

SimpleFeed

Table of Contents

1. Scalable, Easy to Use Activity Feed Implementation.....	1
1.1. Build & Gem Status.....	1
1.2. Test Coverage Map.....	1
2. Features.....	3
2.1. Publishing Events.....	3
2.2. Consuming Events (Reading / Rendering the Feed).....	4
2.3. Modifying User's Feed.....	4
2.4. Aggregating Events.....	4
3. Commercial & Enterprise Support.....	5
4. Usage.....	6
4.1. Example.....	6
4.2. Providers.....	6
4.3. Configuration.....	6
4.4. Reading from and writing to the feed.....	7
4.5. User IDs.....	8
4.5.1. Activity Keys.....	8
4.5.2. Partitioning Schema.....	8
4.5.3. Relationship between an Activity and a User.....	9
One to One.....	9
One to Many.....	9
4.6. The Two Forms of the Feed API.....	10
5. Complete API.....	15
6. Providers.....	17
6.1. SimpleFeed: : Providers: : Redis: : Provider.....	17
7. Running the Examples and Specs.....	18
7.1. Generating Ruby API Documentation.....	22
7.2. Installation.....	22
7.3. Development.....	22
7.4. Contributing.....	22
7.5. License.....	22
7.6. Acknowledgements.....	23

Chapter 1. Scalable, Easy to Use Activity Feed Implementation.



please feel free to read this README in the formatted-for-print [PDF Version](#).

1.1. Build & Gem Status

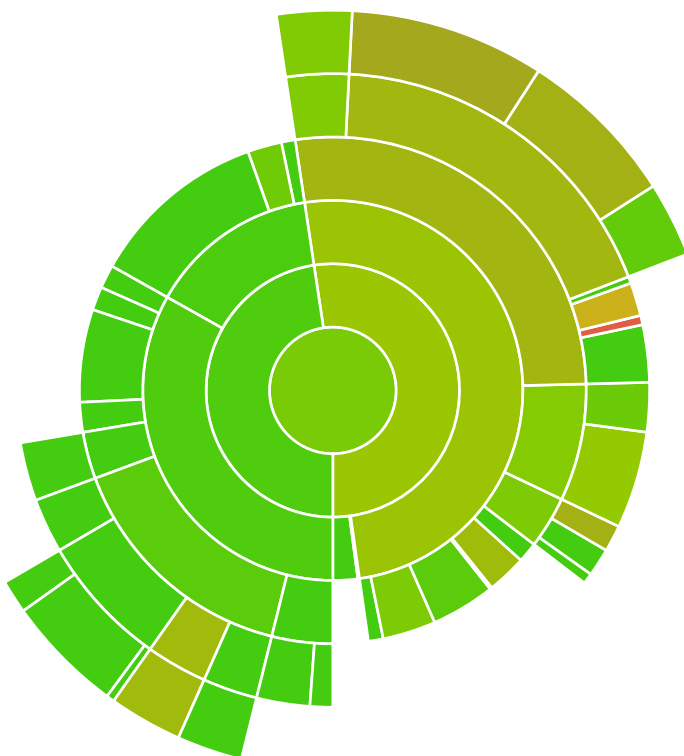
license MIT [License Scan OK]

[RSpec] [Rubocop]

gem v3.1.2 [Inline docs]

codecov 96%

1.2. Test Coverage Map



Please read the (somewhat outdated) blog post [Feeding Frenzy with SimpleFeed](#) launching this library. Please leave comments or questions in the discussion thread at the bottom of that post. Thanks!

If you like to see this project grow, your donation of any amount is much appreciated.

[\[Donate\]](#) | https://www.paypalobjects.com/en_US/i/btn/btn_donate_SM.gif

This is a fast, pure-ruby implementation of an activity feed concept commonly used in social

networking applications. The implementation is optimized for read-time performance and high concurrency (lots of users), and can be extended with custom backend providers. One data provider come bundled: the production-ready Redis provider.

