
Acts As Votable (aka Acts As Likeable)



Acts As Votable is a Ruby Gem specifically written for Rails/ActiveRecord models. The main goals of this gem are:

- Allow any model to be voted on, like/dislike, upvote/downvote, etc.
- Allow any model to be voted under arbitrary scopes.
- Allow any model to vote. In other words, votes do not have to come from a user, they can come from any model (such as a Group or Team).
- Provide an easy to write/read syntax.

Installation

Supported Ruby and Rails versions

- Ruby $\geq 2.5.0$
- Rails ≥ 5.1

Install

Just add the following to your Gemfile to install the latest release.

```
1 gem 'acts_as_votable'
```

And follow that up with a `bundle install`.

Database Migrations

Acts As Votable uses a votes table to store all voting information. To generate and run the migration just use.

```
1 rails generate acts_as_votable:migration
2 rails db:migrate
```

You will get a performance increase by adding in cached columns to your model's tables. You will have to do this manually through your own migrations. See the caching section of this document for more information.

Usage

Votable Models

```
1 class Post < ApplicationRecord
2   acts_as_votable
3 end
4
5 @post = Post.new(name: 'my post!')
6 @post.save
7
8 @post.liked_by @user
9 @post.votes_for.size # => 1
```

Like/Dislike Yes/No Up/Down

Here are some voting examples. All of these calls are valid and acceptable. The more natural calls are the first few examples.

```
1 @post.liked_by @user1
2 @post.downvote_from @user2
3 @post.vote_by voter: @user3
4 @post.vote_by voter: @user4, vote: 'bad'
5 @post.vote_by voter: @user5, vote: 'like'
```

By default all votes are positive, so `@user3` has cast a ‘good’ vote for `@post`.

`@user1`, `@user3`, and `@user5` all voted in favor of `@post`.

`@user2` and `@user4` voted against `@post`.

Just about any word works for casting a vote in favor or against post. Up/Down, Like/Dislike, Positive/Negative... the list goes on-and-on. Boolean flags **true** and **false** are also applicable.

Revisiting the previous example of code.

```
1 # positive votes
2 @post.liked_by @user1
3 @post.vote_by voter: @user3
4 @post.vote_by voter: @user5, vote: 'like'
5
6 # negative votes
7 @post.downvote_from @user2
8 @post.vote_by voter: @user2, vote: 'bad'
9
10 # tally them up!
11 @post.votes_for.size # => 5
12 @post.weighted_total # => 5
```

```
13 @post.get_likes.size # => 3
14 @post.get_upvotes.size # => 3
15 @post.get_dislikes.size # => 2
16 @post.get_downvotes.size # => 2
17 @post.weighted_score # => 1
```

Active Record scopes are provided to make life easier.

```
1 @post.votes_for.up.by_type(User)
2 @post.votes_for.down
3 @user1.votes.up
4 @user1.votes.down
5 @user1.votes.up.for_type(Post)
```

Once scoping is complete, you can also trigger a get for the voter/votable

```
1 @post.votes_for.up.by_type(User).voters
2 @post.votes_for.down.by_type(User).voters
3
4 @user.votes.up.for_type(Post).votables
5 @user.votes.up.votables
```

You can also ‘unvote’ a model to remove a previous vote.

```
1 @post.liked_by @user1
2 @post.unliked_by @user1
3
4 @post.disliked_by @user1
5 @post.undisliked_by @user1
```

Unvoting works for both positive and negative votes.

Examples with scopes

You can add a scope to your vote

```
1 # positive votes
2 @post.liked_by @user1, vote_scope: 'rank'
3 @post.vote_by voter: @user3, vote_scope: 'rank'
4 @post.vote_by voter: @user5, vote: 'like', vote_scope: 'rank'
5
6 # negative votes
7 @post.downvote_from @user2, vote_scope: 'rank'
8 @post.vote_by voter: @user2, vote: 'bad', vote_scope: 'rank'
9
10 # tally them up!
11 @post.find_votes_for(vote_scope: 'rank').size # => 5
12 @post.get_likes(vote_scope: 'rank').size # => 3
13 @post.get_upvotes(vote_scope: 'rank').size # => 3
```

```
14 @post.get_dislikes(vote_scope: 'rank').size # => 2
15 @post.get_downvotes(vote_scope: 'rank').size # => 2
16
17 # votable model can be voted under different scopes
18 # by the same user
19 @post.vote_by voter: @user1, vote_scope: 'week'
20 @post.vote_by voter: @user1, vote_scope: 'month'
21
22 @post.votes_for.size # => 2
23 @post.find_votes_for(vote_scope: 'week').size # => 1
24 @post.find_votes_for(vote_scope: 'month').size # => 1
```

Adding weights to your votes

You can add weight to your vote. The default value is 1.

```
1 # positive votes
2 @post.liked_by @user1, vote_weight: 1
3 @post.vote_by voter: @user3, vote_weight: 2
4 @post.vote_by voter: @user5, vote: 'like', vote_scope: 'rank',
   vote_weight: 3
5
6 # negative votes
7 @post.downvote_from @user2, vote_scope: 'rank', vote_weight: 1
8 @post.vote_by voter: @user2, vote: 'bad', vote_scope: 'rank',
   vote_weight: 3
9
10 # tally them up!
11 @post.find_votes_for(vote_scope: 'rank').sum(:vote_weight) # => 6
12 @post.get_likes(vote_scope: 'rank').sum(:vote_weight) # => 6
13 @post.get_upvotes(vote_scope: 'rank').sum(:vote_weight) # => 6
14 @post.get_dislikes(vote_scope: 'rank').sum(:vote_weight) # => 4
15 @post.get_downvotes(vote_scope: 'rank').sum(:vote_weight) # => 4
```

