

---

## Amoeba

Easy cloning of `active_record` objects including associations and several operations under associations and attributes.



### What?

The goal was to be able to easily and quickly reproduce ActiveRecord objects including their children, for example copying a blog post maintaining its associated tags or categories.

This gem is named “Amoeba” because amoebas are (small life forms that are) good at reproducing. Their children and grandchildren also reproduce themselves quickly and easily.

### Technical Details

An ActiveRecord extension gem to allow the duplication of associated child record objects when duplicating an active record model.

Rails 5.2, 6.0, 6.1 compatible. For Rails 4.2 to 5.1 use version 3.x.

### Features

- Supports the following association types
  - `has_many`
  - `has_one :through`
  - `has_many :through`
  - `has_and_belongs_to_many`
- A simple DSL for configuration of which fields to copy. The DSL can be applied to your rails models or used on the fly.
- Supports STI (Single Table Inheritance) children inheriting their parent amoeba settings.
- Multiple configuration styles such as inclusive, exclusive and indiscriminate (aka copy everything).
- Supports cloning of the children of Many-to-Many records or merely maintaining original associations
- Supports automatic drill-down i.e. recursive copying of child and grandchild records.

- 
- Supports preprocessing of fields to help indicate uniqueness and ensure the integrity of your data depending on your business logic needs, e.g. prepending “Copy of” or similar text.
  - Supports preprocessing of fields with custom lambda blocks so you can do basically whatever you want if, for example, you need some custom logic while making copies.
  - Amoeba can perform the following preprocessing operations on fields of copied records
    - set
    - prepend
    - append
    - nullify
    - customize
    - regex

## Usage

### Installation

is hopefully as you would expect:

```
1 gem install amoeba
```

or just add it to your Gemfile:

```
1 gem 'amoeba'
```

Configure your models with one of the styles below and then just run the `amoeba_dup` method on your model where you would run the `dup` method normally:

```
1 p = Post.create(:title => "Hello World!", :content => "Lorum ipsum  
  dolor")  
2 p.comments.create(:content => "I love it!")  
3 p.comments.create(:content => "This sucks!")  
4  
5 puts Comment.all.count # should be 2  
6  
7 my_copy = p.amoeba_dup  
8 my_copy.save  
9  
10 puts Comment.all.count # should be 4
```

By default, when enabled, amoeba will copy any and all associated child records automatically and associate them with the new parent record.

You can configure the behavior to only include fields that you list or to only include fields that you don't exclude. Of the three, the most performant will be the indiscriminate style, followed by the

---

inclusive style, and the exclusive style will be the slowest because of the need for an extra explicit check on each field. This performance difference is likely negligible enough that you can choose the style to use based on which is easiest to read and write, however, if your data tree is large enough and you need control over what fields get copied, inclusive style is probably a better choice than exclusive style.

## Configuration

Please note that these examples are only loose approximations of real world scenarios and may not be particularly realistic, they are only for the purpose of demonstrating feature usage.

**Indiscriminate Style** This is the most basic usage case and will simply enable the copying of any known associations.

If you have some models for a blog about like this:

```
1 class Post < ActiveRecord::Base
2   has_many :comments
3 end
4
5 class Comment < ActiveRecord::Base
6   belongs_to :post
7 end
```

simply add the amoeba configuration block to your model and call the enable method to enable the copying of child records, like this:

```
1 class Post < ActiveRecord::Base
2   has_many :comments
3
4   amoeba do
5     enable
6   end
7 end
8
9 class Comment < ActiveRecord::Base
10  belongs_to :post
11 end
```

Child records will be automatically copied when you run the `amoeba_dup` method.

**Inclusive Style** If you only want some of the associations copied but not others, you may use the inclusive style:

---

```
1 class Post < ActiveRecord::Base
2   has_many :comments
3   has_many :tags
4   has_many :authors
5
6   amoeba do
7     enable
8     include_association :tags
9     include_association :authors
10  end
11 end
12
13 class Comment < ActiveRecord::Base
14   belongs_to :post
15 end
```

Using the inclusive style within the amoeba block actually implies that you wish to enable amoeba, so there is no need to run the enable method, though it won't hurt either:

```
1 class Post < ActiveRecord::Base
2   has_many :comments
3   has_many :tags
4   has_many :authors
5
6   amoeba do
7     include_association :tags
8     include_association :authors
9   end
10 end
11
12 class Comment < ActiveRecord::Base
13   belongs_to :post
14 end
```

