Active Record

Historique

ActiveRecord de Rails

Les migrations Active Record

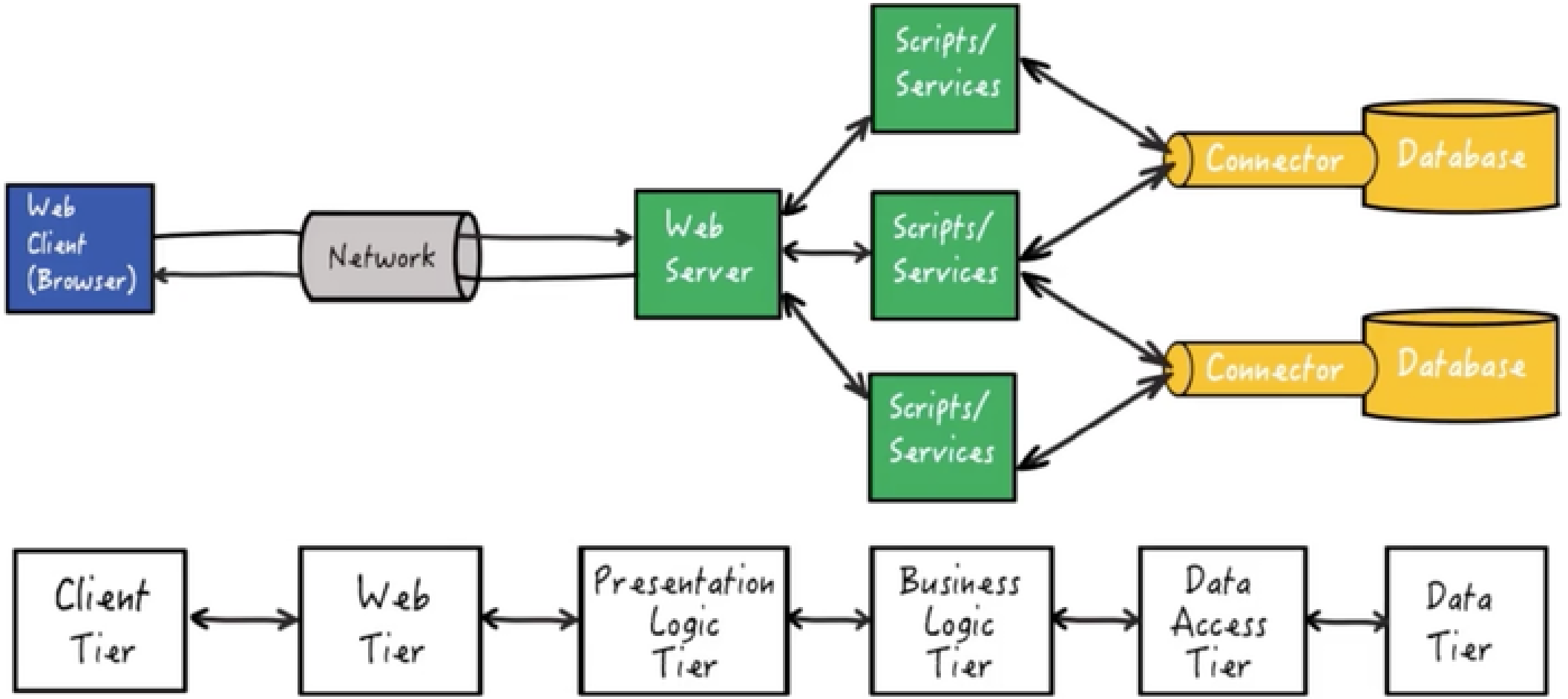
Schema

CRUD

Associations

Scoping

ActiveRecord: an architectural design pattern



Pattern ActiveRecord : Historique

- Nommé en 2003 par Martin Fowler (+ Super fort en code / - hot takes bof)
- En PHP, les ORM Propel et Eloquent utilisent le pattern ActiveRecord.
- Utilisé pour faire du CRUD de façon indépendante à la DB (Pgsql, mysql, sqlite, Oracle, ...)
- Construit un cadre pour utiliser des RDBMS en OOP

Programmation Objet	BDD Relationnelle
Classe	Table
Instance / Objet	Record (ligne)
Attributs	Record Values (colonnes)

ActiveRecord de Rails

- La génération de requètes SQL optimisées (même complexes avec Arel)
- Les validations de modèles :
  - Validation des assignations aux attributs
  - Gestion des contraintes d'intégrités
  - Utilisation d'index avec contraintes
- Les associations entre les tables via les foreign keys :
  - Une table `blog_posts` disposant d'une FK `author_id` vers la table `users` permettra : `blog_post.author` et `user.blog_posts`.
- L'évolution de la structure de la base et des données au fil du temps (migrations)