Important Notes and Acknowledgements:

- ¥ SimpleFeed *does not depend on Ruby on Rails* and is a pure-ruby implementation
- ¥ SimpleFeed requires MRI Ruby 2.3 or later
- ¥ SimpleFeed is currently live in production
- ¥ SimpleFeed is open source thanks to the generosity of [Simbi, Inc.](#)

Chapter 2. Features

SimpleFeed is a Ruby Library that can be plugged into any application to power a fast, Redis-based activity feed implementation so common on social networking sites. SimpleFeed offers the following features:

- ¥ Modelled after graph-relationships similar to those on Twitter (bi-directional independent follow relationships):
 - ! Feed maintains a reverse-chronological order for heterogeneous events for each user.
 - ! It offers a constant time lookup for user's feed, avoiding complex SQL joins to render it.
 - ! An API to read/paginate the feed for a given user
 - ! As well as to query the total unread items in the feed since it was last read by the user (typically shown on App icons).
- ¥ Scalable and well performing Redis-based activity feed
 - ! Scales to millions of users (will need to use Twemproxy to shard across several Redis instances)
 - ! Stores a fixed number of events for each unique "user" ~ the default is 1000. When the feed reaches 1001 events, the oldest event is offloaded from the activity.
- ¥ Implementation properties:
 - ! Fully thread-safe implementation, writing events can be done in eg. Sidekiq.
 - ! Zero assumptions about what you are storing: the "data" is just a string. Serialize it with JSON, Marshall, YAML, or whatever.
 - ! You can create as many different types of feeds per application as you like (no Ruby Singletons used).
 - ! Customize mapping from `user_id` to the activity id based on your business logic (more on this later).

2.1. Publishing Events

Pushing events to the feed requires the following:

- ¥ An `Event` consisting of:
 - ! String `data` that, most commonly, is a foreign key to a database table, but can really be anything you like.
 - ! Float `at` (typically, the timestamp, but can be any `float` number)
- ¥ One or more user IDs, or event consumers: basically ~ who should see the event being published in their feed.

You publish an event by choosing a set of users whose feed should be updated. For example, were you re-implementing Twitter, your array of `user_ids` when publishing an event would be all followers of the Tweet's author. While the `data` would probably be the Tweet ID.



Publishing an event to the feeds of N users is roughly a $O(N * \log(N))$ operation

2.2. Consuming Events (Reading / Rendering the Feed)

You can fetch the chronologically ordered events for a particular user, using:

¥ Methods on the `activity` such as `paginate`, `fetch`.

! Reading feed for one user (or one type of user) is a $O(1)$ operation

¥ For each activity (user) you can fetch the `total_count` and the `unread_count` Ñ the number of total and new items in the feed, where `unread_count` is computed since the user last reset their `read status`.

! Note: `total_count` can never exceed the maximum size of the feed that you configured. The default is 1000 items.

! The `last_read` timestamp can be automatically reset when the user is shown the feed via `paginate` method (whether or not its reset is controlled via a method argument).

2.3. Modifying User's Feed

For any given user, you can:

¥ Wipe their feed with `wipe`

¥ Selectively remove items from the feed with `delete_if`.

! For instance, if a user un-follows someone they shouldn't see their events anymore, so you'd have to call `delete_if` and remove any events published by the unfollowed user.

2.4. Aggregating Events

This is a feature planned for future versions.

Help us much appreciated, even if you are not a developer, but have a clear idea about how it should work.

