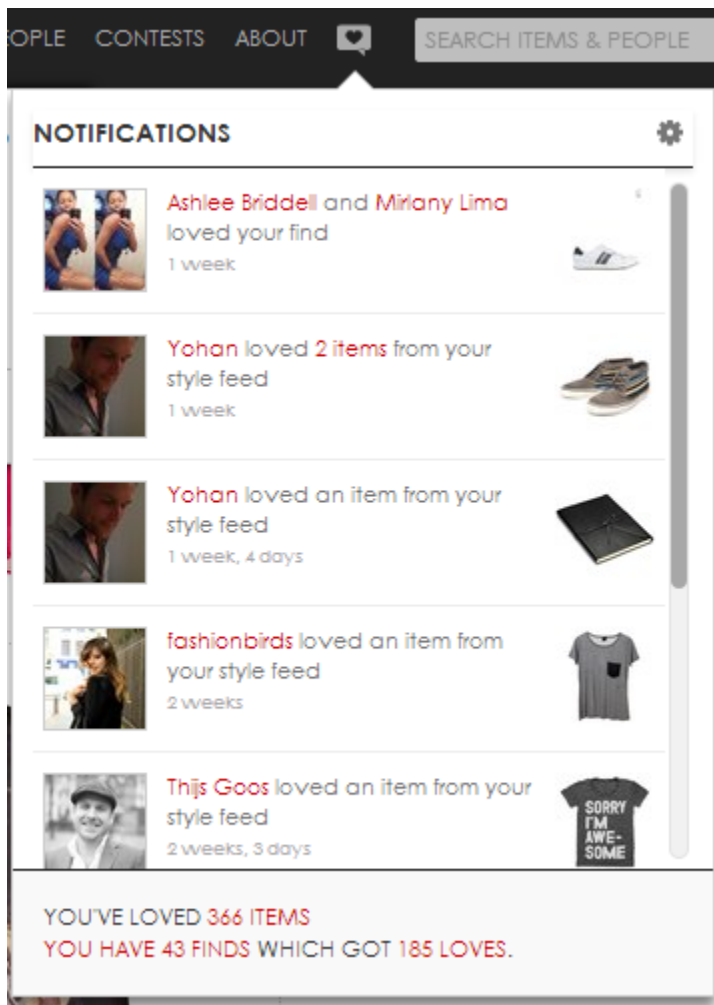
While we are not planning to support different queue jobs libraries in the near future using something different than celery should be quite easy and can be mostly done subclassing the feeds manager.

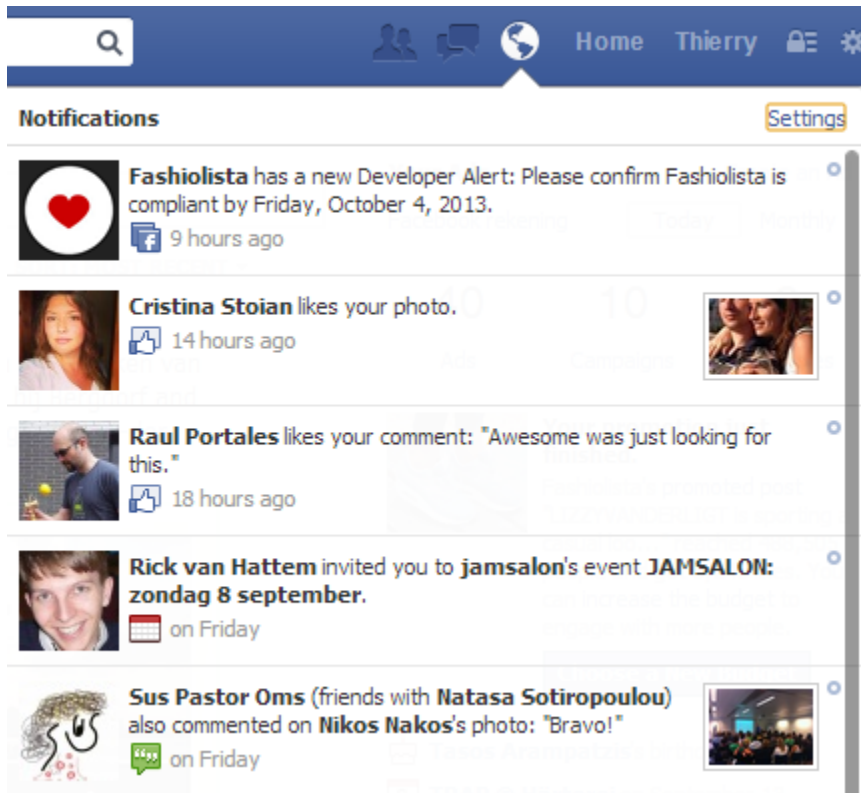
## Tutorial: building a notification feed

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

### 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.





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.