You may also specify fields to be copied by passing an array. If you call the `include_association` with a single value, it will be appended to the list of already included fields. If you pass an array, your array will overwrite the original values.

```
1 class Post < ActiveRecord::Base
2   has_many :comments
3   has_many :tags
4   has_many :authors
5
6   amoeba do
7     include_association [:tags, :authors]
8   end
9 end
10
11 class Comment < ActiveRecord::Base
```

---

```
12   belongs_to :post
13 end
```

These examples will copy the post's tags and authors but not its comments.

The inclusive style, when used, will automatically disable any other style that was previously selected.

**Exclusive Style** If you have more fields to include than to exclude, you may wish to shorten the amount of typing and reading you need to do by using the exclusive style. All fields that are not explicitly excluded will be copied:

```
1 class Post < ActiveRecord::Base
2   has_many :comments
3   has_many :tags
4   has_many :authors
5
6   amoeba do
7     exclude_association :comments
8   end
9 end
10
11 class Comment < ActiveRecord::Base
12   belongs_to :post
13 end
```

This example does the same thing as the inclusive style example, it will copy the post's tags and authors but not its comments. As with inclusive style, there is no need to explicitly enable amoeba when specifying fields to exclude.

The exclusive style, when used, will automatically disable any other style that was previously selected, so if you selected include fields, and then you choose some exclude fields, the `exclude_association` method will disable the previously selected inclusive style and wipe out any corresponding include fields.

**Conditions** Also if you need to path extra condition for include or exclude relationship you can path method name to `:if` option.

```
1 class Post < ActiveRecord::Base
2   has_many :comments
3   has_many :tags
4
5   amoeba do
6     include_association :comments, if: :popular?
7   end
8
```

---

```
9   def popular?
10     likes > 15
11   end
12 end
```

After call `Post.first.amoeba_dup` if `likes` is larger 15 than all comments will be duplicated too, but in another situation - no relations will be cloned. Same behavior will be for `exclude_association`.

**Be aware!** If you wrote:

```
1  class Post < ActiveRecord::Base
2    has_many :comments
3    has_many :tags
4
5    amoeba do
6      exclude_association :tags
7      include_association :comments, if: :popular?
8    end
9
10   def popular?
11     likes > 15
12   end
13 end
```

inclusion strategy will be chosen regardless of the result of `popular?` method call (the same for reverse situation).

**Cloning** If you are using a Many-to-Many relationship, you may tell amoeba to actually make duplicates of the original related records rather than merely maintaining association with the original records. Cloning is easy, merely tell amoeba which fields to clone in the same way you tell it which fields to include or exclude.

```
1  class Post < ActiveRecord::Base
2    has_and_belongs_to_many :warnings
3
4    has_many :post_widgets
5    has_many :widgets, :through => :post_widgets
6
7    amoeba do
8      enable
9      clone [:widgets, :warnings]
10   end
11 end
12
13 class Warning < ActiveRecord::Base
14   has_and_belongs_to_many :posts
15 end
```

---

```
16
17 class PostWidget < ActiveRecord::Base
18   belongs_to :widget
19   belongs_to :post
20 end
21
22 class Widget < ActiveRecord::Base
23   has_many :post_widgets
24   has_many :posts, :through => :post_widgets
25 end
```

This example will actually duplicate the warnings and widgets in the database. If there were originally 3 warnings in the database then, upon duplicating a post, you will end up with 6 warnings in the database. This is in contrast to the default behavior where your new post would merely be re-associated with any previously existing warnings and those warnings themselves would not be duplicated.

**Limiting Association Types** By default, amoeba recognizes and attempts to copy any children of the following association types:

- has one
- has many
- has and belongs to many

You may control which association types amoeba applies itself to by using the `recognize` method within the amoeba configuration block.

```
1 class Post < ActiveRecord::Base
2   has_one :config
3   has_many :comments
4   has_and_belongs_to_many :tags
5
6   amoeba do
7     recognize [:has_one, :has_and_belongs_to_many]
8   end
9 end
10
11 class Comment < ActiveRecord::Base
12   belongs_to :post
13 end
14
15 class Tag < ActiveRecord::Base
16   has_and_belongs_to_many :posts
17 end
```

This example will copy the post's configuration data and keep tags associated with the new post, but will not copy the post's comments because amoeba will only recognize and copy children of

---

`has_one` and `has_and_belongs_to_many` associations and in this example, comments are not an `has_and_belongs_to_many` association.

