

ActiveRecord

- Validations
- Callbacks des modèles
- Callbacks des contrôleurs



Petit point : Sauvegarde d'éléments liés

- ActiveRecord sauvegarde automatiquement les éléments liés par FK

```
author = Author.new(name: 'Liu Cixin')
book = Book.new(name: 'Problème à trois corps', author: author)
book.save # le livre et l'auteur sont sauvegardés dans le bon ordre des FK
```

Est-ce que l'auteur sera inséré si l'insertion du livre échoue ?

- ActiveRecord permet de faire des transactions pour ajouter de l'atomicité (TOR) :

```
ActiveRecord::Base.transaction do
  david_account.withdrawal(100)
  mary_account.deposit(100)
end
```

- `ActiveRecord::Base.transaction` lève une exception si la transaction échoue - donc penser au retry si nécessaire

Validations

- S'assurer qu'uniquement des données correctes sont sauvées

Exemple : Un utilisateur a forcément un e-mail

- Eviter que le code fasse des fausses suppositions

Exemple: Erreur 500: `send_mail(user.email)` - L'utilisateur n'avait pas d'e-mail

- Implémenter des règles métier

Exemple : Un compte bancaire ne peut pas avoir un solde < 0

- 2 types de validations :

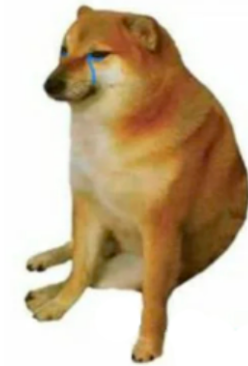
- Par la base de données en ajoutant des contraintes d'intégrité
- Par le client, qui se contraint à respecter des règles

2 types de discussions

client: insert(checked_data)



RDBMS: Oui...

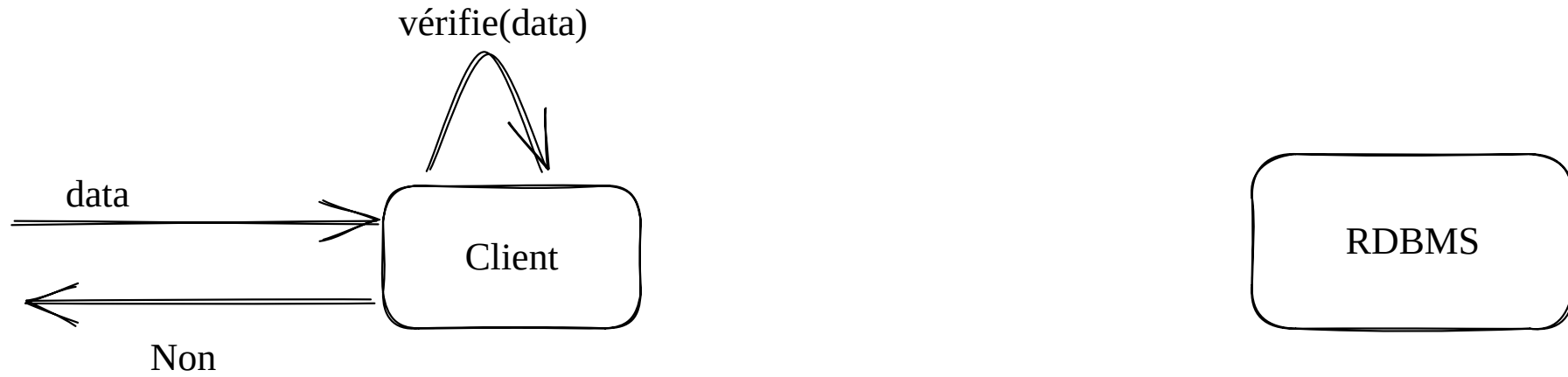


client:
insert(unchecked_data)



