DO NOT READ THIS FILE ON GITHUB, GUIDES ARE PUBLISHED ON https://guides.rubyonrails.org.

Active Record Callbacks

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.

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.

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.

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 < ApplicationRecord
  validates :login, :email, presence: true

  before_validation :ensure_login_has_a_value

  private
    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 < ApplicationRecord
  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 < ApplicationRecord
  before_validation :normalize_name, on: :create

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

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

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

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

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:

Creating an Object

- before_validation
- after_validation
- before_save
- around_save

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

Updating an Object

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

Destroying an Object

- `before_destroy`
- `around_destroy`
- `after_destroy`
- `after_commit` / `after_rollback`

WARNING. `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.

WARNING. Avoid updating or saving attributes in callbacks. For example, don't call `update(attribute: "value")` within a callback. This can alter the state of the model and may result in unexpected side effects during commit. Instead, you can safely assign values directly (for example, `self.attribute = "value"`) in `before_create` / `before_update` or earlier callbacks.

NOTE: `before_destroy` callbacks should be placed before `dependent: :destroy` associations (or use the `prepend: true` option), to ensure they execute before the records are deleted by `dependent: :destroy`.

`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 < ApplicationRecord
  after_initialize do |user|
    puts "You have initialized an object!"
  end

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

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

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

after_touch

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

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

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

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

It can be used along with `belongs_to`:

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

```ruby
class Company < ApplicationRecord
  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
```

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

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

Running Callbacks

The following methods trigger callbacks:

- `create`
- `create!`
- `destroy`
- `destroy!`
- `destroy_all`
- `destroy_by`
- `save`
- `save!`
- `save(validate: false)`
- `toggle!`
- `touch`
- `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.

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

Skipping Callbacks

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

- `decrement!`
- `decrement_counter`
- `delete`
- `delete_all`
- `delete_by`
- `increment!`
- `increment_counter`
- `insert`
- `insert!`
- `insert_all`
- `insert_all!`
- `touch_all`
- `update_column`
- `update_columns`
- `update_all`
- `update_counters`
- `upsert`
- `upsert_all`

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.

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 callback raises an exception, the execution chain gets halted and a ROLLBACK is issued. To intentionally stop a chain use:

```
throw :abort
```

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

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 < ApplicationRecord
  has_many :articles, dependent: :destroy
end

class Article < ApplicationRecord
  after_destroy :log_destroy_action

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

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

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 `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.

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 < ApplicationRecord
  before_save :normalize_card_number, if: :paid_with_card?
end
```

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

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 < ApplicationRecord
  before_save :normalize_card_number,
    if: Proc.new { |order| order.paid_with_card? }
end
```

As the proc is evaluated in the context of the object, it is also possible to write this as:

```
class Order < ApplicationRecord
  before_save :normalize_card_number, if: Proc.new { paid_with_card? }
end
```

Using both :if and :unless

Callbacks can mix both `:if` and `:unless` in the same declaration:

```
class Comment < ApplicationRecord
  before_save :filter_content,
    if: Proc.new { forum.parental_control? },
    unless: Proc.new { author.trusted? }
end
```

Multiple Callback Conditions

The `:if` and `:unless` options also accept an array of procs or method names as symbols:

```
class Comment < ApplicationRecord
  before_save :filter_content,
    if: [:subject_to_parental_control?, :untrusted_author?]
end
```

The callback only runs when all the `:if` conditions and none of the `:unless` conditions are evaluated to `true`.

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 they can be reused.

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 < ApplicationRecord
  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 < ApplicationRecord
  after_destroy PictureFileCallbacks
end
```

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

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 < ApplicationRecord
  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
```

NOTE: 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.

Since using the `after_commit` callback only on create, update, or delete is common, there are aliases for those operations:

- `after_create_commit`
- `after_update_commit`
- `after_destroy_commit`

```
class PictureFile < ApplicationRecord
  after_destroy_commit :delete_picture_file_from_disk

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

WARNING. When a transaction completes, the `after_commit` or `after_rollback` callbacks are called for all models created, updated, or destroyed within that

transaction. However, if an exception is raised within one of these callbacks, the exception will bubble up and any remaining `after_commit` or `after_rollback` methods will *not* be executed. As such, if your callback code could raise an exception, you'll need to rescue it and handle it within the callback in order to allow other callbacks to run.

WARNING. The code executed within `after_commit` or `after_rollback` callbacks is itself not enclosed within a transaction.

WARNING. Using both `after_create_commit` and `after_update_commit` with the same method name will only allow the last callback defined to take effect, as they both internally alias to `after_commit` which overrides previously defined callbacks with the same method name.

```
class User < ApplicationRecord
  after_create_commit :log_user_saved_to_db
  after_update_commit :log_user_saved_to_db

  private
  def log_user_saved_to_db
    puts 'User was saved to database'
  end
end
```

```
irb> @user = User.create # prints nothing

irb> @user.save # updating @user
User was saved to database
```

There is also an alias for using the `after_commit` callback for both create and update together:

- `after_save_commit`

```
class User < ApplicationRecord
  after_save_commit :log_user_saved_to_db

  private
  def log_user_saved_to_db
    puts 'User was saved to database'
  end
end
```

```
irb> @user = User.create # creating a User
User was saved to database

irb> @user.save # updating @user
User was saved to database
```