The Voter

You can have your voters `acts_as_voter` to provide some reserve functionality.

```
1 class User < ApplicationRecord
2   acts_as_voter
3 end
4
5 @user.likes @article
6
7 @article.votes_for.size # => 1
8 @article.get_likes.size # => 1
9 @article.get_dislikes.size # => 0
```

To check if a voter has voted on a model, you can use `voted_for?`. You can check how the voter voted by using `voted_as_when_voted_for`.

```
1 @user.likes @comment1
2 @user.up_votes @comment2
3 # user has not voted on @comment3
4
5 @user.voted_for? @comment1 # => true
6 @user.voted_for? @comment2 # => true
7 @user.voted_for? @comment3 # => false
8
9 @user.voted_as_when_voted_for @comment1 # => true, user liked it
10 @user.voted_as_when_voted_for @comment2 # => false, user didnt like it
11 @user.voted_as_when_voted_for @comment3 # => nil, user has yet to vote
```

You can also check whether the voter has voted up or down.

```
1 @user.likes @comment1
2 @user.dislikes @comment2
3 # user has not voted on @comment3
4
5 @user.voted_up_on? @comment1 # => true
6 @user.voted_down_on? @comment1 # => false
7
8 @user.voted_down_on? @comment2 # => true
9 @user.voted_up_on? @comment2 # => false
10
11 @user.voted_up_on? @comment3 # => false
12 @user.voted_down_on? @comment3 # => false
```

Aliases for methods `voted_up_on?` and `voted_down_on?` are: `voted_up_for?`, `voted_down_for?`, `liked?` and `disliked?`.

Also, you can obtain a list of all the objects a user has voted for. This returns the actual objects instead of instances of the Vote model. All objects are eager loaded

```
1 @user.find_voted_items
2
3 @user.find_up_voted_items
4 @user.find_liked_items
5
6 @user.find_down_voted_items
7 @user.find_disliked_items
```

Members of an individual model that a user has voted for can also be displayed. The result is an ActiveRecord Relation.

```
1 @user.get_voted Comment
2
3 @user.get_up_voted Comment
```

```
4
5 @user.get_down_voted Comment
```

Registered Votes

Voters can only vote once per model. In this example the 2nd vote does not count because `@user` has already voted for `@shoe`.

```
1 @user.likes @shoe
2 @user.likes @shoe
3
4 @shoe.votes_for.size # => 1
5 @shoe.get_likes.size # => 1
```

To check if a vote counted, or registered, use `vote_registered?` on your model after voting. For example:

```
1 @hat.liked_by @user
2 @hat.vote_registered? # => true
3
4 @hat.liked_by => @user
5 @hat.vote_registered? # => false, because @user has already voted this
   way
6
7 @hat.disliked_by @user
8 @hat.vote_registered? # => true, because user changed their vote
9
10 @hat.votes_for.size # => 1
11 @hat.get_positives.size # => 0
12 @hat.get_negatives.size # => 1
```

To permit duplicates entries of a same voter, use option `duplicate`. Also notice that this will limit some other methods that didn't deal with multiples votes, in this case, the last vote will be considered.

```
1 @hat.vote_by voter: @user, duplicate: true
```

Caching

To speed up perform you can add cache columns to your votable model's table. These columns will automatically be updated after each vote. For example, if we wanted to speed up `@post` we would use the following migration:

```
1 class AddCachedVotesToPosts < ActiveRecord::Migration
2   def change
3     change_table :posts do |t|
```

```

4      t.integer :cached_votes_total, default: 0
5      t.integer :cached_votes_score, default: 0
6      t.integer :cached_votes_up, default: 0
7      t.integer :cached_votes_down, default: 0
8      t.integer :cached_weighted_score, default: 0
9      t.integer :cached_weighted_total, default: 0
10     t.float :cached_weighted_average, default: 0.0
11   end
12
13   # Uncomment this line to force caching of existing votes
14   # Post.find_each(&:update_cached_votes)
15 end
16 end

```

If you have a scope for your vote, let's say `subscribe`, your migration will be slightly different like below:

```

1  class AddCachedVotesToPosts < ActiveRecord::Migration
2    def change
3      change_table :posts do |t|
4        t.integer :cached_scoped_subscribe_votes_total, default: 0
5        t.integer :cached_scoped_subscribe_votes_score, default: 0
6        t.integer :cached_scoped_subscribe_votes_up, default: 0
7        t.integer :cached_scoped_subscribe_votes_down, default: 0
8        t.integer :cached_weighted_subscribe_score, default: 0
9        t.integer :cached_weighted_subscribe_total, default: 0
10       t.float :cached_weighted_subscribe_average, default: 0.0
11     end
12     # Uncomment this line to force caching of existing scoped votes
13     # Post.find_each { |p| p.update_cached_votes("subscribe") }
14   end
15 end
16 end

```

`cached_weighted_average` can be helpful for a rating system, e.g.:

Order by average rating:

```
1 Post.order(cached_weighted_average: :desc)
```

Display average rating:

```

1 <%= post.weighted_average.round(2) %> / 5
2 <!-- 3.5 / 5 -->

```

Votable model's updated_at

You can control whether `updated_at` column of votable model will be touched or not by passing `cacheable_strategy` option to `acts_as_votable` method.

By default, `update` strategy is used. Pass `:update_columns` as `cacheable_strategy` if you don't want to touch model's `updated_at` column.

```
1 class Post < ApplicationRecord
2   acts_as_votable cacheable_strategy: :update_columns
3 end
```

Testing

All tests follow the RSpec format and are located in the spec directory. They can be run with:

```
1 rake spec
```

License

Acts as votable is released under the MIT License.