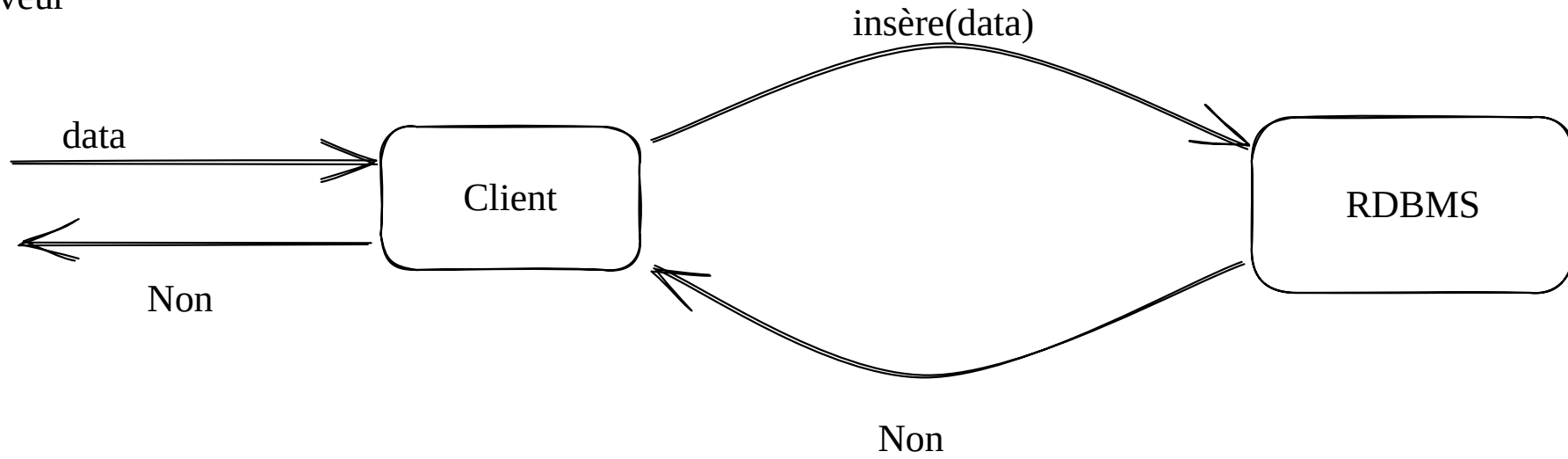
RDBMS: No

2 types de validations



Côté client

Côté serveur



Validation de contraintes d'intégrité

- On évite d'utiliser les contraintes d'intégrité de la base

```
ALTER TABLE -- Exemple de contrainte en SQL
  accounts
ADD CONSTRAINT
  positive_balance CHECK (balance > 0);
```

Database constraints and/or stored procedures make the validation mechanisms *database-dependent* and can make testing and maintenance more difficult.

However... database-level validations can safely handle some things (such as uniqueness in heavily-used tables) that can be difficult to implement otherwise....

[but] it's the opinion of the Rails team that model-level validations are the most appropriate in most circumstances. - DHH

Validations ActiveRecord : Concepts

- L'instance de modèle effectue ses propres validations en fonction de règles.

```
class Creature < ApplicationRecord
  validates :name, presence: true
end
```

- Les vérifications peuvent être demandées manuellement :

```
Creature.new(name: "Olivier 50 pompes").valid? # => true
Creature.new(name: nil).valid? # => false
Creature.new(name: " ").valid? # => false
Creature.new(name: "Bob").validate # => false
```

Validations ActiveRecord : Validation automatique

- Mais sont aussi appelées automatiquement par `create` , `save` et `update`
- Cela empêche la sauvegarde d'un objet invalide

```
creature = Creature.create(name: nil) # => #<Creature:0x... id: nil, name: "" ...>
creature.persisted? # => false

creature = Creature.new(name: nil) # => #<Creature:0x... id: nil, name: "" ...>
creature.save # => false
creature.persisted? # => false

creature = Creature.find_by(name: "Gork") # => #<Creature:0x... id: 3, name: "Gork" ...>
creature.update(name: " ") # => false
creature.name # => "Gork"
```


Validations ActiveRecord : Active Model Errors

- `#errors` retourne un objet de type `ActiveModel::Errors` permettant d'accéder aux erreurs de validations.
- Il contient les erreurs après l'appel manuel ou automatique des validations.

```
creature = Creature.new(name: "")
creature.errors.any? # => false
creature.validate
creature.errors.any? # => true
creature.errors.full_messages # => ["Name can't be blank"]
```