## Field Preprocessors

**Nullify** If you wish to prevent a regular (non `has_*` association based) field from retaining it's value when copied, you may “zero out” or “nullify” the field, like this:

```
1 class Topic < ActiveRecord::Base
2   has_many :posts
3 end
4
5 class Post < ActiveRecord::Base
6   belongs_to :topic
7   has_many :comments
8
9   amoeba do
10     enable
11     nullify :date_published
12     nullify :topic_id
13   end
14 end
15
16 class Comment < ActiveRecord::Base
17   belongs_to :post
18 end
```

This example will copy all of a post's comments. It will also nullify the publishing date and dissociate the post from its original topic.

Unlike inclusive and exclusive styles, specifying null fields will not automatically enable amoeba to copy all child records. As with any active record object, the default field value will be used instead of `nil` if a default value exists on the migration.

**Set** If you wish to just set a field to an arbitrary value on all duplicated objects you may use the `set` directive. For example, if you wanted to copy an object that has some kind of approval process associated with it, you likely may wish to set the new object's state to be open or “in progress” again.

```
1 class Post < ActiveRecord::Base
2   amoeba do
3     set :state_tracker => "open_for_editing"
4   end
5 end
```

In this example, when a post is duplicated, it's `state_tracker` field will always be given a value of `open_for_editing` to start.



---

**Prepend** You may add a string to the beginning of a copied object's field during the copy phase:

```
1 class Post < ActiveRecord::Base
2   amoeba do
3     enable
4     prepend :title => "Copy of "
5   end
6 end
```

**Append** You may add a string to the end of a copied object's field during the copy phase:

```
1 class Post < ActiveRecord::Base
2   amoeba do
3     enable
4     append :title => "Copy of "
5   end
6 end
```

**Regex** You may run a search and replace query on a copied object's field during the copy phase:

```
1 class Post < ActiveRecord::Base
2   amoeba do
3     enable
4     regex :contents => { :replace => /dog/, :with => 'cat' }
5   end
6 end
```

## Custom Methods

**Customize** You may run a custom method or methods to do basically anything you like, simply pass a lambda block, or an array of lambda blocks to the `customize` directive. Each block must have the same form, meaning that each block must accept two parameters, the original object and the newly copied object. You may then do whatever you wish, like this:

```
1 class Post < ActiveRecord::Base
2   amoeba do
3     prepend :title => "Hello world! "
4
5     customize(lambda { |original_post, new_post|
6       if original_post.foo == "bar"
7         new_post.baz = "qux"
8       end
9     })
10  end
```

---

```
11     append :comments => "... know what I'm sayin?"
12   end
13 end
```

or this, using an array:

```
1  class Post < ActiveRecord::Base
2    has_and_belongs_to_many :tags
3
4    amoeba do
5      include_association :tags
6
7      customize([
8        lambda do |orig_obj,copy_of_obj|
9          # good stuff goes here
10        end,
11
12        lambda do |orig_obj,copy_of_obj|
13          # more good stuff goes here
14        end
15      ])
16    end
17 end
```

**Override** Lambda blocks passed to `customize` run, by default, after all copying and field pre-processing. If you wish to run a method before any customization or field pre-processing, you may use `override` the cousin of `customize`. Usage is the same as above.

```
1  class Post < ActiveRecord::Base
2    amoeba do
3      prepend :title => "Hello world! "
4
5      override(lambda { |original_post,new_post|
6        if original_post.foo == "bar"
7          new_post.baz = "qux"
8        end
9      })
10
11      append :comments => "... know what I'm sayin?"
12    end
13 end
```

**Chaining** You may apply a single preprocessor to multiple fields at once.

```
1  class Post < ActiveRecord::Base
2    amoeba do
3      enable
```

---

```
4     prepend :title => "Copy of ", :contents => "Copied contents: "
5   end
6 end
```

