

---

## Introduction

The `default_value_for` plugin allows one to define default values for ActiveRecord models in a declarative manner. For example:

```
1 class User < ActiveRecord::Base
2   default_value_for :name, "(no name)"
3   default_value_for :last_seen do
4     Time.now
5   end
6 end
7
8 u = User.new
9 u.name      # => "(no name)"
10 u.last_seen # => Mon Sep 22 17:28:38 +0200 2008
```

*Note:* critics might be interested in the “When (not) to use `default_value_for`?” section. Please read on.

## Installation

### Rails 3.2 - 4.2 / Ruby 1.9.3 and higher

The current version of `default_value_for` (3.x+) is compatible with Rails 3.2 or higher, and Ruby 1.9.3 and higher.

Add it to your Gemfile:

```
1 gem "default_value_for", "~> 3.0"
```

This gem is signed using PGP with the Phusion Software Signing key: <http://www.phusion.nl/about/gpg>. That key in turn is signed by the rubygems-openpgp Certificate Authority: <http://www.rubygems-openpgp-ca.org/>.

You can verify the authenticity of the gem by following The Complete Guide to Verifying Gems with rubygems-openpgp: <http://www.rubygems-openpgp-ca.org/blog/the-complete-guide-to-verifying-gems-with-rubygems-openpgp.html>

### Rails 3.0 - 3.1 / Ruby 1.9.3 and lower

To use `default_value_for` with older versions of Ruby and Rails, you must use the previous stable release, 2.0.3. This version works with Rails 3.0, 3.1, and 3.2; and Ruby 1.8.7 and higher. It **does not** work with Rails 4.

---

```
1 gem "default_value_for", "~> 2.0.3"
```

## Rails 2

To use `default_value_for` with Rails 2.x you must use an older version:

```
1 ./script/plugin install git://github.com/FooBarWidget/default_value_for
  .git -r release-1.0.7
```

## The `default_value_for` method

The `default_value_for` method is available in all ActiveRecord model classes.

The first argument is the name of the attribute for which a default value should be set. This may either be a Symbol or a String.

The default value itself may either be passed as the second argument:

```
1 default_value_for :age, 20
```

...or it may be passed as the return value of a block:

```
1 default_value_for :age do
2   if today_is_sunday?
3     20
4   else
5     30
6   end
7 end
```

If you pass a value argument, then the default value is static and never changes. However, if you pass a block, then the default value is retrieved by calling the block. This block is called not once, but every time a new record is instantiated and default values need to be filled in.

The latter form is especially useful if your model has a UUID column. One can generate a new, random UUID for every newly instantiated record:

```
1 class User < ActiveRecord::Base
2   default_value_for :uuid do
3     UuidGenerator.new.generate_uuid
4   end
5 end
6
7 User.new.uuid # => "51d6d6846f1d1b5c9a...."
8 User.new.uuid # => "ede292289e3484cb88...."
```

---

Note that record is passed to the block as an argument, in case you need it for whatever reason:

```
1 class User < ActiveRecord::Base
2   default_value_for :uuid do |x|
3     x # <--- a User object
4     UuidGenerator.new.generate_uuid
5   end
6 end
```

## default\_value\_for options

- `allows_nil` (default: true) - Sets explicitly passed nil values if option is set to true.

You can pass this options hash as 2nd parameter and have to pass the default value through the `:value` option in this case e.g.:

```
1 default_value_for :age, :value => 20, :allows_nil => false
```

You can still pass the default value through a block:

```
1 default_value_for :uuid, :allows_nil => false do
2   UuidGenerator.new.generate_uuid
3 end
```

## The default\_values method

As a shortcut, you can use `+default_values+` to set multiple default values at once.

```
1 default_values :age => 20,
2               :uuid => lambda { UuidGenerator.new.generate_uuid }
```

If you like to override `default_value_for` options for each attribute you can do so:

```
1 default_values :age => { :value => 20 },
2               :uuid => { :value => lambda { UuidGenerator.new.
                           generate_uuid }, :allows_nil => false }
```

The difference is purely aesthetic. If you have lots of default values which are constants or constructed with one-line blocks, `+default_values+` may look nicer. If you have default values constructed by longer blocks, `default_value_for` suit you better. Feel free to mix and match.