- `errors[:attribut]` permet d'obtenir les erreurs sur un attribut précis

```
creature = creature.update(name: nil)
creature.errors[:name].any? # => true
creature.errors[:name] # => ["can't be blank"]
```

Validations ActiveRecord : Gestion d'erreur

- Gérer des nils partout est pénible, on va plutôt lever des exceptions :

```
Creature.new(name: nil).validate! # => lève l'exception:  
# ActiveRecord::RecordInvalid - Validation failed: Name can't be blank
```

- Pareil lorsqu'on sauvegarde :

```
Creature.create!(name: nil) # => lève l'exception:  
# ActiveRecord::RecordInvalid - Validation failed: Name can't be blank
```

- La plupart des méthodes de modèles ont une version avec un `!` qui lève une exception :

`create!` , `update!` , `save!` , `destroy!` , ...

Validations ActiveRecord : Il existe plein de validateurs

```
# Valider qu'une string a une certaine taille
validates :name, length: { minimum: 2 }
# Valider qu'un mot de passe est dans l'interval en longueur
validates :password, length: { in: 6..20 }
# Valider qu'une string est dans un tableau accepté
validates :dice_type, inclusion: { in: %w[d2 d4 d6 d8 d12 d20 d100] }
# Valider qu'un champ est plus grand qu'un autre
validates :start_date, comparison: { lesser_than: :end_date }
# Valider qu'une string match une regexp
validates :username, format: { with: /^[a-zA-Z]+$/ }
# Valider l'absence d'un champ (dépréciation/usage conditionnel)
validates :old_field, absence: true
# Plusieurs validations sur le même attribut
validates :name, length: { minimum: 5 }, uniqueness: true
```

Validations : Valider les relations

- On peut valider aussi les relations directes :

```
class Book < ApplicationRecord
  belongs_to :author
  validates :author, presence: true
end
```

- On peut aussi valider les relations indirectes :

```
class Author < ApplicationRecord
  has_many :books
  validates :books, presence: true
end
```

Validations : Valider à travers les relations

- On peut s'assurer qu'une relation est présente et valide pour mieux gérer les erreurs :

```
class Library < ApplicationRecord
  has_many :books
  validates_associated :books
end
```

- Ne pas utiliser `validates_associated` des deux côtés de votre relation, les validations s'appelleraient en boucle infinie.
- On s'en servira pour faire de la gestion d'erreur et éviter de faire échouer la totalité de la transaction.

Validations conditionnelles

- Avec `:if` / `:unless`

```
# user.rb
validates :name, presence: true, unless: :admin?
# creature.rb
validates :size, exclusion: {in: %w[big giant]}, if: -> { race == 'hobbit' }
# photon.rb
validates :position, presence: true, if: -> { speed&.zero? }
validates :speed, presence: true, unless: -> { position.present? }
```

- Avec `:on`

```
# On vérifie la présence de invitation_code uniquement à la création.
validates :invitation_code, presence: true, on: :create
# On peut créer un objet sans nom, mais on devra le remplir à la mise à jour.
validates :name, presence: true, on: :update
```

Validations exclusives

- Les `Troll` et `Ogre` sont de grande taille.
- Les `Human` et `Elf` sont de moyenne taille.
- Les `Hobbit` et `Dwarf` sont de petite taille.

```
class Creature < ApplicationRecord
  enum :size, [:small, :medium, :big]

  validates :race, inclusion: { in: %w[troll ogre]}, if: :big?
  validates :race, inclusion: { in: %w[human elf]}, if: :medium?
  validates :race, inclusion: { in: %w[hobbit dwarf]}, if: :small?
end
```

Validations personnalisées : via un validateur custom

```
class NameStartsWithX < ActiveRecord::Validator
  def validate(record)
    unless record.name.start_with? 'X'
      record.errors.add :name, "Need a name starting with X please!"
    end
  end
end

class Person < ApplicationRecord
  validates_with NameStartsWithX
end
```