**Stacking** You may apply multiple preprocessing directives to a single model at once.

```
1 class Post < ActiveRecord::Base
2   amoeba do
3     prepend :title => "Copy of ", :contents => "Original contents: "
4     append :contents => " (copied version)"
5     regex :contents => {:replace => /dog/, :with => 'cat'}
6   end
7 end
```

This example should result in something like this:

```
1 post = Post.create(
2   :title => "Hello world",
3   :contents => "I like dogs, dogs are awesome."
4 )
5
6 new_post = post.amoeba_dup
7
8 new_post.title # "Copy of Hello world"
9 new_post.contents # "Original contents: I like cats, cats are awesome.
   (copied version)"
```

Like `nullify`, the preprocessing directives do not automatically enable the copying of associated child records. If only preprocessing directives are used and you do want to copy child records and no `include_association` or `exclude_association` list is provided, you must still explicitly enable the copying of child records by calling the `enable` method from within the `amoeba` block on your model.

## Precedence

You may use a combination of configuration methods within each model's `amoeba` block. Recognized association types take precedence over inclusion or exclusion lists. Inclusive style takes precedence over exclusive style, and these two explicit styles take precedence over the indiscriminate style. In other words, if you list fields to copy, `amoeba` will only copy the fields you list, or only copy the fields you don't exclude as the case may be. Additionally, if a field type is not recognized it will not be copied, regardless of whether it appears in an inclusion list. If you want `amoeba` to automatically copy all of your child records, do not list any fields using either `include_association` or `exclude_association`.

---

The following example syntax is perfectly valid, and will result in the usage of inclusive style. The order in which you call the configuration methods within the amoeba block does not matter:

```
1 class Topic < ActiveRecord::Base
2   has_many :posts
3 end
4
5 class Post < ActiveRecord::Base
6   belongs_to :topic
7   has_many :comments
8   has_many :tags
9   has_many :authors
10
11   amoeba do
12     exclude_association :authors
13     include_association :tags
14     nullify :date_published
15     prepend :title => "Copy of "
16     append :contents => " (copied version)"
17     regex :contents => {:replace => /dog/, :with => 'cat'}
18     include_association :authors
19     enable
20     nullify :topic_id
21   end
22 end
23
24 class Comment < ActiveRecord::Base
25   belongs_to :post
26 end
```

This example will copy all of a post's tags and authors, but not its comments. It will also nullify the publishing date and dissociate the post from its original topic. It will also preprocess the post's fields as in the previous preprocessing example.

Note that, because of precedence, inclusive style is used and the list of exclude fields is never consulted. Additionally, the `enable` method is redundant because amoeba is automatically enabled when using `include_association`.

The preprocessing directives are run after child records are copied and are run in this order.

1. Null fields
2. Prepends
3. Appends
4. Search and Replace

Preprocessing directives do not affect inclusion and exclusion lists.

---

## Recurring

You may cause amoeba to keep copying down the chain as far as you like, simply add amoeba blocks to each model you wish to have copy its children. Amoeba will automatically recurse into any enabled grandchildren and copy them as well.

```
1 class Post < ActiveRecord::Base
2   has_many :comments
3
4   amoeba do
5     enable
6   end
7 end
8
9 class Comment < ActiveRecord::Base
10  belongs_to :post
11  has_many :ratings
12
13  amoeba do
14    enable
15  end
16 end
17
18 class Rating < ActiveRecord::Base
19  belongs_to :comment
20 end
```

In this example, when a post is copied, amoeba will copy each all of a post's comments and will also copy each comment's ratings.

## Has One Through

Using the `has_one :through` association is simple, just be sure to enable amoeba on the each model with a `has_one` association and amoeba will automatically and recursively drill down, like so:

```
1 class Supplier < ActiveRecord::Base
2   has_one :account
3   has_one :history, :through => :account
4
5   amoeba do
6     enable
7   end
8 end
9
10 class Account < ActiveRecord::Base
11  belongs_to :supplier
```

---

```
12   has_one :history
13
14   amoeba do
15     enable
16   end
17 end
18
19 class History < ActiveRecord::Base
20   belongs_to :account
21 end
```