As a side note, due to specifics of Ruby's parser, you cannot say,

```
1 default_value_for :uuid { UuidGenerator.new.generate_uuid }
```

because it will not parse. One needs to write

---

```
1 default_value_for(:uuid) { UuidGenerator.new.generate_uuid }
```

instead. This is in part the inspiration for the `+default_values+` syntax.

## Rules

### Instantiation of new record

Upon instantiating a new record, the declared default values are filled into the record. You've already seen this in the above examples.

### Retrieval of existing record

Upon retrieving an existing record in the following case, the declared default values are *not* filled into the record. Consider the example with the UUID:

```
1 user = User.create
2 user.uuid # => "529c91b8bbd3e..."
3
4 user = User.find(user.id)
5 # UUID remains unchanged because it's retrieved from the database!
6 user.uuid # => "529c91b8bbd3e..."
```

But when the declared default value is set to not allow nil and nil is passed the default values will be set on retrieval. Consider this example:

```
1 default_value_for(:number, :allows_nil => false) { 123 }
2
3 user = User.create
4
5 # manual SQL by-passing active record and the default value for gem
  # logic through ActiveRecord's after_initialize callback
6 user.update_attribute(:number, nil)
7
8 # declared default value should be set
9 User.find(user.id).number # => 123 # = declared default value
```

### Mass-assignment

If a certain attribute is being assigned via the model constructor's mass-assignment argument, that the default value for that attribute will *not* be filled in:

---

```
1 user = User.new(:uuid => "hello")
2 user.uuid # => "hello"
```

However, if that attribute is protected by `+attr_protected+` or `+attr_accessible+`, then it will be filled in:

```
1 class User < ActiveRecord::Base
2   default_value_for :name, 'Joe'
3   attr_protected :name
4 end
5
6 user = User.new(:name => "Jane")
7 user.name # => "Joe"
8
9 # the without protection option will work as expected
10 user = User.new({:name => "Jane"}, :without_protection => true)
11 user.name # => "Jane"
```

Explicitly set nil values for accessible attributes will be accepted:

```
1 class User < ActiveRecord::Base
2   default_value_for :name, 'Joe'
3 end
4
5 user = User(:name => nil)
6 user.name # => nil
7
8 ... unless the accessible attribute is set to not allowing nil:
9
10 class User < ActiveRecord::Base
11   default_value_for :name, 'Joe', :allows_nil => false
12 end
13
14 user = User(:name => nil)
15 user.name # => "Joe"
```

## Inheritance

Inheritance works as expected. All default values are inherited by the child class:

```
1 class User < ActiveRecord::Base
2   default_value_for :name, 'Joe'
3 end
4
5 class SuperUser < User
6 end
7
8 SuperUser.new.name # => "Joe"
```

---

## Attributes that aren't database columns

`default_value_for` also works with attributes that aren't database columns. It works with anything for which there's an assignment method:

```
1 # Suppose that your 'users' table only has a 'name' column.
2 class User < ActiveRecord::Base
3   default_value_for :name, 'Joe'
4   default_value_for :age, 20
5   default_value_for :registering, true
6
7   attr_accessor :age
8
9   def registering=(value)
10     @registering = true
11   end
12 end
13
14 user = User.new
15 user.age      # => 20
16 user.instance_variable_get('@registering')  # => true
```

## Default values are duplicated

The given default values are duplicated when they are filled in, so if you mutate a value that was filled in with a default value, then it will not affect all subsequent default values:

```
1 class Author < ActiveRecord::Base
2   # This model only has a 'name' attribute.
3 end
4
5 class Book < ActiveRecord::Base
6   belongs_to :author
7
8   # By default, a Book belongs to a new, unsaved author.
9   default_value_for :author, Author.new
10 end
11
12 book1 = Book.new
13 book1.author.name # => nil
14 # This does not mutate the default value:
15 book1.author.name = "John"
16
17 book2 = Book.new
18 book2.author.name # => nil
```

