ActiveRecord / Sessions

- inverse_of
- Includes
- Join
- Cookies
- Session
- Authentification

Associations bi-directionnelle : Détection automatique

- Rails détecte les associations bi-directionnelle à partir du nom des associations.

```ruby
class Author < ApplicationRecord
  has_many :books
end

class Book < ApplicationRecord
  belongs_to :author
end
```

```
irb> author = Author.first
irb> book = authors.books.first
irb> author.first_name == book.first.author.first_name # => true
irb> author.first_name = 'David'
irb> author.first_name == book.author.first_name # => true ; author.object_id == book.author.object_id
```

- Active Record ne charge qu'une seule copie de l'objet Author, ça évite une requête à la base de donnée, et évite les données incohérentes.

Associations bi-directionnelle : Détection automatique impossible

- Rails ne parvient pas à determiner les associations inverses quand on utilise `:through`, `:foreign_key`, `:order`,

```ruby
class Author < ApplicationRecord
  has_many :books, inverse_of: 'writer'
end

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

```
irb> author = Author.first
irb> book = authors.books.first
irb> author.first_name == book.first.author.first_name # => true
irb> author.first_name = 'David'
irb> author.first_name == book.author.first_name # => false ; author.object_id != book.author.object_id
```

Associations bi-directionnelle : inverse_of

- `inverse_of` permet d'indiquer quelle est la relation inverse d'un `belongs_to`, `has_one`, `has_many`

```ruby
class Author < ApplicationRecord
  has_many :books, inverse_of: 'writer'
end

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

```
irb> author = Author.first
irb> book = authors.books.first
irb> author.first_name == book.first.author.first_name # => true
irb> author.first_name = 'David'
irb> author.first_name == book.author.first_name # => true ; author.object_id == book.author.object_id
```

Includes : eager loader les données

- Eviter le N+1

```
<% Message.limit(25).each do |message| %>
<h1>
    <%= message.title %>
    <small><%= message.author.name %></small>
</h1>
<% end %>

# (0.2ms) SELECT "messages".* FROM "messages"
# (0.2ms) SELECT "authors".* FROM "authors" WHERE "author"."id" = ?
# (0.2ms) SELECT "authors".* FROM "authors" WHERE "author"."id" = ?
# (0.2ms) SELECT "authors".* FROM "authors" WHERE "author"."id" = ?
....
# => 26 requêtes
```

Includes : eager loader les données

- Toutes les associations spécifiées sont chargées en utilisant le nombre minimum de requêtes possible.

```
<% Message.includes(:author).limit(10).each do |message| %>
<h1>
    <%= message.title %>
    <small><%= message.author.name %></small>
</h1>
<% end %>

# (0.2ms) SELECT "messages".* FROM "messages"
# (0.3ms) SELECT "authors".* FROM "authors" WHERE "authors"."id" IN (?, ?, ..)

# => 2 requêtes
```

Includes : eager loader les associations

- On peut spécifier une liste d'associations

```
Message.includes(:author, :comments)
```

- Et des associations imbriquées

```
Customer.includes(orders: {books: [:supplier, :author]}).find(1)
```

Join : INNER JOIN

- Jointure à partir d'une ou plusieurs associations

```
Book.joins(:comments) # Les livres ayant des commentaires
# SELECT books.* FROM books
#  INNER JOIN comments ON comments.book_id = books.id
```

- Récupère un livre pour chaque livre avec un commentaire
    - Attention aux doublons, pour les éviter : `Book.joins(:comments).distinct`
- Jointure à partir de plusieurs associations

```
Book.joins(:author, :comments) # Les livres avec leur auteur qui ont au moins un commentaire
# SELECT books.* FROM books
#  INNER JOIN authors ON authors.id = books.author_id
#  INNER JOIN comments ON comments.book_id = books.id
```

Join : INNER JOIN

- Jointures imbriquées

```
Book.joins(comments: :customer) # Les livres qui ont un commentaire par un client
# SELECT books.* FROM books
#   INNER JOIN reviews ON reviews.book_id = books.id
#   INNER JOIN customers ON customers.id = reviews.customer_id
```