Les migrations : Principe

- Permet de faire évoluer sa base de données au fil du temps
- Les migrations sont stockées dans le répertoire `db/migrate`, le nom du fichier est **obligatoirement** de la forme `AAAAMMJJHHMMSS_create_products.rb`
    - ⚠️ L'horodatage permet à Rails de déterminer quelle migration doit être exécutée et dans quel ordre, et le nom du fichier correspond au nom de la classe de migration. (`CreateProducts` dans cet exemple)
- Chaque migration est une nouvelle "version" de la base de données
    - Elle ajoute, supprime, et/ou modifie des tables, des colonnnes, et/ou des enregistrements.
- Chaque migration est réversible (en théorie)
- Rails sauvegarde la version de la migration actuelle

Les migrations Active Record : Utilisation

- `bin/rails db:migrate` : lance les migrations non encore appliquées sur la base de données.
- `bin/rails db:rollback` : annule (revert) une migration (effectue l'opération inverse).
- `bin/rails db:rollback STEP=3` : revert les 3 dernières migrations
- `bin/rails db:migrate:redo` : revert et exécute à nouveau la dernière migration (accepte le paramètre `STEP`)
- `bin/rails db:migrate VERSION=20080906120000` : lance ou annule les migrations jusqu'à la migration d'une version souhaitée (via l'horodatage des fichiers)

Les migrations Active Record : Exemple

```ruby
class CreateProducts < ActiveRecord::Migration[7.0]
  def change
    create_table :products do |t|  # créé une table "products"
      t.string :name               # créé un colonne 'name' de type :name
      t.text :description          # créé un colonne 'description' de type :text

      t.timestamps                 # créé les colonnes 'created_at' et 'updated_at'
    end
  end
end
```

- La création de la colonne (clé primaire) `id` est implicite (par défaut)
- Si la base de donnée le supporte, les migrations sont exécutées dans une transaction, pour être annulée si elle échoue. (Atomicité)
- Dans la méthode `change`, on définit ce que la migration doit faire quand elle est executée, Rails détermine comment effectuer l'opération inverse.

Les migrations Active Record : définir un rollback (reversible)

- On peut spécifier à Rails quoi faire pour revert une migration, quand cela ne pas peut être déteminé automatiquement.
- On utilse la méthode `reversible`

```
class ChangeProductsPrice < ActiveRecord::Migration[7.0]
  def change
    reversible do |dir|
      change_table :products do |t|
        dir.up   { t.change :price, :string }
        dir.down { t.change :price, :integer }
      end
    end
  end
end
```

Les migrations Active Record : définir un rollback (up/down)

- On peut définir aussi définir deux methodes `up` et `down`.

```ruby
class ChangeProductsPrice < ActiveRecord::Migration[7.0]
  def up
    change_column :products, :price, :string
  end

  def down
    change_column :products, :price, :integer
  end
end
```

- Cette version est pratique si on fait une migration des données en plus du schéma.

Les migrations Active Record : Quelques méthodes de la doc

```
create_table            # Créer une table
drop_table              # Détruire une table
rename_table            # Renommer une table
add_column              # Ajouter une colonne à une table
remove_column           # Supprimer une colonne à une table
rename_column           # Renommer une colonne d'une table
change_column           # Changer (type, nom, contraintes) d'une colonne
add_index               # Ajouter un index à une table
remove_index            # Supprimer un index d'une table
rename_index            # Renommer l'index d'une table
add_reference           # Ajouter une FK à une table
remove_reference        # Supprimer une FK d'une table
add_timestamps          # Ajouter un created_at/updated_at à une table
remove_timestamps       # Enlever les timestamps
enable_extension        # Activer un plugin de DB
disable_extension       # Désactiver un plugin de DB
```

Générateur de migration Active Record

- Le générateur Rails permet de créer des migrations (utilisé par le générateur de modèle).
- `bin/rails generate migration NomDeLaMigration` créé une migration `NomDeLaMigration` dans un fichier horodaté.
- Des conventions de nommages permettent d'indiquer quel type de modification on souhaite réaliser sur la base et de créer le squelette de migration correspondant.
  - `CreateModel` : Ajouter une table
  - `AddColumnsToModels` : Ajouter des colonnes
  - `RemoveColumnsToModels` : Enlever des colonnes
  - `AddModelRefToOtherModel` : Ajouter une Foreign Key
  - ...

