

---

## ActsAsParanoid

build unknown

A Rails plugin to add soft delete.

This gem can be used to hide records instead of deleting them, making them recoverable later.

### Support

#### This version targets Rails 6.1+ and Ruby 2.7+ only

If you're working with Rails 6.0 and earlier, or with Ruby 2.6 or earlier, please require an older version of the `acts_as_paranoid` gem.

### Known issues

- Using `acts_as_paranoid` and `ActiveStorage` on the same model leads to a `SystemStackError`.
- You cannot directly create a model in a deleted state, or update a model after it's been deleted.

### Usage

#### Install gem:

```
1 gem 'acts_as_paranoid'
```

```
1 bundle install
```

#### Create migration

```
1 bin/rails generate migration AddDeletedAtToParanoiac deleted_at:
  datetime:index
```

#### Enable ActsAsParanoid

```
1 class Paranoiac < ActiveRecord::Base
2   acts_as_paranoid
3 end
```

By default, `ActsAsParanoid` assumes a record's *deletion* is stored in a `datetime` column called `deleted_at`.

---

## Options

If you are using a different column name and type to store a record's *deletion*, you can specify them as follows:

- `column: 'deleted'`
- `column_type: 'boolean'`

While *column* can be anything (as long as it exists in your database), *type* is restricted to:

- `boolean`
- `time` or
- `string`

Note that the `time` type corresponds to the database column type `datetime` in your Rails migrations and schema.

If your column type is a `string`, you can also specify which value to use when marking an object as deleted by passing `:deleted_value` (default is “deleted”). Any records with a non-matching value in this column will be treated normally, i.e., as not deleted.

If your column type is a `boolean`, it is possible to specify `allow_nulls` option which is `true` by default. When set to `false`, entities that have `false` value in this column will be considered not deleted, and those which have `true` will be considered deleted. When `true` everything that has a not-null value will be considered deleted.

## Filtering

If a record is deleted by `ActsAsParanoid`, it won't be retrieved when accessing the database.

So, `Paranoiaiac.all` will **not** include the **deleted records**.

When you want to access them, you have 2 choices:

```
1 Paranoiaiac.only_deleted # retrieves only the deleted records
2 Paranoiaiac.with_deleted # retrieves all records, deleted or not
```

When using the default `column_type` of `'time'`, the following extra scopes are provided:

```
1 time = Time.now
2
3 Paranoiaiac.deleted_after_time(time)
4 Paranoiaiac.deleted_before_time(time)
5
6 # Or roll it all up and get a nice window:
7 Paranoiaiac.deleted_inside_time_window(time, 2.minutes)
```

---

## Real deletion

In order to really delete a record, just use:

```
1 paranoia.destroy_fully!  
2 Paranoiac.delete_all!(conditions)
```

**NOTE:** The `.destroy!` method is still usable, but equivalent to `.destroy`. It just hides the object.

Alternatively you can permanently delete a record by calling `destroy` or `delete_all` on the object **twice**.

If a record was already deleted (hidden by `ActsAsParanoid`) and you delete it again, it will be removed from the database.

Take this example:

```
1 p = Paranoiac.first  
2  
3 # does NOT delete the first record, just hides it  
4 p.destroy  
5  
6 # deletes the first record from the database  
7 Paranoiac.only_deleted.where(id: p.id).first.destroy
```

This behaviour can be disabled by setting the configuration option. In a future version, **false** will be the default setting.

- `double_tap_destroys_fully: false`

## Recovery

Recovery is easy. Just invoke `recover` on it, like this:

```
1 Paranoiac.only_deleted.where("name = ?", "not dead yet").first.recover
```

All associations marked as `dependent: :destroy` are also recursively recovered.

If you would like to disable this behavior, you can call `recover` with the `recursive` option:

```
1 Paranoiac.only_deleted.where("name = ?", "not dead yet").first.recover(  
  recursive: false)
```

If you would like to change this default behavior for one model, you can use the `recover_dependent_associations` option

---

```
1 class Paranoiac < ActiveRecord::Base
2   acts_as_paranoiac recover_dependent_associations: false
3 end
```

By default, dependent records will be recovered if they were deleted within 2 minutes of the object upon which they depend.

This restores the objects to the state before the recursive deletion without restoring other objects that were deleted earlier.

The behavior is only available when both parent and dependant are using timestamp fields to mark deletion, which is the default behavior.

This window can be changed with the `dependent_recovery_window` option:

```
1 class Paranoiac < ActiveRecord::Base
2   acts_as_paranoiac
3   has_many :paranoics, dependent: :destroy
4 end
5
6 class Paranoic < ActiveRecord::Base
7   belongs_to :paranoiac
8
9   # Paranoic objects will be recovered alongside Paranoic objects
10  # if they were deleted within 10 minutes of the Paranoic object
11  acts_as_paranoic dependent_recovery_window: 10.minutes
12 end
```

or in the recover statement

```
1 Paranoiac.only_deleted.where("name = ?", "not dead yet").first
2   .recover(recovery_window: 30.seconds)
```

## recover!

You can invoke `recover!` if you wish to raise an error if the recovery fails. The error generally stems from ActiveRecord.

```
1 Paranoiac.only_deleted.where("name = ?", "not dead yet").first.recover!
2 # => ActiveRecord::RecordInvalid: Validation failed: Name already
    exists
```

