rails c

github.com/przprz/rails_console

TOC

Create, Read, Update, Delete (CRUD) operations

- comprehensive documentation: https://guides.rubyonrails.org/active_record_querying.html

CRUD: Read

Simplest methods:

`.first`,
`.second`,
`.third`,
`.last` - retrieve 1st, 2nd, and so on

`.take(3)` - retrieve 3 objects

CRUD: Read

`find()`

`.find(id)` - find **one record** by id, may throw AR::RecordNotFound

```
ClaimPayout.find(104980)
```

`.find(id1, id2, id3,)` - find **many records**, also may throw AR::RecordNotFound

`.find([id1, id2, id3])` - same as above but array passed

```
ClaimPayout.find(104980, 1)

ClaimPayout.find(104980, 1, :i_am_not_here)
```

CRUD: Read

`find_by()`

`.find_by(name: 'Andżej')` - find **one record** by some other attribute (may return nil)

```
ClaimPayout.find_by(claim_enquiry_id: 178849,) # hey i'm a comment, and there's a comma after the id
```

Note

It will return the **first** record matching the criteria.

CRUD: Read

Note: all the above methods return *instances* of objects or *arrays of instances*:

```
> ClaimPayout.find_by(claim_enquiry_id: 178849,).class
class ClaimPayout < ApplicationRecord {
                          :id => :integer,
             :claim_enquiry_id => :integer,
                 :collected_at => :datetime,
                   :created_at => :datetime,
                 :legal_entity => :string,
       :selected_payout_option => :string,
                       :token => :string,
                  :updated_at => :datetime
}
```

```
> ClaimPayout.take(3).class
Array < Object
```

It means that they can't be chained. But the methods from next slides can!

CRUD: Read

`where()`

`.where(city: 'Ulaanbaatar')` - finds **many records** (watch out - may return looooooong output)

`.where(continent: 'Africa', hemisphere: 'southern')` - `where` can be used with many attributes

`.where(gender: %w[ L G B T Q H ])` - `where` can be used with array of values

`.where("id < ?", computed_array_of_values(some_params))` - you can **interpolate argument(s)** with `where`

```
ClaimPayout.where      :selected_payout_option => "credit_card_transfer"

ClaimPayout.where(selected_payout_option: ["credit_card_transfer", "free_bank_transfer"])
```

CRUD: Read

`where()` returns *ActiveRecord::Relation* and can be **chained** and **negated**

```
Claim.where("id < ?", 9).class # Claim::ActiveRecord_Relation < ActiveRecord::Relation
```

```
ClaimPayout.where("id < ?", 200).where.not(:selected_payout_option => "credit_card_transfer") # we can negate the predicate
ClaimPayout.where("id < ?", DearComputer.calculate_identifiers(please: true)) # we can pass it a method to generate criteria
ClaimPayout.where("created_at > ?", 3.days.ago) # we can pass dates easily
```

Note

`.not` applies only to the `where` that stands right after it

CRUD: Read

It is often used in `scope`s. Scopes allow to save some commonly used queries.

```ruby
# https://github.com/AirHelp/ah-webapp/blob/d88b8d712ab6654ccf6292d6e1a74c0aad97f6e6/app/models/claim_enquiry_document.rb#L46-L45
class ClaimEnquiryDocument < ApplicationRecord
  # ... some methods
  scope :assignment_form, -> { where(document_type: ASSIGNMENT_FORM) }
  # ... some other methods
end

ClaimEnquiryDocument.assignment_form # find ClaimEnquiryDocument with ASSIGNMENT_FORM as the document_type
```

CRUD: Read

Other useful methods

`.pluck()`

`.pluck(:name)` - get 'name' attribute from all records in a collection

```
ClaimPayout.where("created_at>?", 3.days.ago)
  .pluck(:selected_payout_option).sort.uniq # find which payout options clients used recently
```

`.limit()` - retrieve only a portion of records

```
ClaimPayout.where("created_at>?", 3.days.ago).limit(10)
```

CRUD: Read

`.order()`

`.order(:attribute_name)` is equivalent to `.order(attribute_name: :asc)`

`.order(attribute_name: :desc)`

```
ClaimPayout.where("created_at>?", 3.days.ago).order(collected_at: :desc)
```

CRUD: Read

Querying multiple tables with `.joins()`

- Use case: we need to find the number of claim payouts for the enquiries from web channel.

```
ClaimPayout.joins(:claim_enquiry).where(claim_enquiries: { channel: :ch_web }).count
```

Side note

We can use a symbol for non-spaced strings, e.g. :ch_web instead of 'ch_web'

CRUD: Read

- Use case: we need to find credit card payouts for the enquiries created in the last 2 weeks.

```
ClaimPayout.joins(:claim_enquiry)
  .where("claim_enquiries.created_at >= ?", 2.weeks.ago)
  .where(selected_payout_option: 'credit_card_transfer')
```

Note

- we need to pass singular form to `joins()`, but plural (i.e. the same that the DB uses) to the `where()` method.

