Active Record Callbacks

January 13, 2015

This guide teaches you how to hook into the life cycle of your Active Record objects.
After reading this guide, you will know:

- The life cycle of Active Record objects.
- How to create callback methods that respond to events in the object life cycle.
- How to create special classes that encapsulate common behavior for your callbacks.

1 The Object Life Cycle

During the normal operation of a Rails application, objects may be created, updated, and destroyed. Active Record provides hooks into this *object life cycle* so that you can control your application and its data.

Callbacks allow you to trigger logic before or after an alteration of an object's state.

2 Callbacks Overview

Callbacks are methods that get called at certain moments of an object's life cycle. With callbacks it is possible to write code that will run whenever an Active Record object is created, saved, updated, deleted, validated, or loaded from the database.

2.1 Callback Registration

In order to use the available callbacks, you need to register them. You can implement the callbacks as ordinary methods and use a macro-style class method to register them as callbacks:

```
class User < ActiveRecord::Base
  validates :login, :email, presence: true

  before_validation :ensure_login_has_a_value

  protected
    def ensure_login_has_a_value
      if login.nil?
        self.login = email unless email.blank?
      end
```

```
    end
end
```

The macro-style class methods can also receive a block. Consider using this style if the code inside your block is so short that it fits in a single line:

```
class User < ActiveRecord::Base
  validates :login, :email, presence: true

  before_create do
    self.name = login.capitalize if name.blank?
  end
end
```

Callbacks can also be registered to only fire on certain life cycle events:

```
class User < ActiveRecord::Base
  before_validation :normalize_name, on: :create

  # :on takes an array as well
  after_validation :set_location, on: [ :create, :update ]

  protected
    def normalize_name
      self.name = self.name.downcase.titleize
    end

    def set_location
      self.location = LocationService.query(self)
    end
end
```

It is considered good practice to declare callback methods as protected or private. If left public, they can be called from outside of the model and violate the principle of object encapsulation.

3 Available Callbacks

Here is a list with all the available Active Record callbacks, listed in the same order in which they will get called during the respective operations:

3.1 Creating an Object

- `before_validation`
- `after_validation`
- `before_save`
- `around_save`

- before_create
- around_create
- after_create
- after_save
- after_commit/after_rollback

3.2 Updating an Object

- before_validation
- after_validation
- before_save
- around_save
- before_update
- around_update
- after_update
- after_save
- after_commit/after_rollback

3.3 Destroying an Object

- before_destroy
- around_destroy
- after_destroy
- after_commit/after_rollback

after_save runs both on create and update, but always *after* the more specific callbacks after_create and after_update, no matter the order in which the macro calls were executed.

3.4 after_initialize and after_find

The after_initialize callback will be called whenever an Active Record object is instantiated, either by directly using new or when a record is loaded from the database. It can be useful to avoid the need to directly override your Active Record initialize method.

The after_find callback will be called whenever Active Record loads a record from the database. after_find is called before after_initialize if both are defined.

The after_initialize and after_find callbacks have no before_* counterparts, but they can be registered just like the other Active Record callbacks.

```
class User < ActiveRecord::Base
  after_initialize do |user|
    puts "You have initialized an object!"
  end

  after_find do |user|
    puts "You have found an object!"
```

```
  end
end

>> User.new
You have initialized an object!
=> #<User id: nil>

>> User.first
You have found an object!
You have initialized an object!
=> #<User id: 1>
```

3.5 after_touch

The `after_touch` callback will be called whenever an Active Record object is touched.

```
class User < ActiveRecord::Base
  after_touch do |user|
    puts "You have touched an object"
  end
end

>> u = User.create(name: 'Kuldeep')
=> #<User id: 1, name: "Kuldeep", created_at: "2013-11-25 12:17:49", updated_at:
"2013-11-25 12:17:49">

>> u.touch
You have touched an object
=> true
```