Générateur de migration Active Record : Création d'une table

- `bin/rails generate migration CreateCompany`

```ruby
# db/migrate/20230312110000_create_company.rb :
class CreateCompany < ActiveRecord::Migration[7.0]
  def change
    create_table :companies do |t|

      t.timestamps
    end
  end
end
```

- ⚠️ Pluriel ! : `CreateCompany` pour créer un table "companies"
- On peut préciser des champs à créer lors de la création de la table : `bin/rails generate migration CreateCompany name:string creatures_limit:integer`

Générateur de migration Active Record : Ajout de colonnes

- `bin/rails generate migration AddNameToUsers first_name:string last_name:string`

```ruby
#  db/migrate/20230312110218_add_name_to_users.rb :

class AddNameToUsers < ActiveRecord::Migration[7.0]
  def change
    add_column :users, :first_name, :string
    add_column :users, :last_name, :string
  end
end
```

- `bin/rails generate migration RemoveFirstNameToUsers first_name:string` pour supprimer des colonnes

Active Record : Types

- Rails propose des types standard pour les colonnes, suivant l'usage de la colonne
- Agnosticité par rapport à la base
- Ils sont retranscrits par l'adapteur de base de données dans le type correspondant

Type ActiveRecord	Utilisation	MySQL	postgre
:string	Text court	varchar(255)	varchar(255)
:text	Texte long	text	text
:integer	Nombre entier	int(4)	integer(4)
:float	Nombre à double précision	float(24)	float
:decimal	Nombre à grande précision	decimal	decimal

Active Record : Types (suite)

Type	Utilisation	MySQL	postgre
:datetime	Date et heure	datetime	timestamptz (configurable)
:time	Heure	time	time
:date	Date	date	date
:boolean	Boolean (true/false)	tinyint(1)	boolean
:binary	Données binaires	blob	bitea
:json	Donnée au format JSON	json	json
:jsonb	Donnée au format JSONB	*inexistant*	jsonb
:uuid	Identifiant unique	*inexistant*	uuid

⚠️ Liste non exhaustive.

Active Record : Schema

- Au fur et à mesure des migrations, Active Record met à jour le fichier `db/schema.rb`.
- Ce fichier contient le schéma actuel de la base de données.
- Par défaut en ruby, il peut être généré en SQL (via le paramètre `config.active_record.schema_format` dans `config/application.rb`).
- `bin/rails db:migrate` génère un nouveau schema.
- Plutôt que de dérouler la liste des migrations, on peut appliquer directement le schéma en executant `bin/rails db:schema:load`.
- Charger le schéma est utile pour :
  - les vieux projets dont les migrations ne fonctionnent plus
  - quand il y a beaucoup de trop de migrations et que c'est long 😴

Active Record : Schema (Exemple)

```ruby
# db/schema.rb

ActiveRecord::Schema[7.0].define(version: 2023_03_07_203950) do
  create_table "users", force: :cascade do |t|
    t.string "email", null: false
    t.string "crypted_password"
    t.string "salt"
    t.datetime "created_at", null: false
    t.datetime "updated_at", null: false
    t.index ["email"], name: "index_users_on_email", unique: true
  end

  create_table "colors", force: :cascade do |t|
    t.jsonb "color"
    t.string "html_color"
    t.boolean "available", default: true
    t.datetime "created_at", null: false
    t.datetime "updated_at", null: false
  end
end
```

Query des records

- Il existe des méthodes de classes pour récupérer un record :

```ruby
User.find_by(email: 'admin@fbi.gov', role: :admin)
User.first
```

- Et d'autres méthodes permettent de récupérer des relations / queries :

```ruby
User.where(role: :admin).limit(3).class # => User::ActiveRecord_Relation
```

- On peut faire plein de choses avec une relation :

```ruby
User.where("age < ?", 18).update_all(status: :underaged) # Update
User.where(last_name: 'Musk').destroy_all # Suppression
User.where(role: :admin).exists? # Existance
User.all.to_a # Récupérer les records
```

Insérer des records

- Utiliser une méthode de classe :

```
user = User.create(name: "Cheng Xin")
```

- Instancier un record, l'éditer et le sauvegarder :

```
user = User.new
user.name = "Cheng Xin"
user.save
```

- Utiliser un bloc

```
user = User.create do |new_user| # marche avec #new aussi
  new_user.name = "Cheng Xin"
end
```

Mettre à jour des records

- Mettre à jour à partir d'une relation

```ruby
User.where("age < ?", 18).update_all(status: :underaged) # Update global sur condition
```

- Mettre à jour un record en particulier

```ruby
user = User.find_by(name: 'Jexx')
user.name = 'Jeff'
user.save # La requête s'effectue ici
```

