Écrire un ORM en Ruby

Julien Kirch

2020-07-06

Partie 1 : introduction

Dans ce texte je vais décrire pas à pas comment écrire un ORM SQL minimal en Ruby.

Avant de plonger dans le code, je vais ici introduire le sujet et expliquer certains des choix que j'ai fait.

Pourquoi un ORM ?

Un ORM est une couche qui fait le lien (un mapping) entre une base de données et la représentation des données dans un programme. Elle permet d'automatiser ou de simplifier certaines actions, notamment la génération de requêtes et enrobe les appels à la base dans une API dans le style du langage de programmation utilisé.

Elle peut aussi fournir une compatibilité entre plusieurs systèmes de bases de données et un système de cache.

Les ORMs sont probablement les outils les plus complexes auxquels on a affaire quand on commence à développer des applications web.

De ce fait, on peut avoir du mal à imaginer comment ils peuvent bien fonctionner et peuvent donner l'impression d'être presque magiques.

Impression souvent renforcée quand on lit le code de ces outils. Il est souvent assez dense et utilise des constructions différentes de celles du code applicatif auquel on est habitué.

J'ai pu observer deux conséquences à cette situation :

- les personnes ne se sentent pas à l'aise quand elles utilisent des ORMs , même longtemps après ;
- les personnes ont un avis négatif des ORMs basé essentiellement sur ce ressenti au moment de leur première rencontre (qu'on ait un avis négatif sur les ORMs basés sur des éléments objectifs ne me pose pas de problèmes).

Cela peut gêner les personnes dans leur travail, ou biaiser leur jugement.

Pour lutter contre cela, je vous propose une série d'articles décrivant pas à pas comment écrire un ORM SQL minimal en Ruby, par minimal j'entends un ORM capable de faire de l'insertion, du requêtage simple et gérer des relations entre objets.

Mon objectif est qu'après avoir lu ces articles, vous compreniez comment ils fonctionnent et que vous vous soyez approprié ce type d'outils.

Un exemple mono-base

La plupart des ORMs SQL sont compatibles avec de nombreuses bases de données. Cette compatibilité s'implémente généralement en permettant que toutes les méthodes de l'ORM soient surchargeables en fonction des spécificités de chaque base.

Par exemple pour une classe de génération de requêtes `QueryBuilder`, on aura ainsi un `SQLiteQueryBuilder`, un `PostgreSQLQueryBuilder`…

La difficulté est de parvenir à structurer le code pour pouvoir prendre en compte les spécificités de chaque base sans que cela tourne au plat de spaghetti, sachant que, suivant les bases, ces particularités ne sont pas toutes aux mêmes endroits.

Mais cela ne porte pas à conséquence sur le fonctionnement général de l'outil. Mon exemple sera donc mono-base et ciblera SQLite car elle fournit tout ce dont j'ai besoin et qu'elle est facile à installer.

Génération de code à froid plutôt qu'à chaud

Contrairement à des ORMs comme Active Record ou Sequel, l'ORM que je vais implémenter utilisera de la génération de code à froid — c'est-à-dire sous forme de fichiers — plutôt qu'à chaud sous forme de code généré à l'exécution.

Générer du code dans des fichiers permet d'examiner les comportements sans avoir à débuguer et permet un meilleur support de la part des outils comme de l'auto-complétion.

Lorsque le modèle de donnée change, il faut relancer le script de génération de code, et on peut ainsi suivre les évolutions dans le code généré en même temps que celles du code qui l'utilise.

Le code généré ainsi ne doit pas être modifié manuellement pour éviter les conflits en cas de mise à jour. Pour ajouter des méthodes aux classes générées, il vaut mieux utiliser le monkey-patching. Par exemple si les modèles sont générés dans un fichier `models.rb`, on peut utiliser un fichier `models-extension.rb` pour ajouter du code aux différentes classes des modèles.

Mon avis est que ce type d'approche est préférable à la génération de code à chaud sauf très bonne raison.

Quand j'utilise Active Record et que j'ai besoin de poser des breakpoints et d'examiner les objets en mémoire pour connaître les méthodes disponibles sur un modèle, ça me rend triste.

Travailler ainsi demande un plus gros effort de mémorisation.

En Ruby, générer du code à l'exécution est facile, ce qui est très bien quand c'est la bonne solution. Mais je pense que la communauté a tendance à trop l'utiliser. Surtout que par ailleurs le rechargement de code fonctionne très bien, par exemple dans Ruby on Rails.

Pour la génération, je vais utiliser erb. Si elle est souvent utilisée pour générer du HTML, la syntaxe erb n'est pas du tout spécifique au HTML (comme peut l'être Slim) et peut donc servir à générer du Ruby.

Voila pour l'introduction, dans la partie suivante je vais m'occuper de la structure du projet.

Partie 2 : structure

Après avoir introduit le sujet, je vais ici poser les bases de l'outil.

Le code (framework et code d'exemple) sera dans un projet unique et ne sera pas packagé sous forme d'une gem.

Le faire ajouterait pas mal de code et un peu de complexité sans que cela apporte quelque chose.