- Spécifier des conditions

```
Book.joins(:comments).where('comments.created_at' => (1.week.ago..Time.now).distinct

# Les livres uniques avec des commentaires datant de moins d'une semaine.

Book.joins(:author).where('author.first_name' => "Nabilla" ).distinct

# Les livres uniques avec un auteur qui se prénomme Nabilla.
```

Cookies

- Les cookies permettent de stocker des informations sur le navigateur de l'utilisateur.
- Limité à une petite quantité (4ko) de texte
- Ils sont stockés en clair sur la navigateur et facilement accessibles, recopiables, modifiables
    - On ne stocke donc dedans aucune donnée sensible !
- Ils ont une date d'expiration (par défaut à la fin de la session du navigateur)
- On peut aussi y accéder en Javascript (sauf si on l'interdit explicitement)

Cookies - Header HTTP Set-cookie

- Si le serveur souhaite créer ou modifier des cookies, ils sont envoyés dans le header `Set-Cookie` de la réponse HTTP.

```
HTTP/2.0 200 OK
Content-Type: text/html
Set-Cookie: delicieux_cookie=choco
Set-Cookie: savoureux_cookie=menthe

[contenu de la page]
```

- Peut spécifier la date d'expiration (`Expires`), restreindre l'accès (`Secure`, `HttpOnly`), et définir où les cookies sont envoyés (`Domain`, `Path`, `SameSite`)

```
Set-Cookie: savoureux_cookie=menthe; Expires=Mon, 27 Mar 2050 07:28:00 GMT; Secure; HttpOnly
```

Cookies - Header HTTP Cookie

- Pour toutes requêtes suivantes, le client envoie tous les cookies enregistrés (qui ont le droit d'être envoyés), dans le header `Cookie` de la requête HTTP.

```
GET /page_exemple.html HTTP/2.0
Host: www.example.org
Cookie: delicieux_cookie=choco; savoureux_cookie=menthe
```

- Dans Rails, le module `ActionController#cookies` permet de lire et écrire les cookies HTTP.

Cookies - ActionController#cookies - Écrire un cookie

- Écrire un cookie basique

```ruby
cookies[:user_name] = 'david'

# On doit sérialiser "à la main" les données
cookies[:lat_lon] = JSON.generate([47.68, -122.37])
```

- Préciser une date d'expiration

```ruby
cookies[:login] = { value: "XJ-122", expires: Time.utc(2020, 10, 15, 5) }

cookies[:login] = { value: "XJ-122", expires: 1.hour }

# Définir un cookie 'permanent', qui expire dans 20 ans
cookies.permanent[:login] = "XJ-122"
```

Cookies - ActionController#cookies - Écrire un cookie (suite)

- Sécuriser les cookies

```
# Définir un cookie signé, qui empêche la modification de la valeur du cookie
cookies.signed[:user_id] = current_user.id

# Définir un cookie chiffré, qui empêche la modification et la lecture de sa valeur
cookies.encrypted[:discount] = 45

# Définir un cookie chiffré, qui expire dans 20 ans
cookies.signed.permanent[:login] = 'XJ-122'
```

- Supprimer un cookie

```
cookies.delete :user_name
```

Cookies - ActionController#cookies

- `cookies[:name]` permet de lire le cookie

```
cookies[:user_name] # => "david"
cookies[:login]     # => "XJ-122"

# On doit sérialiser les données le cas échéant
JSON.parse(cookies[:lat_lon]) # => [47.68, -122.37]

# Lire des cookies sécurisés
# encrypted et signed retournent nil si le cookie a été modifié
cookies.encrypted[:discount]  # => 45
cookies.signed[:login]        # => "XJ-122"

# Pour connaître le nombre de cookies
cookies.size                  # => 2
```