Chapter 3. Commercial & Enterprise Support

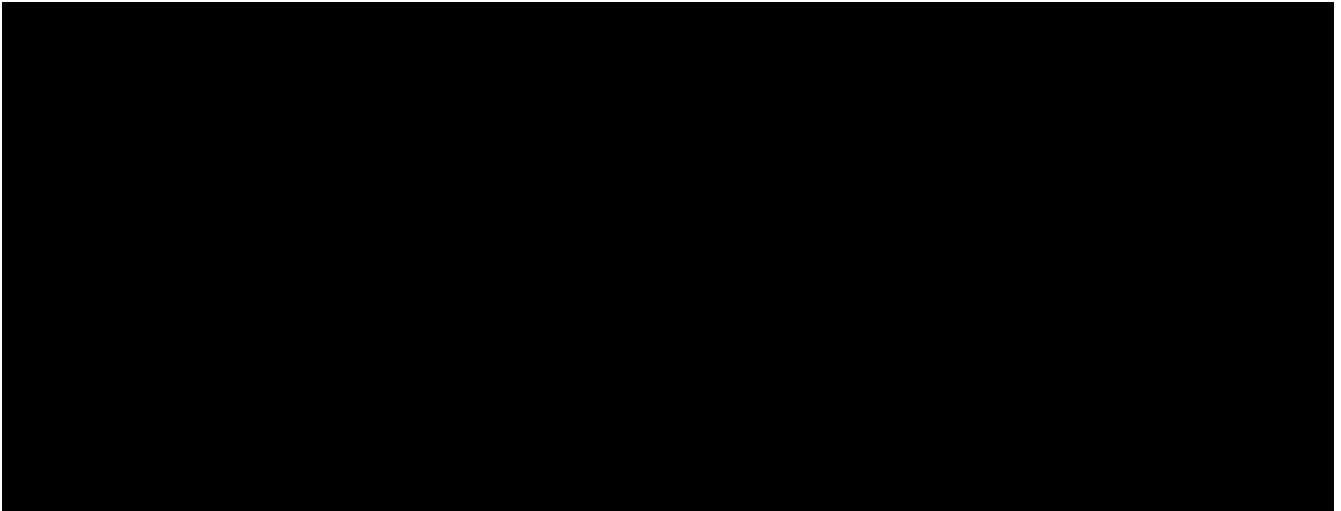
Commercial Support plans are available for SimpleFeed through author's [ReinventONE Inc](#) consulting company. Please reach out to kig AT reinvent.one for more information.

Chapter 4. Usage

4.1. Example

Please read the additional documentation, including the examples, on the [project's Github Wiki](#).

Below is a screen shot of an actual activity feed powered by this library.



4.2. Providers

A key concept to understanding SimpleFeed gem, is that of a *provider*, which is effectively a persistence implementation for the events belonging to each user.

One providers are supplied with this gem: the production-ready `:redis` provider, which uses the [sorted set Redis data type](#) to store and fetch the events, scored by time (but not necessarily).

You initialize a provider by using the `SimpleFeed.provider([Symbol])` method.

4.3. Configuration

Below we configure a feed called `:newsfeed`, which in this example will be populated with the various events coming from the followers.


```
require 'simplefeed'

# Let's define a Redis-based feed, and wrap Redis in a in a ConnectionPool.

SimpleFeed.define(:newsfeed) do |f|
  f.provider = SimpleFeed.provider(:redis,
    redis: -> { ::Redis.new },
    pool_size: 10)
  f.per_page = 50 # default page size
  f.batch_size = 10 # default batch size
  f.namespace = 'nf' # only needed if you use the same redis for more than one feed
end
```

After the feed is defined, the gem creates a similarly named method under the `SimpleFeed` namespace to access the feed. For example, given a name such as `:newsfeed` the following are all valid ways of accessing the feed:

```
SimpleFeed.newsfeed
SimpleFeed.get(:newsfeed)
```

You can also get a full list of currently defined feeds with `SimpleFeed.feed_names` method.

4.4. Reading from and writing to the feed

For the impatient, here is a quick way to get started with the `SimpleFeed`.

```
# Let's use the feed we defined earlier and create activity for all followers of the
current user
publish_activity = SimpleFeed.newsfeed.activity(@current_user.followers.map(&:id))

# Store directly the value and the optional time stamp
publish_activity.store(value: 'hello', at: Time.now)
# => true # indicates that value 'hello' was not yet in the feed (all events must be
unique)

# Or, using the event form:
publish_activity.store(event: SimpleFeed::Event.new('good bye', Time.now))
# => true
```

As we've added the two events for these users, we can now read them back, sorted by the time and paginated:

```
# Let's grab the first follower
user_activity = SimpleFeed.newsfeed.activity(@current_user.followers.first.id)

# Now we can paginate the events, while resetting this user's last-read timestamp:
user_activity.paginate(page: 1, reset_last_read: true)

# [
#   [0] #<SimpleFeed::Event: value=hello, at=1480475294.0579991>,
#   [1] #<SimpleFeed::Event: value=good bye, at=1480472342.8979871>,
# ]
```

!!

Note that we stored the activity by passing an array of users, but read the activity for just one user. This is how you'll use SimpleFeed most of the time, with the exception of the alternative mapping described below.

4.5. User IDs

In the previous section you saw the examples of publishing events to many feeds, and then reading the activity for a given user.