## Has Many Through

Copying of `has_many :through` associations works automatically. They perform the copy in the same way as the `has_and_belongs_to_many` association, meaning the actual child records are not copied, but rather the associations are simply maintained. You can add some field preprocessors to the middle model if you like but this is not strictly necessary:

```
1 class Assembly < ActiveRecord::Base
2   has_many :manifests
3   has_many :parts, :through => :manifests
4
5   amoeba do
6     enable
7   end
8 end
9
10 class Manifest < ActiveRecord::Base
11   belongs_to :assembly
12   belongs_to :part
13
14   amoeba do
15     prepend :notes => "Copy of "
16   end
17 end
18
19 class Part < ActiveRecord::Base
20   has_many :manifests
21   has_many :assemblies, :through => :manifests
22
23   amoeba do
24     enable
25   end
26 end
```

---

## On The Fly Configuration

You may control how amoeba copies your object, on the fly, by passing a configuration block to the model's amoeba method. The configuration method is static but the configuration is applied on a per instance basis.

```
1 class Post < ActiveRecord::Base
2   has_many :comments
3
4   amoeba do
5     enable
6     prepend :title => "Copy of "
7   end
8 end
9
10 class Comment < ActiveRecord::Base
11   belongs_to :post
12 end
13
14 class PostsController < ActionController
15   def duplicate_a_post
16     old_post = Post.create(
17       :title => "Hello world",
18       :contents => "Lorum ipsum"
19     )
20
21     old_post.class.amoeba do
22       prepend :contents => "Here's a copy: "
23     end
24
25     new_post = old_post.amoeba_dup
26
27     new_post.title # should be "Copy of Hello world"
28     new_post.contents # should be "Here's a copy: Lorum ipsum"
29     new_post.save
30   end
31 end
```

## Inheritance

If you are using the Single Table Inheritance provided by ActiveRecord, you may cause amoeba to automatically process child classes in the same way as their parents. All you need to do is call the `propagate` method within the amoeba block of the parent class and all child classes should copy in a similar manner.

```
1 create_table :products, :force => true do |t|
2   t.string :type # this is the STI column
```

---

```
3
4 # these belong to all products
5 t.string :title
6 t.decimal :price
7
8 # these are for shirts only
9 t.decimal :sleeve_length
10 t.decimal :collar_size
11
12 # these are for computers only
13 t.integer :ram_size
14 t.integer :hard_drive_size
15 end
16
17 class Product < ActiveRecord::Base
18   has_many :images
19   has_and_belongs_to_many :categories
20
21   amoeba do
22     enable
23     propagate
24   end
25 end
26
27 class Shirt < Product
28 end
29
30 class Computer < Product
31 end
32
33 class ProductsController
34   def some_method
35     my_shirt = Shirt.find(1)
36     my_shirt.amoeba_dup
37     my_shirt.save
38
39     # this shirt should now:
40     # - have its own copy of all parent images
41     # - be in the same categories as the parent
42   end
43 end
```

This example should duplicate all the images and sections associated with this Shirt, which is a child of Product

**Parenting Style** By default, propagation uses submissive parenting, meaning the config settings on the parent will be applied, but any child settings, if present, will either add to or overwrite the parent settings depending on how you call the DSL methods.



---

You may change this behavior, the so called “parenting style”, to give preference to the parent settings or to ignore any and all child settings.

**Relaxed Parenting** The `:relaxed` parenting style will prefer parent settings.

```
1 class Product < ActiveRecord::Base
2   has_many :images
3   has_and_belongs_to_many :sections
4
5   amoeba do
6     exclude_association :images
7     propagate :relaxed
8   end
9 end
10
11 class Shirt < Product
12   include_association :images
13   include_association :sections
14   prepend :title => "Copy of "
15 end
```

In this example, the conflicting `include_association` settings on the child will be ignored and the parent `exclude_association` setting will be used, while the `prepend` setting on the child will be honored because it doesn’t conflict with the parent.

**Strict Parenting** The `:strict` style will ignore child settings altogether and inherit any parent settings.

```
1 class Product < ActiveRecord::Base
2   has_many :images
3   has_and_belongs_to_many :sections
4
5   amoeba do
6     exclude_association :images
7     propagate :strict
8   end
9 end
10
11 class Shirt < Product
12   include_association :images
13   include_association :sections
14   prepend :title => "Copy of "
15 end
```

In this example, the only processing that will happen when a `Shirt` is duplicated is whatever processing is allowed by the parent. So in this case the parent’s `exclude_association` directive takes prece-

---

dence over the child's `include_association` settings, and not only that, but none of the other settings for the child are used either. The `prepend` setting of the child is completely ignored.