Sessions: HTTP : Protocole stateless

- HTTP est un protocole sans état, c'est à dire que le serveur oubli le client entre chaque requête.
- Chaque requête est indépendante, et tous les clients sont traités de la même manière.
- Plus simple pour le serveur, mais peu rendre compliqué et lourd le mécanisme d'identification
- Plutôt utilisé par les APIs
- Un exemple d'authentification stateless: La `basic authentication` - On fournit un header spécifique
  - `Authorization: Basic <credentials>`
  - Où `<credentials>` = base64(<username>:<password>)

Sessions: HTTP : Authentification stateless

```ruby
class ApplicationController < ActionController::Base
  before_action :authenticate_http_basic

  def authenticate_http_basic
    # Helper de rails pour parser le header Authorization en mode basic
    authenticate_with_http_basic do |username, password|
      # Gestion des passwords chiffrés
      user = User.find_by(username: username)
      @current_user = user.authenticate(password) if user
    end
  end
end
```

Sessions : HTTP "stateful"

- Permet d'ajouter du "stateful" à l'application, en fournissant une session par utilisateur pour y lire et y stocker des données.
- Lors d'une requête, si le serveur ne reçoit pas d’identifiant de session, il va en créer un et va le transmettre dans sa réponse dans un cookie.
- Le client va donc transmettre ensuite cet identifiant de session à chaque requête.
- Par défaut Rails stocke toute la session (id et contenu) sur le client dans un cookie chiffré et ne nécessite aucune configuration.
    - On stockera en général peu de choses en session. (ce n'est pas utile et on est limité par la taille max du cookie)
    - On peut modifier ce comportement et stocker les sessions en cache, en base de donnée, ou via memcached (mais cela a ses inconvénients).

Sessions : Accès

- Écriture

```ruby
session[:user_id] = 5
session[:last_activity] = Time.zone.now
session[:active_filters] = {
  department: :all,
  groups: [:web1, :web2, :web3],
}
```

- Lecture

```ruby
current_user = User.find(session[:user_id])
session[:last_activity].class # => ActiveSupport::TimeWithZone
session[:active_filters][:groups].include?(:web3)  # => true
```

- Rails serialize automatiquement les données à la lecture et l'écriture

Sessions : Reset

- `reset_session` supprime la session actuelle et en créé une nouvelle vide

```ruby
class UserSessionsController < ActionController::Base
  def logout
    reset_session
    @current_user = nil
  end
end
```

Sessions : Authentification stateful - Login

```ruby
class UserSessionsController < ApplicationController
  skip_before_action :check_login # slide suivante

  def create
    if params[:username] && params[:password]
      user = User.find_by(username: params[:username])
      if user && user.authenticate(params[:password])
        # La session est disponible partout
        session[:current_user] = user
        redirect_to root_url
        return
      end
    end

    redirect_to(login_url)
  end
end
```

Sessions : Retrouver un utilisateur logué

```ruby
class ApplicationController < ActionController::Base
  before_action :check_login

  def check_login
    if session[:current_user]
      @current_user = session[:current_user]
    else
      redirect_to(login_url)
    end
  end
end
```

Correction TP

- Strong Params
- CRUD
- Enums
- Migration en 2 (ou 3 temps)
- Gérer les foreign keys
- Filtrer des queries

Strong Parameters

- Permettent de valider des inputs
- Rails auto-wrap les params dans une clef qui correspond à la resource du contrôleur
  Exemple: `CreaturesController`, rails wrappe tout dans `creature`.

```ruby
class CreaturesController
  private
  def create_params
    permitted = params.require(:creature).permit(:name)
    permitted.require(:name) # rend le nom obligatoire
    permitted
  end
end
```

On peut forcer ou autoriser la présence d'un paramètre.