It can be used along with `belongs_to`:

```
class Employee < ActiveRecord::Base
  belongs_to :company, touch: true
  after_touch do
    puts 'An Employee was touched'
  end
end

class Company < ActiveRecord::Base
  has_many :employees
  after_touch :log_when_employees_or_company_touched

  private
  def log_when_employees_or_company_touched
    puts 'Employee/Company was touched'
  end
```

```
end

>> @employee = Employee.last
=> #<Employee id: 1, company_id: 1, created_at: "2013-11-25 17:04:22", updated_at:
"2013-11-25 17:05:05">

# triggers @employee.company.touch
>> @employee.touch
Employee/Company was touched
An Employee was touched
=> true
```

4 Running Callbacks

The following methods trigger callbacks:

- `create`
- `create!`
- `decrement!`
- `destroy`
- `destroy!`
- `destroy_all`
- `increment!`
- `save`
- `save!`
- `save(validate:  false)`
- `toggle!`
- `update_attribute`
- `update`
- `update!`
- `valid?`

Additionally, the `after_find` callback is triggered by the following finder methods:

- `all`
- `first`
- `find`
- `find_by`
- `find_by_*`
- `find_by_*!`
- `find_by_sql`
- `last`

The `after_initialize` callback is triggered every time a new object of the class is initialized.

The `find_by_*` and `find_by_*!` methods are dynamic finders generated automatically for every attribute. Learn more about them at the Dynamic finders section

5 Skipping Callbacks

Just as with validations, it is also possible to skip callbacks by using the following methods:

- `decrement`
- `decrement_counter`
- `delete`
- `delete_all`
- `increment`
- `increment_counter`
- `toggle`
- `touch`
- `update_column`
- `update_columns`
- `update_all`
- `update_counters`

These methods should be used with caution, however, because important business rules and application logic may be kept in callbacks. Bypassing them without understanding the potential implications may lead to invalid data.

6 Halting Execution

As you start registering new callbacks for your models, they will be queued for execution. This queue will include all your model's validations, the registered callbacks, and the database operation to be executed.

The whole callback chain is wrapped in a transaction. If any *before* callback method returns exactly `false` or raises an exception, the execution chain gets halted and a ROLLBACK is issued; *after* callbacks can only accomplish that by raising an exception.

Any exception that is not `ActiveRecord::Rollback` will be re-raised by Rails after the callback chain is halted. Raising an exception other than `ActiveRecord::Rollback` may break code that does not expect methods like `save` and `update_attributes` (which normally try to return `true` or `false`) to raise an exception.

7 Relational Callbacks

Callbacks work through model relationships, and can even be defined by them. Suppose an example where a user has many articles. A user's articles should be destroyed if the user is destroyed. Let's add an `after_destroy` callback to the `User` model by way of its relationship to the `Article` model:

```
class User < ActiveRecord::Base
  has_many :articles, dependent: :destroy
end

class Article < ActiveRecord::Base
  after_destroy :log_destroy_action
```

```
  def log_destroy_action
    puts 'Article destroyed'
  end
end

>> user = User.first
=> #<User id: 1>
>> user.articles.create!
=> #<Article id: 1, user_id: 1>
>> user.destroy
Article destroyed
=> #<User id: 1>
```

8 Conditional Callbacks

As with validations, we can also make the calling of a callback method conditional on the satisfaction of a given predicate. We can do this using the `:if` and `:unless` options, which can take a symbol, a string, a `Proc` or an `Array`. You may use the `:if` option when you want to specify under which conditions the callback **should** be called. If you want to specify the conditions under which the callback **should not** be called, then you may use the `:unless` option.

8.1 Using `:if` and `:unless` with a `Symbol`

You can associate the `:if` and `:unless` options with a symbol corresponding to the name of a predicate method that will get called right before the callback. When using the `:if` option, the callback won't be executed if the predicate method returns false; when using the `:unless` option, the callback won't be executed if the predicate method returns true. This is the most common option. Using this form of registration it is also possible to register several different predicates that should be called to check if the callback should be executed.

```
class Order < ActiveRecord::Base
  before_save :normalize_card_number, if: :paid_with_card?
end
```

8.2 Using `:if` and `:unless` with a String

You can also use a string that will be evaluated using `eval` and hence needs to contain valid Ruby code. You should use this option only when the string represents a really short condition:

```
class Order < ActiveRecord::Base
  before_save :normalize_card_number, if: "paid_with_card?"
end
```