- On peut aussi faire un update directement

```ruby
user = User.find_by(name: 'Jeff')
user.update(name: 'Unethical Bezos')
```

Supprimer des records

- On peut supprimer des records à partir d'une relation

```
User.where('age < ?', 13).destroy_all # la législation est trop difficile
```

- Attention à ne pas utiliser sans contraintes!

```
User.destroy_all # Adios la base, faut espérer qu'il y a un backup
```

- On peut aussi faire une suppression directement

```
user = User.find_by(name: 'Cicada 3301')
user.destroy
```

Exemple de gestion de messages

Imaginons les tables suivantes :

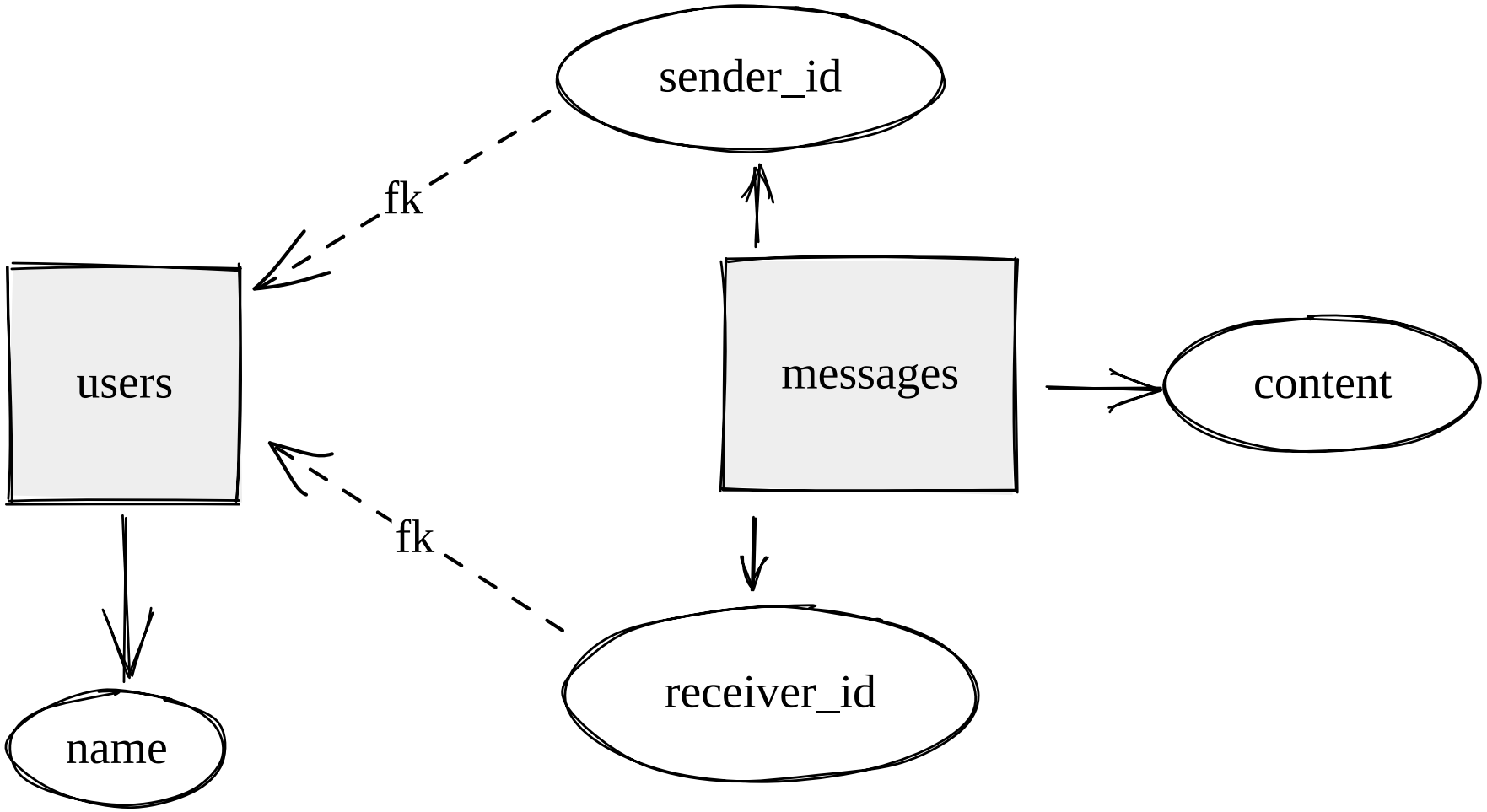


Schéma de l'exemple

```
create_table "messages", force: :cascade do |t|
  t.string "content"
  t.integer "sender_id", null: false
  t.integer "receiver_id", null: false
  t.datetime "created_at", null: false
  t.datetime "updated_at", null: false
  t.index ["receiver_id"], name: "index_messages_on_receiver_id"
  t.index ["sender_id"], name: "index_messages_on_sender_id"
end

create_table "users", force: :cascade do |t|
  t.string "name"
  t.datetime "created_at", null: false
  t.datetime "updated_at", null: false
end

add_foreign_key "messages", "users", column: "receiver_id"
add_foreign_key "messages", "users", column: "sender_id"
```

Associations dans les modèles

ActiveRecord permet d'expliciter les associations entre ses modèles pour avoir des relations (queries) sur les modèles :

```
# Depuis l'utilisateur, des relations (queries) :
user = User.find_by(name: "Jeff")
user.sent_messages              # une query qui définit les messages envoyés
user.received_messages          # pareil mais pour les messages reçus
user.sent_messages_to           # pareil avec les users à qui celui-ci a écrit
user.received_messages_from     # pareil mais pour les users qui lui ont écrit

# Depuis le message, des relations unitaires :
message = Message.last
message.sender    # un User
message.receiver  # un User
```

Expliciter les associations sur les modèles

- Sur le modèle `Message`, on dit qu'il appartient à un user `sender` et un autre user `receiver` :

```ruby
class Message < ApplicationRecord
  belongs_to :sender, class_name: 'User'
  belongs_to :receiver, class_name: 'User'
end
```

- Sur le modèle `User`, on va pouvoir définir les associations qu'on veut :

```ruby
class User < ApplicationRecord
  has_many :sent_messages, class_name: 'Message', foreign_key: :sender_id
  has_many :received_messages, class_name: 'Message', foreign_key: :receiver_id

  has_many :sent_messages_to, through: :sent_messages, source: :receiver