---

## 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)
```

## 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 meaningful 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 Cassandra/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)

## **Cassandra storage backend**

This document is specific to the Cassandra backend.

## 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.

## 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 streamframework.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).



## Stream Framework API Docs

### `stream_framework` Package

#### `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 = 1373266755000000000042008` 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 = 1373266755000000000042008 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 versions 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`

**default\_settings Module****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

**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

## tasks Module

## 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*)

## 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** **activities** – 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(activity 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 (referred 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\_ids)'

**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

## 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

## notification\_feed Module

## 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*) activ-

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**

**redis Package**

**activity\_storage Module**

**connection Module**

**timeline\_storage Module**

## Subpackages

**structures Package**

**base Module**

**hash Module**

**list Module**

**sorted\_set Module**

**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 loosely based on http://activitystrea.ms/specs/atom/1.0/#activity.summary

    id = 0

    serialize()
```

## 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`,  
[37](#)
- `stream_framework.feeds.base`, [34](#)
- `stream_framework.feeds.memory`, [37](#)
- `stream_framework.settings`, [31](#)
- `stream_framework.storage.base`, [39](#)
- `stream_framework.storage.cassandra`, [43](#)
- `stream_framework.storage.memory`, [42](#)
- `stream_framework.utils`, [32](#)
- `stream_framework.verbs`, [43](#)
- `stream_framework.verbs.base`, [43](#)



## A

activities\_to\_ids() (stream\_framework.storage.base.BaseStorage method), 40  
 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), 40  
 activity\_count (stream\_framework.activity.AggregatedActivity attribute), 30  
 activity\_ids (stream\_framework.activity.AggregatedActivity attribute), 30  
 activity\_serializer (stream\_framework.feeds.base.BaseFeed attribute), 35  
 activity\_storage\_class (stream\_framework.feeds.base.BaseFeed attribute), 35  
 activity\_storage\_class (stream\_framework.feeds.memory.Feed attribute), 37  
 activity\_to\_id() (stream\_framework.storage.base.BaseStorage method), 40  
 ActivityNotFound, 31  
 actor\_count (stream\_framework.activity.AggregatedActivity attribute), 30  
 actor\_ids (stream\_framework.activity.AggregatedActivity attribute), 30  
 Add (class in stream\_framework.verbs.base), 43  
 add() (stream\_framework.feeds.base.BaseFeed method), 35  
 add() (stream\_framework.storage.base.BaseActivityStorage method), 39  
 add() (stream\_framework.storage.base.BaseTimelineStorage method), 41  
 add\_many() (stream\_framework.feeds.aggregated\_feed.base.AggregatedFeed method), 38  
 add\_many() (stream\_framework.feeds.base.BaseFeed method), 35  
 add\_many() (stream\_framework.storage.base.BaseActivityStorage method), 39  
 add\_many() (stream\_framework.storage.base.BaseTimelineStorage method), 41  
 add\_many\_aggregated() (stream\_framework.feeds.aggregated\_feed.base.AggregatedFeed method), 38  
 add\_to\_storage() (stream\_framework.storage.base.BaseActivityStorage method), 39  
 add\_to\_storage() (stream\_framework.storage.memory.InMemoryActivityStorage method), 42  
 add\_to\_storage() (stream\_framework.storage.memory.InMemoryTimelineStorage method), 42  
 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), 40  
 AggregatedActivity (class in stream\_framework.activity), 29  
 AggregatedFeed (class in stream\_framework.feeds.aggregated\_feed.base), 37  
 aggregator\_class (stream\_framework.feeds.aggregated\_feed.base.AggregatedFeed attribute), 38  
 append() (stream\_framework.activity.AggregatedActivity method), 30