8.3 Using `:if` and `:unless` with a `Proc`

Finally, it is possible to associate `:if` and `:unless` with a `Proc` object. This option is best suited when writing short validation methods, usually one-liners:

```
class Order < ActiveRecord::Base
  before_save :normalize_card_number,
    if: Proc.new { |order| order.paid_with_card? }
end
```

8.4 Multiple Conditions for Callbacks

When writing conditional callbacks, it is possible to mix both `:if` and `:unless` in the same callback declaration:

```
class Comment < ActiveRecord::Base
  after_create :send_email_to_author, if: :author_wants_emails?,
    unless: Proc.new { |comment| comment.article.ignore_comments? }
end
```

9 Callback Classes

Sometimes the callback methods that you'll write will be useful enough to be reused by other models. Active Record makes it possible to create classes that encapsulate the callback methods, so it becomes very easy to reuse them.

Here's an example where we create a class with an `after_destroy` callback for a `PictureFile` model:

```
class PictureFileCallbacks
  def after_destroy(picture_file)
    if File.exist?(picture_file.filepath)
      File.delete(picture_file.filepath)
    end
  end
end
```

When declared inside a class, as above, the callback methods will receive the model object as a parameter. We can now use the callback class in the model:

```
class PictureFile < ActiveRecord::Base
  after_destroy PictureFileCallbacks.new
end
```

Note that we needed to instantiate a new `PictureFileCallbacks` object, since we declared our callback as an instance method. This is particularly useful if the callbacks make use of the state of the instantiated object. Often, however, it will make more sense to declare the callbacks as class methods:

```
class PictureFileCallbacks
  def self.after_destroy(picture_file)
    if File.exist?(picture_file.filepath)
      File.delete(picture_file.filepath)
    end
  end
end
```

If the callback method is declared this way, it won't be necessary to instantiate a `PictureFileCallbacks` object.

```
class PictureFile < ActiveRecord::Base
  after_destroy PictureFileCallbacks
end
```

You can declare as many callbacks as you want inside your callback classes.

10 Transaction Callbacks

There are two additional callbacks that are triggered by the completion of a database transaction: `after_commit` and `after_rollback`. These callbacks are very similar to the `after_save` callback except that they don't execute until after database changes have either been committed or rolled back. They are most useful when your active record models need to interact with external systems which are not part of the database transaction.

Consider, for example, the previous example where the `PictureFile` model needs to delete a file after the corresponding record is destroyed. If anything raises an exception after the `after_destroy` callback is called and the transaction rolls back, the file will have been deleted and the model will be left in an inconsistent state. For example, suppose that `picture_file_2` in the code below is not valid and the `save!` method raises an error.

```
PictureFile.transaction do
  picture_file_1.destroy
  picture_file_2.save!
end
```

By using the `after_commit` callback we can account for this case.

```
class PictureFile < ActiveRecord::Base
  after_commit :delete_picture_file_from_disk, on: [:destroy]

  def delete_picture_file_from_disk
    if File.exist?(filepath)
      File.delete(filepath)
    end
  end
end
```

the `:on` option specifies when a callback will be fired. If you don't supply the `:on` option the callback will fire for every action.

The `after_commit` and `after_rollback` callbacks are guaranteed to be called for all models created, updated, or destroyed within a transaction block. If any exceptions are raised within one of these callbacks, they will be ignored so that they don't interfere with the other callbacks. As such, if your callback code could raise an exception, you'll need to rescue it and handle it appropriately within the callback.

11 Feedback

You're encouraged to help improve the quality of this guide.

Please contribute if you see any typos or factual errors. To get started, you can read our documentation contributions section.

You may also find incomplete content, or stuff that is not up to date. Please do add any missing documentation for master. Make sure to check Edge Guides first to verify if the issues are already fixed or not on the master branch. Check the Ruby on Rails Guides Guidelines for style and conventions.

If for whatever reason you spot something to fix but cannot patch it yourself, please open an issue.

And last but not least, any kind of discussion regarding Ruby on Rails documentation is very welcome in the rubyonrails-docs mailing list.