Optionally, you may also raise the error by passing `raise_error: true` to the `recover` method. This behaves the same as `recover!`.

```
1 Paranoiac.only_deleted.where("name = ?", "not dead yet").first.recover(
  raise_error: true)
```

---

## Validation

ActiveRecord's built-in uniqueness validation does not account for records deleted by ActsAsParanoid. If you want to check for uniqueness among non-deleted records only, use the macro `validates_as_paranoid` in your model. Then, instead of using `validates_uniqueness_of`, use `validates_uniqueness_of_without_deleted`. This will keep deleted records from counting against the uniqueness check.

```
1 class Paranoiac < ActiveRecord::Base
2   acts_as_paranoid
3   validates_as_paranoid
4   validates_uniqueness_of_without_deleted :name
5 end
6
7 p1 = Paranoiac.create(name: 'foo')
8 p1.destroy
9
10 p2 = Paranoiac.new(name: 'foo')
11 p2.valid? #=> true
12 p2.save
13
14 p1.recover #=> fails validation!
```

## Status

A paranoid object could be deleted or destroyed fully.

You can check if the object is deleted with the `deleted?` helper

```
1 Paranoiac.create(name: 'foo').destroy
2 Paranoiac.with_deleted.first.deleted? #=> true
```

After the first call to `.destroy` the object is `deleted?`.

You can check if the object is fully destroyed with `destroyed_fully?` or `deleted_fully?`.

```
1 Paranoiac.create(name: 'foo').destroy
2 Paranoiac.with_deleted.first.deleted? #=> true
3 Paranoiac.with_deleted.first.destroyed_fully? #=> false
4 p1 = Paranoiac.with_deleted.first
5 p1.destroy # this fully destroys the object
6 p1.destroyed_fully? #=> true
7 p1.deleted_fully? #=> true
```

---

## Scopes

As you've probably guessed, `with_deleted` and `only_deleted` are scopes. You can, however, chain them freely with other scopes you might have.

For example:

```
1 Paranoiac.pretty.with_deleted
```

This is exactly the same as:

```
1 Paranoiac.with_deleted.pretty
```

You can work freely with scopes and it will just work:

```
1 class Paranoiac < ActiveRecord::Base
2   acts_as_paranoid
3   scope :pretty, where(pretty: true)
4 end
5
6 Paranoiac.create(pretty: true)
7
8 Paranoiac.pretty.count ==> 1
9 Paranoiac.only_deleted.count ==> 0
10 Paranoiac.pretty.only_deleted.count ==> 0
11
12 Paranoiac.first.destroy
13
14 Paranoiac.pretty.count ==> 0
15 Paranoiac.only_deleted.count ==> 1
16 Paranoiac.pretty.only_deleted.count ==> 1
```

## Associations

Associations are also supported.

From the simplest behaviors you'd expect to more nifty things like the ones mentioned previously or the usage of the `:with_deleted` option with `belongs_to`

```
1 class Parent < ActiveRecord::Base
2   has_many :children, class_name: "ParanoiacChild"
3 end
4
5 class ParanoiacChild < ActiveRecord::Base
6   acts_as_paranoid
7   belongs_to :parent
8
9   # You may need to provide a foreign_key like this
```

---

```
10 belongs_to :parent_including_deleted, class_name: "Parent",
11     foreign_key: 'parent_id', with_deleted: true
12 end
13
14 parent = Parent.first
15 child = parent.children.create
16 parent.destroy
17
18 child.parent #=> nil
19 child.parent_including_deleted #=> Parent (it works!)
```

## Callbacks

There are couple of callbacks that you may use when dealing with deletion and recovery of objects. There is `before_recover` and `after_recover` which will be triggered before and after the recovery of an object respectively.

Default ActiveRecord callbacks such as `before_destroy` and `after_destroy` will be triggered around `.destroy!` and `.destroy_fully!`.

```
1 class Paranoiac < ActiveRecord::Base
2   acts_as_paranoid
3
4   before_recover :set_counts
5   after_recover :update_logs
6 end
```

## Caveats

Watch out for these caveats:

- You cannot use scopes named `with_deleted` and `only_deleted`
- You cannot use scopes named `deleted_inside_time_window`, `deleted_before_time`, `deleted_after_time` if your paranoid column's type is `time`
- You cannot name association `*_with_deleted`
- `unscoped` will return all records, deleted or not

## Acknowledgements

- To Rick Olson for creating `acts_as_paranoid`
- To cheerfulstoic for adding recursive recovery

- 
- To Jonathan Vaught for adding paranoid validations
  - To Geoffrey Hichborn for improving the overall code quality and adding support for `after_commit`
  - To flah00 for adding support for STI-based associations (with `:dependent`)
  - To vikramdhillon for the idea and initial implementation of support for string column type
  - To Craig Walker for Rails 3.1 support and fixing various pending issues
  - To Charles G. for Rails 3.2 support and for making a desperately needed global code refactoring
  - To Gonalo Silva for supporting this gem prior to v0.4.3
  - To Jean Boussier for initial Rails 4.0.0 support
  - To Matijs van Zuijlen for Rails 4.1 and 4.2 support
  - To Andrey Ponomarenko for Rails 5 support
  - To Daniel Rice, Josh Bryant, and Romain Alexandre for Rails 6.0 support.

See [LICENSE](#).