CRUD: Read

`to_sql()` - turn the AR query into a SQL query

- Use case: we want to run a SQL query in prod, but we don't speak SQL 😔

```
ClaimPayout.where(selected_payout_option: 'credit_card_transfer').to_sql
=> "SELECT \"claim_payouts\".* FROM \"claim_payouts\" WHERE \"claim_payouts\".\"selected_payout_option\" = 'credit_card_transfer'"
```

CRUD: Read

Finally, we can run arbitrary SQL with

```
sql = "select * from ... your sql query here"
ActiveRecord::Base.connection.execute(sql)
```

CRUD: Create, Update

Update a single record

```
u = User.last
u.update(name: 'Julia')
```

is equivalent to

```
u = User.last
u.first_name = "Julia"
u.save
```

We might want to reload the instance to see the changes:

```
u.reload
```

CRUD: Create, Update

Note

These methods **trigger validations**, and will save the object to the database only if it is valid.

The bang versions (e.g. `update!`) **raise an exception** if the record is invalid.

There are **many other methods for updating** records.

There's a neat cheat-sheet here https://makandracards.com/makandra/42641-different-ways-to-set-attributes-in-activerecord

You can choose more exotic one depending on your use case (like: you need to skip validations, callbacks, etc.)

CRUD: Create, Update

Example: update many records at once, skip validations & callbacks

```
ClaimPayout.where(selected_payout_option: 'credit_card_transfer')
  .where("created_at >= ?", 3.hour.ago)
  .update_all(payoneer_signup_complete: true)
```

Playing around

- in Ruby we can `monkey-patch` - *change the original behaviour*

```ruby
# app/lib/expensive_machine.rb
# The machine that goes *ping*, original version (https://www.youtube.com/watch?v=arCITMfxvEc)
class ExpensiveMachine
  def ping!
    :ping!
  end
end

ExpensiveMachine.new.ping! # => :ping!
```

Playing around

```ruby
# Paste this updated version to the rails console (or irb)
class ExpensiveMachine
  def ping!(patch=false)
    if patch
      return :pong!
    end
    :ping!
  end
end

ExpensiveMachine.new.ping! # => :ping!
ExpensiveMachine.new.ping!('true') # => :pong!
```

Playing around

- we can search for methods

```
ClaimPayout.first.methods.grep /_at$/ # find 'timestamp' methods of the object, like :created_at, :collected_at, etc.
```

- or even display the source code

```
file, line = ClaimPayout.first.method(:currency).source_location
y IO.readlines(file)[line-1, 10]
```

Configuration

- put it in your .irbrc

```
# ~/.irbrc
def ta
  [1, 2, :three, 'four']
end

def ha
  {a: 1, :b => nil, 'three' => false, 4 => 4}
end
```

- example: lets copy `awesome!` method that AirHelp consoles have

We'll need to look for `"awesome_print"` in `dockerfiles` repo

Keyboard shortcuts

Most **readline** shortcuts are supported

TIP: many other applications also support some of these (try in your browser)

- tab - autocompletion
- ctrl+l - `clear` screen # so you don't need to type `clear` anymore
- ctrl+p/ctrl+n - show previous/next command (but it acts weirdly via ssh)
- ctrl+a/ctrl+e - go to beginning/end of line
- alt+b/alt+f - go to previous/next word
- ctrl+b/ctrl+f - go to previous/next character
- ctrl+w/alt+d - delete previous/next word

Keyboard shortcuts

- ctrl+u/ctrl+k - yank text from current position to the beginning/end of line
- ctrl+y - paste yanked text
- ctrl+r - recursive search
- ctrl+j/ctrl+m - insert new line (acts as ENTER key)

Tips

`$ rails console -e production --sandbox` - when you quit `sandbox` session everything is rolled back! #safety

`$ rails console -- --nomultiline` - use `nomultiline` if you need to paste some long code #monkey-patching

`reload!` - loads latest version of code (clears monkey-patches)

`_` - stores result of previous command

`ap Claim.last` - pretty prints

`y Caim.last` - "yaml" prints (display content by serializing it to YAML)

`ActiveRecord::Base.connection.tables` - list all tables in application

Tips

Rails console uses `irb` by default https://github.com/ruby/irb

Try `pry` https://github.com/pry/pry:

- Source code browsing (including core C source with the pry-doc gem), documentation browsing
- Navigation around state (cd, ls and friends)
- Open methods in editors (edit-method Class#method)
- Command shell integration (start editors, run git, and rake from within Pry), gist integration
- Runtime invocation (use Pry as a developer console or debugger)
- Ability to view and replay history, and many, many more...

```
# Gemfile
gem 'pry-rails', :group => :development
```

Links

https://guides.rubyonrails.org/command_line.html#bin-rails-console

https://guides.rubyonrails.org/active_record_validations.html

https://hackernoon.com/meeting-the-query-interface-in-ruby-on-rails-9bu3yec

https://guides.rubyonrails.org/active_record_querying.html

https://pragmaticstudio.com/tutorials/rails-console-shortcuts-tips-tricks

https://medium.com/better-programming/rails-console-magic-tricks-da1fdd657d32

https://www.bounga.org/tips/2018/10/23/rails-console-tips/