## B

BaseActivity (class in stream\_framework.activity), 31  
 BaseActivityStorage (class in stream\_framework.storage.base), 39  
 BaseAggregator (class in stream\_framework.aggregators.base), 32  
 BaseFeed (class in stream\_framework.feeds.base), 34

BaseStorage (class in stream\_framework.storage.base), 39  
 BaseTimelineStorage (class in stream\_framework.storage.base), 41  
 flush() (stream\_framework.storage.memory.InMemoryActivityStorage method), 42  
 Follow (class in stream\_framework.verbs.base), 44

## C

chunks() (in module stream\_framework.utils), 32  
 Comment (class in stream\_framework.verbs.base), 43  
 contains() (stream\_framework.activity.AggregatedActivity method), 30  
 contains() (stream\_framework.feeds.aggregated\_feed.base.AggregatedFeed method), 38  
 contains() (stream\_framework.storage.memory.InMemoryTimelineStorage method), 42  
 count() (stream\_framework.feeds.base.BaseFeed method), 35  
 count() (stream\_framework.storage.base.BaseTimelineStorage method), 41  
 count() (stream\_framework.storage.memory.InMemoryTimelineStorage method), 42

## D

datetime\_to\_epoch() (in module stream\_framework.utils), 32  
 default\_serializer\_class (stream\_framework.storage.base.BaseStorage attribute), 40  
 default\_serializer\_class (stream\_framework.storage.base.BaseTimelineStorage attribute), 41  
 DehydratedActivity (class in stream\_framework.activity), 31  
 delete() (stream\_framework.feeds.base.BaseFeed method), 35  
 delete() (stream\_framework.storage.base.BaseTimelineStorage method), 41  
 delete() (stream\_framework.storage.memory.InMemoryTimelineStorage method), 42  
 deserialize\_activities() (stream\_framework.storage.base.BaseStorage method), 40  
 DuplicateActivityException, 31

## E

epoch\_to\_datetime() (in module stream\_framework.utils), 32

## F

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  
 flush() (stream\_framework.feeds.base.BaseFeed class method), 35  
 flush() (stream\_framework.storage.base.BaseStorage method), 40

## G

get() (stream\_framework.storage.base.BaseActivityStorage method), 39  
 get() (stream\_framework.utils.LRUCache method), 32  
 get\_activity\_slice() (stream\_framework.feeds.base.BaseFeed method), 35  
 get\_aggregated\_feed() (stream\_framework.feeds.base.BaseFeed class method), 36  
 get\_aggregated\_feed() (stream\_framework.feeds.aggregated\_feed.base.AggregatedFeed method), 38  
 get\_batch\_interface() (stream\_framework.storage.base.BaseTimelineStorage method), 41  
 get\_batch\_interface() (stream\_framework.storage.memory.InMemoryTimelineStorage class method), 42  
 get\_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\_storage() (stream\_framework.storage.base.BaseActivityStorage method), 39  
 get\_timeline\_storage() (stream\_framework.storage.memory.InMemoryActivityStorage method), 42  
 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\_index\_of() (stream\_framework.activity.AggregatedActivity method), 30  
 get\_index\_of() (stream\_framework.activity.DehydratedActivity method), 31  
 get\_index\_of() (stream\_framework.storage.base.BaseTimelineStorage method), 41  
 get\_index\_of() (stream\_framework.storage.memory.InMemoryTimelineStorage method), 42  
 get\_many() (stream\_framework.storage.base.BaseActivityStorage method), 39  
 get\_metrics\_instance() (in module stream\_framework.utils), 32  
 get\_slice() (stream\_framework.storage.base.BaseTimelineStorage method), 41  
 get\_slice\_from\_storage() (stream\_framework.storage.base.BaseTimelineStorage method), 41  
 get\_slice\_from\_storage() (stream\_framework.storage.memory.InMemoryTimelineStorage method), 42