SimpleFeed supports user IDs that are either numeric (integer) or string-based (eg, UUID). Numeric IDs are best for simplest cases, and are the most compact. String keys offer the most flexibility.

4.5.1. Activity Keys

In the next section we'll talk about generating **keys** from **user_ids**. We mean Redis Hash keys that uniquely map a user (or a set of users) to the activity feed they should see.

There are up to two keys that are computed depending on the situation:

¥ **data_key** is used to store the actual feed events

¥ **meta_key** is used to store user's **last_read** status

4.5.2. Partitioning Schema

!

This feature is only available in SimpleFeed Version 3+.

You can take advantage of string user IDs for situations where your feed requires keys to be composite for instance. Just remember that SimpleFeed does not care about what's in your user ID, and even what you call "a user". It's convenient to think of the activities in terms of users, because typically each user has a unique feed that only they see.

But you can just as easily use zip code as the unique activity ID, and create one feed of events per geographical location, that all folks living in that zip code share. But what about other countries?

Now you use partitioning scheme: make the "user_id" argument a combination **iso_country_code.postal_code**, eg for San Francisco, you'll use **us.94107**, but for Australia you could use, eg **au.3148**.

4.5.3. Relationship between an Activity and a User

One to One

In the most common case, you will have one activity per user.

For instance, in the Twitter example, each Twitter user has a unique tweeter feed that only they see.

The events are published when someone posts a tweet, to the array of all users that follow the Tweet author.

One to Many

However, SimpleFeed supports one additional use-case, where you might have one activity shared among many users.

Imagine a service that notifies residents of important announcements based on user's zip code of residence.

We want this feed to work as follows:

- ¥ All users that share a zip-code should see the same exact feed.
- ¥ However, all users should never share the individual's `last_read` status: so if two people read the same activity from the same zip code, their `unread_count` should change independently.

In terms of the activity keys, this means:

- ¥ `data_key` should be based on the zip-code of each user, and be one to many with users.
- ¥ `meta_key` should be based on the user ID as we want it to be 1-1 with users.

To support this use-case, SimpleFeed supports two optional transformer lambdas that can be applied to each user object when computing their activity feed hash key:

```
SimpleFeed.define(:zipcode_alerts) do |f|
  f.provider = SimpleFeed.provider(:redis, redis: -> { ::Redis.new }, pool_size: 10)
  f.namespace = 'zc'
  f.data_key_transformer = ->(user) { user.zip_code } # actual feed data is stored
  # once per zip code
  f.meta_key_transformer = ->(user) { user.id } # last_read status is stored
  # once per user
end
```

When you publish events into this feed, you would need to provide `User` objects that all respond to `.zip_code` method (based on the above configuration). Since the data is only defined by Zip Code, you probably don't want to be publishing it via a giant array of users. Most likely, you'll want to publish event based on the zip code, and consume them based on the user ID.

To support this user-case, let's modify our transformer lambda (only the `data` one) as follows ~ so

that it can support both the consuming read by a user case, and the publishing a feed by zip code case:

Alternatively, you could do something like this:

```
def data_key_transformer = ->(entity) do
  case entity
  when User
    entity.zip_code.to_i
  when String # UUIDs
    User.find(entity)&.zip_code.to_i
  when ZipCode, Numeric
    entity.to_i
  else
    raise ArgumentError, "Invalid type #{entity.class.name}"
  end
end
```

Just make sure that your users always have `.zip_code` defined, and that `ZipCode.new(94107).to_i` returns exactly the same thing as `@user.zip_code.to_i` or your users won't see the feeds they are supposed to see.

4.6. The Two Forms of the Feed API

The feed API is offered in two forms:

1. single-user form, and
2. a batch (multi-user) form.

The method names and signatures are the same. The only difference is in what the methods return:

1. In the single user case, the return of, say, `#total_count` is an `Integer` value representing the total count for this user.
2. In the multi-user case, the return is a `SimpleFeed::Response` instance, that can be thought of as a `Hash`, that has the user IDs as the keys, and return results for each user as a value.

Please see further below the details about the [Batch API](#).

Single-User API

In the examples below we show responses based on a single-user usage. As previously mentioned, the multi-user usage is the same, except what the response values are, and is discussed further down below.