**Parenting and Precedence** Because of the two general forms of DSL config parameter usage, you may wish to make yourself mindful of how your coding style will affect the outcome of duplicating an object.

Just remember that:

- If you pass an array you will wipe all previous settings
- If you pass single values, you will add to currently existing settings

This means that, for example:

- When using the submissive parenting style, you can child take full precedence on a per field basis by passing an array of config values. This will cause the setting from the parent to be overridden instead of added to.
- When using the relaxed parenting style, you can still let the parent take precedence on a per field basis by passing an array of config values. This will cause the setting for that child to be overridden instead of added to.

**A Submissive Override Example** This version will use both the parent and child settings, so both the images and sections will be copied.

```
1 class Product < ActiveRecord::Base
2   has_many :images
3   has_and_belongs_to_many :sections
4
5   amoeba do
6     include_association :images
7     propagate
8   end
9 end
10
11 class Shirt < Product
12   include_association :sections
13 end
```

The next version will use only the child settings because passing an array will override any previous settings rather than adding to them and the child config takes precedence in the `submissive` parenting style. So in this case only the sections will be copied.

```
1 class Product < ActiveRecord::Base
2   has_many :images
```

---

```
3   has_and_belongs_to_many :sections
4
5   amoeba do
6     include_association :images
7     propagate
8   end
9 end
10
11 class Shirt < Product
12   include_association [:sections]
13 end
```

**A Relaxed Override Example** This version will use both the parent and child settings, so both the images and sections will be copied.

```
1 class Product < ActiveRecord::Base
2   has_many :images
3   has_and_belongs_to_many :sections
4
5   amoeba do
6     include_association :images
7     propagate :relaxed
8   end
9 end
10
11 class Shirt < Product
12   include_association :sections
13 end
```

The next version will use only the parent settings because passing an array will override any previous settings rather than adding to them and the parent config takes precedence in the [relaxed](#) parenting style. So in this case only the images will be copied.

```
1 class Product < ActiveRecord::Base
2   has_many :images
3   has_and_belongs_to_many :sections
4
5   amoeba do
6     include_association [:images]
7     propagate
8   end
9 end
10
11 class Shirt < Product
12   include_association :sections
13 end
```

---

## Validating Nested Attributes

If you end up with some validation issues when trying to validate the presence of a child's `belongs_to` association, just be sure to include the `:inverse_of` declaration on your relationships and all should be well.

For example this will throw a validation error saying that your posts are invalid:

```
1 class Author < ActiveRecord::Base
2   has_many :posts
3
4   amoeba do
5     enable
6   end
7 end
8
9 class Post < ActiveRecord::Base
10  belongs_to :author
11  validates_presence_of :author
12
13  amoeba do
14    enable
15  end
16 end
17
18 author = Author.find(1)
19 author.amoeba_dup
20
21 author.save # this will fail validation
```

Where this will work fine:

```
1 class Author < ActiveRecord::Base
2   has_many :posts, :inverse_of => :author
3
4   amoeba do
5     enable
6   end
7 end
8
9 class Post < ActiveRecord::Base
10  belongs_to :author, :inverse_of => :posts
11  validates_presence_of :author
12
13  amoeba do
14    enable
15  end
16 end
17
18 author = Author.find(1)
```

---

```
19 author.amoeba_dup
20
21 author.save # this will pass validation
```

This issue is not amoeba specific and also occurs when creating new objects using `accepts_nested_attributes_for`, like this:

```
1 class Author < ActiveRecord::Base
2   has_many :posts
3   accepts_nested_attributes_for :posts
4 end
5
6 class Post < ActiveRecord::Base
7   belongs_to :author
8   validates_presence_of :author
9 end
10
11 # this will fail validation
12 author = Author.create({:name => "Jim Smith", :posts => [{:title => "
    Hello World", :contents => "Lorum ipsum dolor}]})
```

This issue with `accepts_nested_attributes_for` can also be solved by using `:inverse_of`, like this:

```
1 class Author < ActiveRecord::Base
2   has_many :posts, :inverse_of => :author
3   accepts_nested_attributes_for :posts
4 end
5
6 class Post < ActiveRecord::Base
7   belongs_to :author, :inverse_of => :posts
8   validates_presence_of :author
9 end
10
11 # this will pass validation
12 author = Author.create({:name => "Jim Smith", :posts => [{:title => "
    Hello World", :contents => "Lorum ipsum dolor}]})
```

The crux of the issue is that upon duplication, the new `Author` instance does not yet have an ID because it has not yet been persisted, so the `:posts` do not yet have an `:author_id` either, and thus no `:author` and thus they will fail validation. This issue may likely affect amoeba usage so if you get some validation failures, be sure to add `:inverse_of` to your models.