However the duplication is shallow. If you modify any objects that are referenced by the default value then it will affect subsequent default values:

---

```
1 class Author < ActiveRecord::Base
2   attr_accessor :useless_hash
3   default_value_for :useless_hash, { :foo => [] }
4 end
5
6 author1 = Author.new
7 author1.useless_hash # => { :foo => [] }
8 # This mutates the referred array:
9 author1.useless_hash[:foo] << 1
10
11 author2 = Author.new
12 author2.useless_hash # => { :foo => [1] }
```

You can prevent this from happening by passing a block to `default_value_for`, which returns a new object instance with fresh references every time:

```
1 class Author < ActiveRecord::Base
2   attr_accessor :useless_hash
3   default_value_for :useless_hash do
4     { :foo => [] }
5   end
6 end
7
8 author1 = Author.new
9 author1.useless_hash # => { :foo => [] }
10 author1.useless_hash[:foo] << 1
11
12 author2 = Author.new
13 author2.useless_hash # => { :foo => [] }
```

## Caveats

A conflict can occur if your model class overrides the ‘initialize’ method, because this plugin overrides ‘initialize’ as well to do its job.

```
1 class User < ActiveRecord::Base
2   def initialize # <-- this constructor causes problems
3     super(:name => 'Name cannot be changed in constructor')
4   end
5 end
```

We recommend you to alias chain your initialize method in models where you use `default_value_for`:

```
1 class User < ActiveRecord::Base
2   default_value_for :age, 20
3
```

---

```
4 def initialize_with_my_app
5   initialize_without_my_app(:name => 'Name cannot be changed in
      constructor')
6 end
7
8 alias_method_chain :initialize, :my_app
9 end
```

Also, stick with the following rules:

- There is no need to `+alias_method_chain+` your initialize method in models that don't use `default_value_for`.
- Make sure that `+alias_method_chain+` is called *after* the last `default_value_for` occurrence.

If your default value is accidentally similar to `default_value_for`'s options hash wrap your default value like this:

```
1 default_value_for :attribute_name, :value => { :value => 123, :
      other_value => 1234 }
```

### When (not) to use `default_value_for`?

You can also specify default values in the database schema. For example, you can specify a default value in a migration as follows:

```
1 create_table :users do |t|
2   t.string :username, :null => false, :default => 'default username'
3   t.integer :age, :null => false, :default => 20
4 end
```

This has similar effects as passing the default value as the second argument to `default_value_for`:

```
1 default_value_for(:username, 'default_username')
2 default_value_for(:age, 20)
```

Default values are filled in whether you use the schema defaults or the `default_value_for` defaults:

```
1 user = User.new
2 user.username # => 'default username'
3 user.age      # => 20
```

It's recommended that you use this over `default_value_for` whenever possible.



---

However, it's not possible to specify a schema default for serialized columns. With `default_value_for`, you can:

```
1 class User < ActiveRecord::Base
2   serialize :color
3   default_value_for :color, [255, 0, 0]
4 end
```

And if schema defaults don't provide the flexibility that you need, then `default_value_for` is the perfect choice. For example, with `default_value_for` you could specify a per-environment default:

```
1 class User < ActiveRecord::Base
2   if Rails.env == "development"
3     default_value_for :is_admin, true
4   end
5 end
```

Or, as you've seen in an earlier example, you can use `default_value_for` to generate a default random UUID:

```
1 class User < ActiveRecord::Base
2   default_value_for :uuid do
3     UuidGenerator.new.generate_uuid
4   end
5 end
```

Or you could use it to generate a timestamp that's relative to the time at which the record is instantiated:

```
1 class User < ActiveRecord::Base
2   default_value_for :account_expires_at do
3     3.years.from_now
4   end
5 end
6
7 User.new.account_expires_at # => Mon Sep 22 18:43:42 +0200 2008
8 sleep(2)
9 User.new.account_expires_at # => Mon Sep 22 18:43:44 +0200 2008
```