Let's take a look at a ruby session, which demonstrates return values of the feed operations for a single user:

```

require 'simplefeed'

# Define the feed using Redis provider, which uses
# SortedSet to keep user's events sorted.
SimpleFeed.define(:followers) do |f|
  f.provider = SimpleFeed.provider(:redis)
  f.per_page = 50
  f.per_page = 2
end

# Let's get the Activity instance that wraps this
activity = SimpleFeed.followers.activity(user_id)      # => [... complex object
removed for brevity ]

# Let's clear out this feed to ensure it's empty
activity.wipe                                          # => true

# Let's verify that the counts for this feed are at zero
activity.total_count                                # => 0
activity.unread_count                                # => 0

# Store some events
activity.store(value: 'hello')                        # => true
activity.store(value: 'goodbye', at: Time.now - 20)   # => true
activity.unread_count                                # => 2

# Now we can paginate the events, while resetting this user's last-read timestamp:
activity.paginate(page: 1, reset_last_read: true)
# [
#   [0] #<SimpleFeed::Event: value=good bye, at=1480475294.0579991>,
#   [1] #<SimpleFeed::Event: value=hello, at=1480475294.057138>
# ]
# Now the unread_count should return 0 since the user just "viewed" the feed.
activity.unread_count                                # => 0
activity.delete(value: 'hello')                       # => true
# the next method yields to a passed in block for each event in the user's feed, and
# deletes
# all events for which the block returns true. The return of this call is the
# array of all events that have been deleted for this user.
activity.delete_if do |event, user_id|
  event.value =~ /good/
end
# => [
#   [0] #<SimpleFeed::Event: value=good bye, at=1480475294.0579991>
# ]
activity.total_count                                # => 0

```

You can fetch all items (optionally filtered by time) in the feed using `#fetch`, `#paginate` and reset the `last_read` timestamp by passing the `reset_last_read: true` as a parameter.

Batch (Multi-User) API

This API should be used when dealing with an array of users (or, in the future, a Proc or an ActiveRecord relation).

There are several reasons why this API should be preferred for operations that perform a similar action across a range of users: *various provider implementations can be heavily optimized for concurrency, and performance.*

The Redis Provider, for example, uses a notion of **pipelining** to send updates for different users asynchronously and concurrently.

Multi-user operations return a **SimpleFeed::Response** object, which can be used as a hash (keyed on user_id) to fetch the result of a given user.

```
# Using the Feed API with, eg #find_in_batches
@event_producer.followers.find_in_batches do |group|

  # Convert a group to the array of IDs and get ready to store
  activity = SimpleFeed.get(:followers).activity(group.map(&:id))
  activity.store(value: "#{@event_producer.name} liked an article")

  # => [Response] { user_id1 => [Boolean], user_id2 => [Boolean]... }
  # true if the value was stored, false if it wasn't.
end
```

Activity Feed DSL (Domain-Specific Language)

The library offers a convenient DSL for adding feed functionality into your current scope.

To use the module, just include **SimpleFeed::DSL** where needed, which exports just one primary method **with_activity**. You call this method and pass an activity object created for a set of users (or a single user), like so:

```

require 'simplefeed/dsl'
include SimpleFeed::DSL

feed = SimpleFeed.newfeed
activity = feed.activity(current_user.id)
data_to_store = %w(France Germany England)

def report(value)
  puts value
end

with_activity(activity, countries: data_to_store) do
  # we can use countries as a variable because it was passed above in **opts
  countries.each do |country|
    # we can call #store without a receiver because the block is passed to
    # instance_eval
    store(value: country) { |result| report(result ? 'success' : 'failure') }
    # we can call #report inside the proc because it is evaluated in the
    # outside context of the #with_activity

    # now let's print a color ASCII dump of the entire feed for this user:
    color_dump
  end
end
printf "Activity counts are: %d unread of %d total\n", unread_count, total_count
end

```

The DSL context has access to two additional methods:

¥ `#event(value, at)` returns a fully constructed `SimpleFeed::Event` instance

¥ `#color_dump` prints to STDOUT the ASCII text dump of the current user's activities (events), as well as the counts and the `last_read` shown visually on the time line.

`#color_dump`

Below is an example output of `color_dump` method, which is intended for the debugging purposes.