## Cloning using custom method

If you need to clone model with custom method you can use `through`:

---

---

```
1 class ChildPrototype < ActiveRecord::Base
2   amoeba do
3     through :become_child
4   end
5
6   def become_child
7     self.dup.becomes(Child)
8   end
9 end
10
11 class Child < ChildPrototype
12 end
```

After cloning we will get instance of `Child` instead of `ChildPrototype`

## Remapping associations

If you will need to do complex cloning with remapping associations name you can use `remapper`:

```
1 class ObjectPrototype < ActiveRecord::Base
2   has_many :child_prototypes
3
4   amoeba do
5     method :become_real
6     remapper :remap_associations
7   end
8
9   def become_real
10    self.dup().becomes( RealObject )
11  end
12
13  def remap_associations( name )
14    :childs if name == :child_prototypes
15  end
16 end
17
18 class RealObject < ObjectPrototype
19   has_many :childs
20 end
21
22 class ChildPrototype < ActiveRecord::Base
23   amoeba do
24     method :become_child
25   end
26
27   def become_child
28     self.dup().becomes( Child )
29   end
30 end
```

---

```
31
32 class Child < ChildPrototype
33 end
```

In result we will get next:

```
1 prototype = ObjectPrototype.new
2 prototype.child_prototypes << ChildPrototype.new
3 object = prototype.amoeba_dup
4 object.class # => RealObject
5 object.childs.first.class #=> Child
```

## Configuration Reference

Here is a static reference to the available configuration methods, usable within the amoeba block on your rails models.

### through

Set method what we will use for cloning model instead of `dup`.

for example:

```
1 amoeba do
2   through :supper_pupper_dup
3 end
4
5 def supper_pupper_dup
6   puts "multiplied by budding"
7   self.dup
8 end
```

## Controlling Associations

**enable** Enables amoeba in the default style of copying all known associated child records. Using the enable method is only required if you wish to enable amoeba but you are not using either the `include_association` or `exclude_association` directives. If you use either inclusive or exclusive style, amoeba is automatically enabled for you, so calling `enable` would be redundant, though it won't hurt.

**include\_association** Adds a field to the list of fields which should be copied. All associations not in this list will not be copied. This method may be called multiple times, once per desired field, or

---

you may pass an array of field names. Passing a single symbol will add to the list of included fields. Passing an array will empty the list and replace it with the array you pass.

**exclude\_association** Adds a field to the list of fields which should not be copied. Only the associations that are not in this list will be copied. This method may be called multiple times, once per desired field, or you may pass an array of field names. Passing a single symbol will add to the list of excluded fields. Passing an array will empty the list and replace it with the array you pass.

**clone** Adds a field to the list of associations which should have their associated children actually cloned. This means for example, that instead of just maintaining original associations with previously existing tags, a copy will be made of each tag, and the new record will be associated with these new tag copies rather than the old tag copies. This method may be called multiple times, once per desired field, or you may pass an array of field names. Passing a single symbol will add to the list of excluded fields. Passing an array will empty the list and replace it with the array you pass.

**propagate** This causes any inherited child models to take the same config settings when copied. This method may take up to one argument to control the so called “parenting style”. The argument should be one of `strict`, `relaxed` or `submissive`.

The default “parenting style” is `submissive`

for example

```
1 amoeba do
2   propagate :strict
3 end
```

will choose the strict parenting style of inherited settings.

**raised** This causes any child to behave with a (potentially) different “parenting style” than its actual parent. This method takes up to a single parameter for which there are three options, `strict`, `relaxed` and `submissive`.

The default “parenting style” is `submissive`

for example:

```
1 amoeba do
2   raised :relaxed
3 end
```



---

will choose the relaxed parenting style of inherited settings for this child. A parenting style set via the `raised` method takes precedence over the parenting style set using the `propagate` method.