  has_many :received_messages_from, through: :received_messages, source: :sender
end
```

Exemple plus simple

- Si on a un modèle `Author` et un autre `Book` et qu'on laisse ActiveRecord utiliser les paramètres par défaut :

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

class Book < ApplicationRecord
  belongs_to :author
end
```

- Ça s'utilise très facilement aussi :

```ruby
author = Author.find_by(name: 'Liu Cixin')
# Book.new(name: 'Dark Forest', author: author)
book = Book.new(name: 'Dark Forest', author:) # depuis ruby 3.1
book.save!
```

Pour faire des relations

Il existe des tonnes de paramètres pour construire les relations et requêtes correspondant à vos use-cases :

- `has_one` : Il existe UNE ligne avec une FK pointant vers moi
- `has_many` : Il existe plusieurs lignes avec une FK pointant vers moi
- `belongs_to` : J'ai une FK pointant vers un record
- `has_and_belongs_to_many` : * - * (many-to-many)
  - faisable avec `has_many through:` et un modèle intermédiaire
- Polymorphic relationships : Quand la FK d'un modèle peut pointer sur plusieurs types de records
- La documentation est très complète : `google active record associations`

Scoping

On peut voir ça comme des requêtes pré-enregistrées sur le modèle :

```
class Book < ApplicationRecord
  scope :old, -> { where('year_published < ?', 50.years.ago )}
  scope :old_and_expensive, -> { old.where('price > 500') }
end
# ...
Book.old_and_expensive.where(author: 'Shakespeare').to_a
```

- S'utilise comme une relation normale
- Permet d'ajouter facilement des comportements à son application

Exemple de scoping

Une solution si on veut éviter qu'un utilisateur se fasse passer pour un ancien utilisateur :

- Ajouter un scope aux utilisateurs :

```
class User < ApplicationRecord
  scope :enabled, -> { where(disabled: false) }
end
```

- Quand on utilise le modèle User pour des queries, on utilise toujours le scope `enabled` :

```
class UsersController < ApplicationController
  def show
    @user = User.enabled.find(params[:user_id])
  end
end
```

Exemple de scoping (suite)

- Quand on supprime l'utilisateur, on n'enlève pas son record de la DB, mais on le désactive.

```
class UsersController < ApplicationController
  def destroy
    User.enabled.find(params[:user_id]).update(disabled: true)
  end
end
```

Résultat :

- On ne peut effectuer des requêtes que sur des utilisateurs non désactivés
- On lève une 404 quand on essaie d'accéder à un utilisateur désactivé
- Exemple : Ça permet de garder des documents ajoutés par l'utilisateur sans casser les FK