Figure 1. #color_dump method output

Chapter 5. Complete API

For completeness sake we'll show the multi-user API responses only. For a single-user use-case the response is typically a scalar, and the input is a singular `user_id`, not an array of ids.

Multi-User (Batch) API

Each API call at this level expects an array of user IDs, therefore the return value is an object, `SimpleFeed::Response`, containing individual responses for each user, accessible via `response[user_id]` method.

```
@multi = SimpleFeed.get(:feed_name).activity(User.active.map(&:id))

@multi.store(value:, at:)
@multi.store(event:)
# => [Response] { user_id => [Boolean], ... } true if the value was stored, false if
it wasn't.

@multi.delete(value:, at:)
@multi.delete(event:)
# => [Response] { user_id => [Boolean], ... } true if the value was removed, false if
it didn't exist

@multi.delete_if do |event, user_id|
  # if the block returns true, the event is deleted and returned
end
# => [Response] { user_id => [deleted_event1, deleted_event2, ...], ... }

# Wipe the feed for a given user(s)
@multi.wipe
# => [Response] { user_id => [Boolean], ... } true if user activity was found and
deleted, false otherwise

# Return a paginated list of all items, optionally with the total count of items
@multi.paginate(page: 1,
  # per_page: @multi.feed.per_page,
  # with_total: false,
  # reset_last_read: false)
# => [Response] { user_id => [Array]<Event>, ... }
# Options:
#   reset_last_read: false Ñ reset last read to Time.now (true), or the provided
timestamp
#   with_total: true Ñ returns a hash for each user_id:
#   => [Response] { user_id => { events: Array<Event>, total_count: 3 }, ... }

# Return un-paginated list of all items, optionally filtered
@multi.fetch(since: nil, reset_last_read: false)
# => [Response] { user_id => [Array]<Event>, ... }
# Options:
```

```
#   reset_last_read: false N reset last read to Time.now (true), or the provided
#   timestamp
#   since: <timestamp> N if provided, returns all items posted since then
#   since: :last_read N if provided, returns all unread items and resets +last_read+

@multi.reset_last_read
# => [Response] { user_id => [Time] last_read, ... }

@multi.total_count
# => [Response] { user_id => [Integer, String] total_count, ... }

@multi.unread_count
# => [Response] { user_id => [Integer, String] unread_count, ... }

@multi.last_read
# => [Response] { user_id => [Time] last_read, ... }
```

Chapter 6. Providers

As we’ve discussed above, a provider is an underlying persistence mechanism implementation.

It is the intention of this gem that:

- ¥ it should be easy to write new providers
- ¥ it should be easy to swap out providers

One provider is included with this gem:

6.1. SimpleFeed::Providers::Redis::Provider

Redis Provider is a production-ready persistence adapter that uses the [sorted set Redis data type](#).

This provider is optimized for large writes and can use either a single Redis instance for all users of your application, or any number of Redis [shards](#) by using a [Twemproxy](#) in front of the Redis shards.

If you set environment variable `REDIS_DEBUG` to `true` and run the example (see below) you will see every operation redis performs. This could be useful in debugging an issue or submitting a bug report.

Chapter 7. Running the Examples and Specs

Source code for the gem contains the `examples` folder with an example file that can be used to test out the providers, and see what they do under the hood.

Both the specs and the example requires a local redis instance to be available.

To run it, checkout the source of the library, and then:

```
git clone https://github.com/kigster/simple-feed.git
cd simple-feed

# on OSX with HomeBrew:
brew install redis
brew services start redis

# check that your redis is up:
redis-cli info

# install bundler and other dependencies
gem install bundler --version 2.1.4
bundle install
bundle exec rspec # make sure tests are passing

# run the example:
ruby examples/redis_provider_example.rb
```

The above command will help you download, setup all dependencies, and run the examples for a single user:

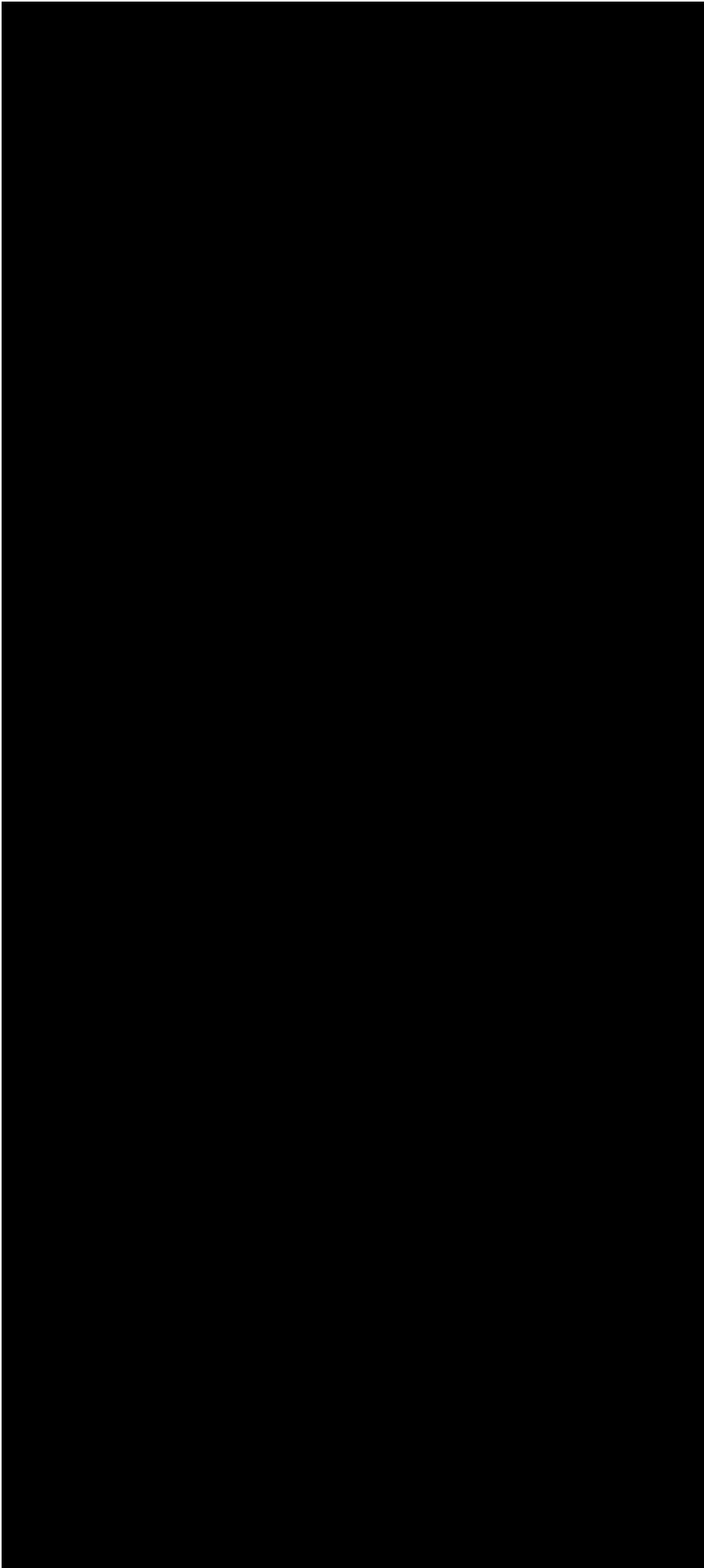


Figure 2. Running Redis Example in a Terminal

If you set `REDIS_DEBUG` variable prior to running the example, you will be able to see every single Redis command executed as the example works its way through. Below is a sample output:

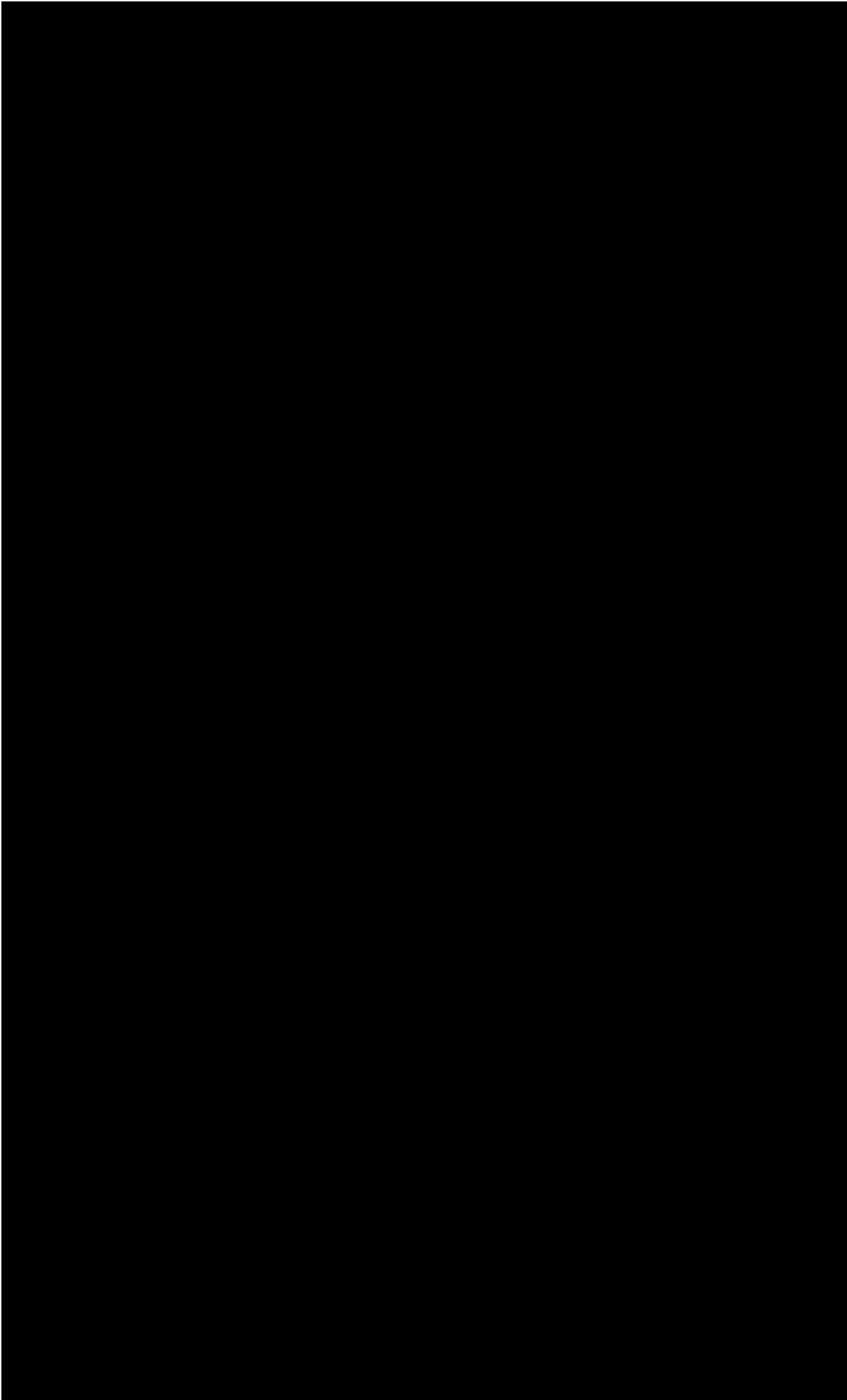


Figure 3. Running Redis Example with REDIS_DEBUG set

7.1. Generating Ruby API Documentation

```
rake doc
```

This should use YARD to generate the documentation, and open your browser once it's finished.

7.2. Installation

Add this line to your application's Gemfile:

```
gem 'simple-feed'
```

And then execute:

```
$ bundle
```

Or install it yourself as:

```
$ gem install simple-feed
```

7.3. Development

After checking out the repo, run `bin/setup` to install dependencies. Then, run `rake spec` to run the tests. You can also run `bin/console` for an interactive prompt that will allow you to experiment.

To install this gem onto your local machine, run `bundle exec rake install`. To release a new version, update the version number in `version.rb`, and then run `bundle exec rake release`, which will create a git tag for the version, push git commits and tags, and push the `.gem` file to rubygems.org.

7.4. Contributing

Bug reports and pull requests are welcome on GitHub at <https://github.com/kigster/simple-feed>

7.5. License

The gem is available as open source under the terms of the [MIT License](#).

[FOSSA Scan Status]

7.6. Acknowledgements

¥ This project is conceived and sponsored by [Simbi, Inc.](#).

¥ Author's personal experience at [Wanelo, Inc.](#) has served as an inspiration.