Validations personnalisées : via une méthode custom

```
class CartLine < ApplicationRecord
  belongs_to :cart
  belongs_to :product

  validates :quantity, numericality: { greater_than_or_equal_to: 1 }
  validate :product_stock_must_be_sufficient

private

def product_stock_must_be_sufficient
  return unless product
  return if product.is_made_on_demand?
  return if product.stock >= quantity

  errors.add(:quantity, :insufficient_stock)
end
```

Validations offline / Validations online

Il existe deux types de validations, les validations :

- Les validations offline, qui s'exécutent sans contact à la base de données

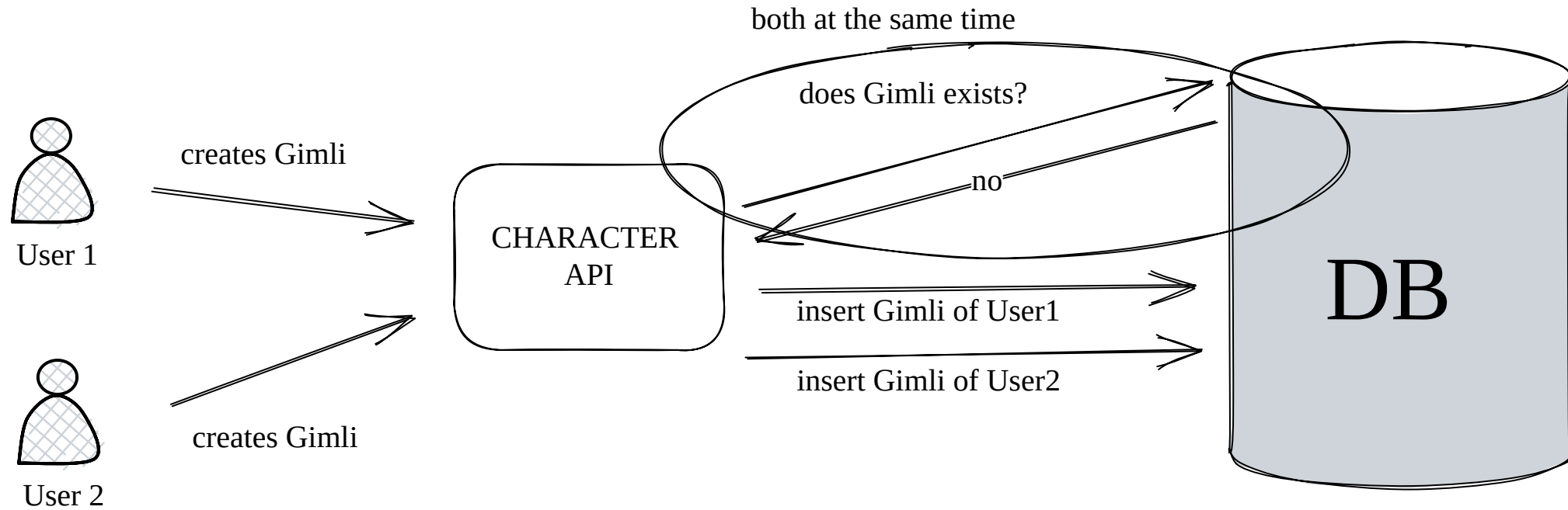
```
validates :name, length: { in: 2..128 }
```

- Les validations online, qui vérifient des choses dans la base de données

```
validates :email, uniqueness: true
```

Exemple : Au moment de la validation d'unicité, une requête d'existence est lancée sur la BDD pour savoir si un autre record existe déjà.

Race condition



Protéger l'unicité via un index

- On peut ajouter un index "unique" sur le nom du personnage.