Finally, it's also possible to specify a default via an association:

```
1 # Has columns: 'name' and 'default_price'
2 class SuperMarket < ActiveRecord::Base
3   has_many :products
4 end
5
6 # Has columns: 'name' and 'price'
7 class Product < ActiveRecord::Base
```

---

```

8   belongs_to :super_market
9
10  default_value_for :price do |product|
11    product.super_market.default_price
12  end
13 end
14
15 super_market = SuperMarket.create(:name => 'Albert Zwijn', :
    default_price => 100)
16 soap = super_market.products.create(:name => 'Soap')
17 soap.price   # => 100

```

### What about `before_validate`/`before_save`?

True, `before_validate` and `before_save` does what we want if we're only interested in filling in a default before saving. However, if one wants to be able to access the default value even before saving, then be prepared to write a lot of code. Suppose that we want to be able to access a new record's UUID, even before it's saved. We could end up with the following code:

```

1  # In the controller
2  def create
3    @user = User.new(params[:user])
4    @user.generate_uuid
5    email_report_to_admin("#{@user.username} with UUID #{@user.uuid}
      created.")
6    @user.save!
7  end
8
9  # Model
10 class User < ActiveRecord::Base
11   before_save :generate_uuid_if_necessary
12
13   def generate_uuid
14     self.uuid = ...
15   end
16
17   private
18   def generate_uuid_if_necessary
19     if uuid.blank?
20       generate_uuid
21     end
22   end
23 end

```

The need to manually call `generate_uuid` here is ugly, and one can easily forget to do that. Can we do better? Let's see:

```

1  # Controller

```

---

```
2 def create
3   @user = User.new(params[:user])
4   email_report_to_admin("#{@user.username} with UUID #{@user.uuid}
5     created.")
6   @user.save!
7 end
8 # Model
9 class User < ActiveRecord::Base
10  before_save :generate_uuid_if_necessary
11
12  def uuid
13    value = read_attribute('uuid')
14    if !value
15      value = generate_uuid
16      write_attribute('uuid', value)
17    end
18    value
19  end
20
21  # We need to override this too, otherwise User.new.attributes won't
22    return
23  # a default UUID value. I've never tested with User.create() so maybe
24    we
25  # need to override even more things.
26  def attributes
27    uuid
28    super
29  end
30
31  private
32  def generate_uuid_if_necessary
33    uuid # Reader method automatically generates UUID if it doesn't
34      exist
35  end
36 end
```

That's an awful lot of code. Using `default_value_for` is easier, don't you think?

### What about other plugins?

I've only been able to find 2 similar plugins:

- Default Value: [http://agilewebdevelopment.com/plugins/default\\_value](http://agilewebdevelopment.com/plugins/default_value)
- ActiveRecord Defaults: [http://agilewebdevelopment.com/plugins/activerecord\\_defaults](http://agilewebdevelopment.com/plugins/activerecord_defaults)

'Default Value' appears to be unmaintained; its SVN link is broken. This leaves only 'ActiveRecord Defaults'. However, it is semantically dubious, which leaves it wide open for corner cases. For ex-

---

ample, it is not clearly specified what ActiveRecord Defaults will do when attributes are protected by `+attr_protected+` or `+attr_accessible+`. It is also not clearly specified what one is supposed to do if one needs a custom `+initialize+` method in the model.

I've taken my time to thoroughly document `default_value_for`'s behavior.

## Credits

I've wanted such functionality for a while now and it baffled me that ActiveRecord doesn't provide a clean way for me to specify default values. After reading [http://groups.google.com/group/rubyonrails-core/browse\\_thread/thread/b509a2fe2b62ac5/3e8243fa1954a935](http://groups.google.com/group/rubyonrails-core/browse_thread/thread/b509a2fe2b62ac5/3e8243fa1954a935), it became clear that someone needs to write a plugin. This is the result.

Thanks to Pratik Naik for providing the initial code snippet on which this plugin is based on: <http://m.onkey.org/2007/7/24/how-to-set-default-values-in-your-model>

Thanks to Matthew Draper for Rails 5 support.

Thanks to Norman Clarke and Tom Mango for Rails 4 support.