

Why would you use an after_commit?

Because maybe we want to perform other operations after a record is updated or deleted.
Maybe send notifications, etc.

after_commit -> executed after the save. After the data is persisted.

Polymorphic associations

The model who has the foreign key, should have polymorphic: true.
The other models should have has_many as :imageable

On migrations

Setting up the migration of a polymorphic association

```
class CreatePictures < ActiveRecord::Migration[5.2]
  def change
    create_table :pictures do |t|
      t.string :name
      t.bigint :imageable_id
      t.string :imageable_type
      t.timestamps
    end


    add_index :pictures, [:imageable_type, :imageable_id]
  end
end
```

Self joins

Table that joins itself.

Self joins

- Remember to create the foreign key



```
class Employee < ApplicationRecord
  has_many :subordinates, class_name: "Employee",
                        foreign_key: "manager_id"

  belongs_to :manager, class_name: "Employee", optional: true
end
```

Migration for self joins

```
class CreateEmployees < ActiveRecord::Migration[5.0]
  def change
    create_table :employees do |t|
      t.references :manager
      t.timestamps
    end
  end
end
```

You can specify the through and the foreign key



```
class Author < ApplicationRecord
  has_many :books
end

class Book < ApplicationRecord
  belongs_to :writer, class_name: 'Author', foreign_key:
    'author_id'
end
```

Find

Used to find a value by id

```
# Find the client with primary key (id) 10.
client = Client.find(10)
# => #<Client id: 10, first_name: "Ryan">
```

Take

Grabs a certain quantity or rows.

There is no implicit order.

```
client = Client.take
# => #<Client id: 1, first_name: "Lifo">
```

Take a certain quantity of rows.

first and last

gets the last and first rows by modification date

Order

A note about order

- order may also do the process asc (ascendently) or desc (descendently)

```
Client.order(orders_count: :asc, created_at: :desc)
# OR
Client.order(:orders_count, created_at: :desc)
# OR
Client.order("orders_count ASC, created_at DESC")
# OR
Client.order("orders_count ASC", "created_at DESC")
```

Find by

It is like the “where” but only returns one record

find by

- It's the same than using .where .take

```
Client.find_by first_name: 'Lifo'
# => #<Client id: 1, first_name: "Lifo">

Client.find_by first_name: 'Jon'
# => nil
```

```
Client.where(first_name: 'Lifo').take
```



Want an error?

- Find by also accepts the bang version, which will return the error `ActiveRecord::RecordNotFound` if not found

```
Client.find_by! first_name: 'does not exist'  
# => ActiveRecord::RecordNotFound
```

Will throw an error if the record does not exist.

Batches

Batches

- If you want to consume a lot of memory

```
# This may consume too much memory if the table is big.  
User.all.each do |user|  
  NewsMailer.weekly(user).deliver_now  
end
```

This will consume a lot of time if the table is really large.

This is because you will get all the records into memory.

We use batches to avoid wasting resources from our server

Batches

- For a lot of records
 - Return the records in batches of 1000 automatically

```
User.find_each do |user|  
  NewsMailer.weekly(user).deliver_now  
end
```



It gets the records by 1000 to 1000.

Where

Receives hash arguments and retrieves objects that matches the specific details.

```
Client.where(locked: true)
```

Or

```
development environment (Rails 6.0.5.1)  
User.where(name: "oskar").or(User.where('email LIKE ?', "%mailto%"))
```

Pure queries or ORM methods

Depends on many things like how many instructions or complex the query will be and if my team will accept query strings.

If your team agrees on doing "String queries", they are probably going to be used on queries that are going to be used a lot.

Selecting specific tables

Select only specific fields from the database.

```
Client.select(:viewable_by, :locked)  
# OR  
Client.select("viewable_by, locked")
```


limit and offset



- limit determines the number of records to fetch
- offset determines where the record fetching will start
 - Although gems like will_paginate exist, this is the underlying method of functionality

```
Client.limit(5).offset(30)
```

having



```
Order.select("date(created_at) as ordered_date, sum(price) as  
total_price").  
group("date(created_at)").having("sum(price) > ?", 100)
```

Having used mostly when we have a select and a group.

unscope



- There may be default scopes on your models
- To work around them:

```
Article.where('id > 10').limit(20).order('id  
asc').unscope(:order)
```

Create an default scope

```

22
23 class User < ApplicationRecord
24   default_scope { order(email: :desc) }
25
26   has_many :notifications
27
28   validates :accepts_terms_of_service, acceptance: { message: "Must
29
30
31   after_commit :update_notifications, on: :update
32
33
34   private

```

```

class User < ApplicationRecord
  default_scope { where(active: true) }

```

Scopes

Use them for example when you don't want your queries to get the non active users, or hidden posts.

Default scope -> all query

Custom scope -> a method we can call.

Unscope only works against default scopes.