**remapper** Set the method what will be used for remapping of association name. Method will have one argument - association name as Symbol. If method will return nil then association will not be remapped.

for example:

```
1 amoeba do
2   remapper :childs_to_parents
3 end
4
5 def childs_to_parents(association_name)
6   :parents if association_name == :childs
7 end
```

## Pre-Processing Fields

**nullify** Adds a field to the list of non-association based fields which should be set to nil during copy. All fields in this list will be set to `nil` - note that any nullified field will be given its default value if a default value exists on this model's migration. This method may be called multiple times, once per desired field, or you may pass an array of field names. Passing a single symbol will add to the list of null fields. Passing an array will empty the list and replace it with the array you pass.

**prepend** Prefix a field with some text. This only works for string fields. Accepts a hash of fields to prepend. The keys are the field names and the values are the prefix strings. An example scenario would be to add a string such as "Copy of" to your title field. Don't forget to include extra space to the right if you want it. Passing a hash will add each key value pair to the list of prepend directives. If you wish to empty the list of directives, you may pass the hash inside of an array like this `[{:title => "Copy of "}]`.

**append** Append some text to a field. This only works for string fields. Accepts a hash of fields to append. The keys are the field names and the values are the prefix strings. An example would be to add " (copied version)" to your description field. Don't forget to add a leading space if you want it. Passing a hash will add each key value pair to the list of append directives. If you wish to empty the list of directives, you may pass the hash inside of an array like this `[{:contents => "(copied version)"}]`.

---

**set** Set a field to a given value. This should work for almost any type of field. Accepts a hash of fields and the values you want them set to.. The keys are the field names and the values are the prefix strings. An example would be to add " (copied version)" to your description field. Don't forget to add a leading space if you want it. Passing a hash will add each key value pair to the list of append directives. If you wish to empty the list of directives, you may pass the hash inside of an array like this `[{:approval_state => "open_for_editing"}]`.

**regex** Globally search and replace the field for a given pattern. Accepts a hash of fields to run search and replace upon. The keys are the field names and the values are each a hash with information about what to find and what to replace it with. in the form of `{:find => "dog", :replace => "cat"}`. An example would be to replace all occurrences of the word "dog" with the word "cat", the parameter hash would look like this `:contents => {:replace => /dog/, :with => "cat"}`. Passing a hash will add each key value pair to the list of regex directives. If you wish to empty the list of directives, you may pass the hash inside of an array like this `[{:contents => {:replace => /dog/, :with => "cat"}}]`.

**override** Runs a custom method so you can do basically whatever you want. All you need to do is pass a lambda block or an array of lambda blocks that take two parameters, the original object and the new object copy. These blocks will run before any other duplication or field processing.

This method may be called multiple times, once per desired customizer block, or you may pass an array of lambdas. Passing a single lambda will add to the list of processing directives. Passing an array will empty the list and replace it with the array you pass.

**customize** Runs a custom method so you can do basically whatever you want. All you need to do is pass a lambda block or an array of lambda blocks that take two parameters, the original object and the new object copy. These blocks will run after all copying and field processing.

This method may be called multiple times, once per desired customizer block, or you may pass an array of lambdas. Passing a single lambda will add to the list of processing directives. Passing an array will empty the list and replace it with the array you pass.

## Known Limitations and Issues

The regular expression preprocessor uses case-sensitive `String#gsub`. Given the performance decreases inherent in using regular expressions already, the fact that character classes can essentially account for case-insensitive searches, the desire to keep the DSL simple and the general use cases for this gem, I don't see a good reason to add yet more decision based conditional syntax to accommodate using case-insensitive searches or singular replacements with `String#sub`. If you find yourself

---

wanting either of these features, by all means fork the code base and if you like your changes, submit a pull request.

The behavior when copying nested hierarchical models is undefined. Copying a category model which has a `parent_id` field pointing to the parent category, for example, is currently undefined.

The behavior when copying polymorphic `has_many` associations is also undefined. Support for these types of associations is planned for a future release.

## For Developers

You may run the rspec tests like this:

```
1 bundle exec rspec spec
```

## TODO

- add ability to cancel further processing from within an override block
- write some spec for the override method