- get\_timeline\_batch\_interface()  
(stream\_framework.feeds.base.BaseFeed  
class method), 36
- 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\_verb\_by\_id() (in module stream\_framework.verbs),  
43
- get\_verb\_storage() (in module stream\_framework.verbs),  
43
- 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), 43
- id (stream\_framework.verbs.base.Comment attribute), 44
- id (stream\_framework.verbs.base.Follow attribute), 44
- id (stream\_framework.verbs.base.Love attribute), 44
- id (stream\_framework.verbs.base.Verb attribute), 44
- 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), 41
- infinitive (stream\_framework.verbs.base.Add attribute),  
43
- infinitive (stream\_framework.verbs.base.Comment  
attribute), 44
- infinitive (stream\_framework.verbs.base.Follow at-  
tribute), 44
- infinitive (stream\_framework.verbs.base.Love attribute),  
44
- InMemoryActivityStorage (class in  
stream\_framework.storage.memory), 42
- InMemoryTimelineStorage (class in  
stream\_framework.storage.memory), 42
- 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.base.BaseFeed at-  
tribute), 36
- key\_format (stream\_framework.feeds.base.UserBaseFeed  
attribute), 37
- ## L
- last\_activities (stream\_framework.activity.AggregatedActivity  
attribute), 30
- last\_activity (stream\_framework.activity.AggregatedActivity  
attribute), 30
- Love (class in stream\_framework.verbs.base), 44
- LRUCache (class in stream\_framework.utils), 32
- ## M
- make\_list\_unique() (in module stream\_framework.utils),  
32
- max\_aggregated\_activities\_length  
(stream\_framework.activity.AggregatedActivity  
attribute), 30
- max\_length (stream\_framework.feeds.base.BaseFeed at-  
tribute), 36
- max\_length (stream\_framework.feeds.base.UserBaseFeed  
attribute), 37
- 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 at-  
tribute), 40
- ## 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
- ## 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
- other\_actor\_count (stream\_framework.activity.AggregatedActivity  
attribute), 30

## P

past\_tense (stream\_framework.verbs.base.Add attribute), 43

past\_tense (stream\_framework.verbs.base.Comment attribute), 44

past\_tense (stream\_framework.verbs.base.Follow attribute), 44

past\_tense (stream\_framework.verbs.base.Love attribute), 44

## 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), 34

register() (in module stream\_framework.verbs), 43

remove() (stream\_framework.activity.AggregatedActivity method), 30

remove() (stream\_framework.feeds.base.BaseFeed method), 36

remove() (stream\_framework.storage.base.BaseActivityStorage method), 39

remove() (stream\_framework.storage.base.BaseTimelineStorage method), 41

remove\_activity() (stream\_framework.feeds.base.BaseFeed class method), 36

remove\_from\_storage() (stream\_framework.storage.base.BaseActivityStorage method), 39

remove\_from\_storage() (stream\_framework.storage.base.BaseTimelineStorage method), 42

remove\_from\_storage() (stream\_framework.storage.memory.InMemoryActivityStorage method), 42

remove\_from\_storage() (stream\_framework.storage.memory.InMemoryTimelineStorage method), 42

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), 39

remove\_many() (stream\_framework.storage.base.BaseTimelineStorage method), 42

remove\_many\_aggregated() (stream\_framework.feeds.aggregated\_feed.base.AggregatedFeed method), 38

reverse\_bisect() (in module stream\_framework.storage.memory), 42

## 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), 44

serialize\_activities() (stream\_framework.storage.base.BaseStorage method), 40

serialize\_activity() (stream\_framework.storage.base.BaseStorage method), 40

serializer (stream\_framework.storage.base.BaseStorage attribute), 40

set() (stream\_framework.utils.LRUCache method), 32

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), 37

stream\_framework.feeds.base (module), 34

stream\_framework.feeds.memory (module), 37

stream\_framework.settings (module), 31

stream\_framework.storage.base (module), 39

stream\_framework.storage.cassandra (module), 43

stream\_framework.storage.memory (module), 42

stream\_framework.utils (module), 32

stream\_framework.verbs (module), 43

stream\_framework.verbs.base (module), 43

stream\_framework.feeds.aggregated\_feed.base.AggregatedFeed (attribute), 38

stream\_framework.feeds.base.BaseFeed (attribute), 36

stream\_framework.feeds.aggregated\_feed.base.AggregatedFeed (attribute), 36

stream\_framework.feeds.memory.Feed (attribute), 37

trim() (stream\_framework.feeds.base.BaseFeed method), 36

try() (stream\_framework.storage.base.BaseTimelineStorage method), 42

try() (stream\_framework.storage.memory.InMemoryTimelineStorage method), 42

trim\_chance (stream\_framework.feeds.base.BaseFeed attribute), 37

## 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`), [37](#)

## V

`Verb` (class in `stream_framework.verbs.base`), [44](#)

`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](#)