CRUD + L

```ruby
class CreaturesController < ApplicationController
  # route: get '/creatures', to: 'creatures#index'
  def index # liste les créatures
  end
  # route: get '/creatures/:id', to: 'creatures#show'
  def show # affiche une créature
  end
  # route: post '/creatures', to: 'creatures#create'
  def create # créer une créature
  end
  # route: put '/creatures/:id', to: 'creatures#update'
  def update  # mettre à jour une créature
  end
  # route: delete '/creatures', to: 'creatures#destroy'
  def destroy  # supprimer une créature
  end
end
```

Enumération

- Concept ActiveRecord pour gérer des flags facilement, basé sur des colonnes `integer`

```ruby
class AddSizeColumnToCreatures < ActiveRecord::Migration[7.0]
  def change
    add_column :creatures, :size, :integer
  end
end
```

```ruby
class Creature < ApplicationRecord
  enum :size, [:small, :big, :giant]
end
```

```ruby
creature = Creature.new(name: "Big Chungus", health_points: 42, size: :big)
```

Migrations en plusieurs phases

Si on veut ajouter un nouveau champ en base de données :

1. On met à jour le schéma pour ajouter le champ (1 migration)
2. On met à jour la partie du code qui affecte le champ (1 update de code)
3. On met à jour les records de la base qui n'ont pas le champ (1 migration)
4. On met à jour le schéma pour forcer la présence du champ (1 migration optionnelle)
5. On affiche et on utilise le champ pour les raisons métier (# updates de code)

Dans le TP :

Ajout de `size` -> Ajout du callback pour setter la `size` -> Update des records en DB -> Affichage de la `size`

Ajouter des Foreign Keys

ActiveRecord vient avec le concept de référence qui simplifie. Il ajoute :

- Le champ pour l'id (ex: `left_fighter_id`)
- L'index pour le champ id (ex: `index_on_left_fighter_id`)
- La Foreign Key pour le champ en question pour lier les tables
- Si le champ référence ne s'appelle pas comme une table (ex: `creature_id`), il faut préciser la table.

```ruby
create_table :combats do |t|
  # ...
  t.references :left_fighter, null: false, foreign_key: { to_table: 'creatures' }
  t.references :right_fighter, null: false, foreign_key: { to_table: 'creatures' }
  t.references :winner, null: true, foreign_key: { to_table: 'creatures' }
end
```

Ajouter des Foreign Keys

On oubliera pas d'ajouter les relations sur les modèles :

```ruby
class Combat < ApplicationRecord
  enum :result, [:draw, :domination]

  belongs_to :left_fighter, class_name: 'Creature'
  belongs_to :right_fighter, class_name: 'Creature'
  # belongs_to rend la relation obligatoire, ici c'est facultatif
  belongs_to :winner, class_name: 'Creature', optional: true
end
```

Utilisation des Foreign Keys

On évitera globalement de manipuler des identifiants et on utilisera des records.

```ruby
def baston!
    return if left_fighter.nil? || right_fighter.nil?

    left_hp = left_fighter.health_points
    right_hp = right_fighter.health_points
    left_fighter.health_points -= right_hp
    right_fighter.health_points -= right_hp

    if left_fighter.alive?
        self.winner = left_fighter
        self.result = :domination
    elsif right_fighter.alive?
        self.winner = right_fighter
        self.result = :domination
    else
        self.result = :draw
    end
end
```

Ne pas oublier de sauvegarder les records liés

- C'est différent de la création automatique des records liés

```ruby
def create
  left_fighter = Creature.find(params[:left_fighter_id])
  right_fighter = Creature.find(params[:right_fighter_id])

  @combat = Combat.new(create_params)
  @combat.left_fighter = left_fighter
  @combat.right_fighter = right_fighter
  @combat.baston! # baston met à jour les créatures
  @combat.left_fighter.save! # il faut save pour persister la perte de pv
  @combat.right_fighter.save!
  @combat.save!

  render json: @combat.as_json(include: COMBATS_RENDER_CONFIG)
rescue ActiveRecord::RecordNotFound
  render json: {}, status: 404
end
```

Filtrer des relations à la volée

```ruby
def index
  @combats = Combat.all

  # on supporte uniquement les réels résultats
  # possibles qui sont listés sur le modèle Combat
  if Combat.results.include?(params[:result])
    @combats = @combats.where(result: params[:result])
  end

  # Si on fournit une query, on l'utilise pour la requête like
  if params.include?(:query)
    @combats = @combats.where("name LIKE ?", "%#{params[:query]}%")
  end

  render json: @combats.as_json(include: COMBATS_RENDER_CONFIG)
end
```