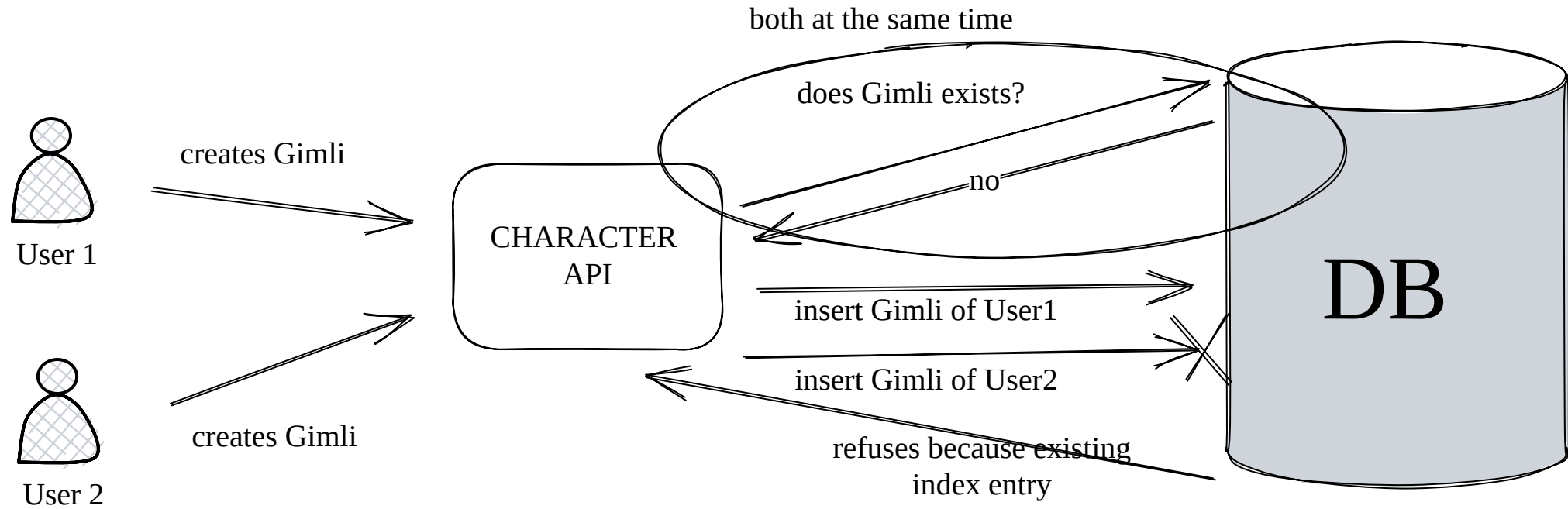
```
class AddUniqueNameToCharacters < ActiveRecord::Migration[7.0]
  def change
    add_index :characters, :name, unique: true, name: 'unique_names'
  end
end
```

Cet index permettra :

- De trouver rapidement un `Character` par nom sans faire un fullscan
- De matériellement empêcher une duplication du nom

```
Character.create(name: 'yop')
# UNIQUE constraint failed: characters.name (ActiveRecord::RecordNotUnique)
```

Race condition... avoided



Callbacks Active Record

- Dans le cycle de vie de l'application, on peut vouloir effectuer des opérations quand quelque chose arrive.
 - un utilisateur doit être validé par e-mail -> envoyer e-mail
 - une commande a été crée -> notifier le service logistique
 - une créature est morte -> l'enlever du groupe
- ActiveRecord permet de venir inscrire des callbacks (des observateurs) lorsque certaines conditions arrivent (conditions observées)

Les callbacks permettent de déclencher une logique avant ou après une alteration de l'état d'un objet.

Callbacks Active Record disponibles

- Lors de la création et la mise à jour d'un objet :
 - `before_validation` , `after_validation` , `before_save` , `around_save`
- Uniquement lors de la création d'un objet :
 - `before_create` , `around_create` , `after_create`
- Uniquement lors de la mise à jour d'un objet :
 - `before_update` , `around_update` , `after_update`
- Uniquement lors de la suppression d'un objet :
 - `before_destroy` , `around_destroy` , `after_destroy`
- Lors de la création, mise à jour et suppression :
 - `after_save` , `after_commit` , `after_rollback`

Callback Active Record disponibles et utilisation

- Lors de l'initialisation d'un objet :
 - `after_initialize`, `after_find`
- On peut s'en servir par exemple pour assigner un champ avant de valider l'objet ou de le sauvegarder.

```
class User < ApplicationRecord
  validates :login, :email, presence: true

  before_create do
    self.name = login.capitalize if name.blank?
  end
end
```


Callback Active Record : Utilisation (suite)

- On peut aussi créer un méthode qu'on déclare en tant que callback

```
class User < ApplicationRecord
  validates :login, :email, presence: true

  before_validation :ensure_login_has_a_value

  private

  def ensure_login_has_a_value
    return unless login.present?

    self.login = email unless email.blank?
  end
end
```

Callback Active Record : Utilisation (suite)

- Utilisation de `:if` / `:unless`

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

- `:if` / `:unless` accepte une liste de prédicats

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

Callback Active Record : Utilisation (suite)

- Pour le TP : assigner une taille en fonction des PVs à la création :

```
before_create do
  size = case health_points
    when 0..10
      :small
    when 11..30
      :big
    else
      :giant
    end
end
```

- Il ne faut pas abuser des callbacks sinon votre app deviendra un *"Callback Hell"*
 - A trigger B qui trigger C qui trigger A mais A trigger D 🤪🤪🤪

Trigger un callback (ou pas)






- Les méthodes suivantes déclenchent les callbacks :

`create` , `create!` , `destroy` , `destroy!` , `destroy_all` , `destroy_by` , `save` , `save!` ,
`save(validate: false)` , `toggle!` , `touch` , `update_attribute` , `update` , `update!` ,
`valid?`

- Les méthodes suivantes ne déclenchent **pas** les callbacks :

`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`

Mettre à jour sa base

- Méthodes **avec** callback : `update` / `destroy_all`
- Méthodes **sans** callback : `update_all` / `delete_all`
- Les méthodes qui ne lancent pas de callbacks effectuent juste une requête SQL
-   Les méthodes qui lancent les callbacks **instancient TOUS les objets**  
- On préférera globalement les requêtes SQL (sans callbacks) pour effectuer des mises à jour massives. 
- Note : Ne pas lancer les callbacks sur un update (sql) ne mettra pas à jour les timestamps

Callbacks dans les contrôleurs

- On peut ajouter des callbacks `before_action`, `after_action`, `around_action`

```
class ApplicationController < ActionController::Base
  before_action :require_login # peut prendre aussi un bloc
  private
  def require_login
    unless logged_in?
      flash[:error] = "You must be logged in to access this section"
      redirect_to new_login_url # coupe la requête
    end
  end
end
```

- Ces callbacks sont hérités par les contrôleurs enfants.
- Ils peuvent arrêter les requêtes entrantes sur le contrôleur.

Gérer des absences de créature

```
class CreaturesController < ActionController::Base
  def show
    @creature = Creature.find(params[:id])
    render json: @creature.as_json(only: [:id, :name, :health_points])
  rescue ActiveRecord::RecordNotFound # les mêmes deux lignes
    render nothing: true, status: 404 # répétées partout
  end

  def update
    @creature = Creature.find(params[:id])
    update_params = params.require(:creature).permit(:name)
    @creature.update(update_params)
    render json: @creature.as_json(only: [:id, :name, :health_points])
  rescue ActiveRecord::RecordNotFound
    render nothing: true, status: 404
  end
  # Pareil dans delete
end
```

Autre type de Callbacks dans les contrôleurs : Les `rescue_from`'s

- Permet de catcher une erreur levée par le contrôleur pour implémenter un gestionnaire.

```
class ApplicationController < ActionController::Base
  rescue_from ActiveRecord::RecordNotFound, with: :record_not_found

  private
  def record_not_found
    render nothing: true, status: 404
  end
end
```

- Ces callbacks sont aussi hérités par les contrôleurs enfants.

Gérer des absences de créature avec les `rescue_from`'s

```
class CreaturesController < ApplicationController
  def show
    @creature = Creature.find(params[:id])
    render json: @creature.as_json(only: [:id, :name, :health_points])
  end

  def update
    @creature = Creature.find(params[:id])
    update_params = params.require(:creature).permit(:name)
    @creature.update(update_params)
    render json: @creature.as_json(only: [:id, :name, :health_points])
  end
end
```