Les dépendances

Tout d'abord il faut fixer la version de Ruby, je vais prendre la dernière disponible au moment d'écrire l'article.

ruby-version.

```
2.7.1
```

Ensuite pour les bibliothèques, en plus de la gem permettant d'utiliser SQLite je vais me servir de Rake pour définir la tâche de génération de code.

Gemfile.

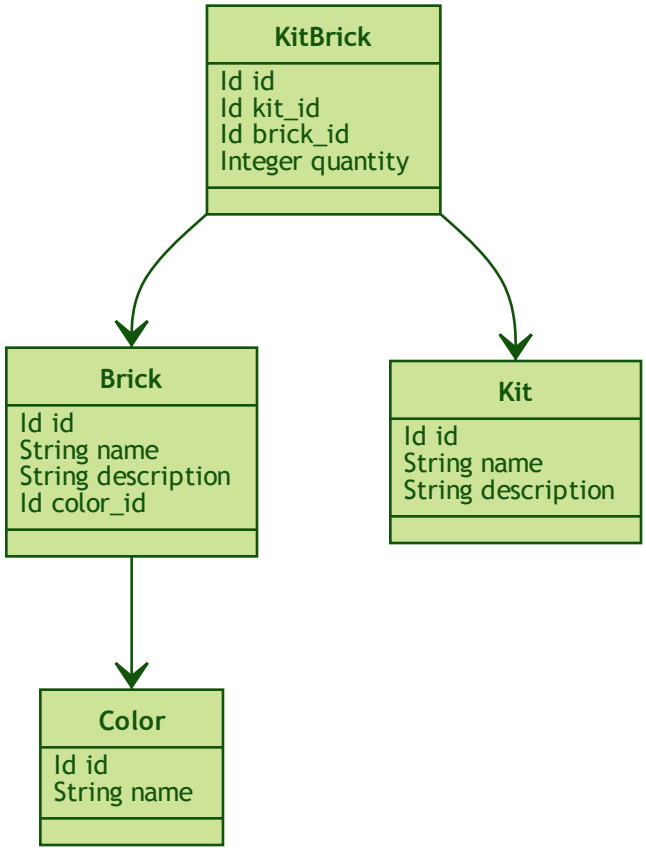
```
source "https://rubygems.org"

gem "rake", "~> 12.0"
gem "sqlite3", "~> 1.4"
```

Comme exemple : un jeu de construction

Dans la suite, je vais prendre comme exemple un jeu de construction.

Ce jeu de construction est composé des différents types de briques (`Brick`), qui sont chacun d'une certaine couleur (`Color`). Des modèles à construire (`Kit`) sont constitués d'un ensemble de types de briques, chacun contient un certain nombre de briques (tant de briques d'une sorte, tant de briques d'une autre sorte), la relation modèle - type de brique étant modélisée par un `KitBrick`.



Voici la structure SQL correspondante avec la syntaxe SQLite :

structure.sql.

```
-- Table color
CREATE TABLE 'color' (
  'id' INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,

  'name' TEXT NOT NULL
);

CREATE UNIQUE INDEX idx_color_unique
  ON color('name');

-- Table brick
CREATE TABLE 'brick' (
  'id' INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,

  'name' TEXT NOT NULL,
  'description' TEXT NOT NULL,
```

```
  'color_id' INTEGER NOT NULL,

  FOREIGN KEY('color_id') REFERENCES 'color'('id')
);

-- Table kit
CREATE TABLE 'kit' (
  'id' INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,

  'name' TEXT NOT NULL,
  'description' TEXT NOT NULL
);

-- Table kit_brick
CREATE TABLE 'kit_brick' (
  'id' INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,

  'kit_id' INTEGER NOT NULL,
  'brick_id' INTEGER NOT NULL,
  'quantity' INTEGER NOT NULL,

  FOREIGN KEY('kit_id') REFERENCES 'kit'('id'),
  FOREIGN KEY('brick_id') REFERENCES 'brick'('id')
);
CREATE UNIQUE INDEX 'idx_kit_brick_uniqu'
  ON 'kit_brick'('kit_id', 'brick_id');
```

Pour créer une base avec la bonne structure, vous pouvez lancer la commande :

```
$ sqlite3 orm-ruby.sqlite < structure.sql
```

Et ensuite explorer la base ainsi :

```
$ sqlite3 orm-ruby.sqlite
SQLite version 3.24.0 2018-06-04 14:10:15
sqlite> select count(*) from color;
0
sqlite>
```

Certains ORMs comme Rails et Sequel fournissent des outils pour gérer les modifications de schéma de la base. J'ai fait le choix ici de ne pas implémenter cette fonctionnalité car — même si elle peut partager du code avec le reste de l'ORM — elle est largement séparée et n'influe donc pas sur le noyau de l'outil.

Dans le monde Java, l'ORM hibernate qui est très utilisé ne fournit ainsi pas d'outil de migration.

Un DSL pour configurer les modèles

Pour générer les classes de modèle, je vais utiliser un fichier de configuration qui sera lu par un script. Au début le fichier de configuration contiendra seulement les noms des modèles et les noms des tables correspondantes.

La syntaxe s'inspire des DSL de configuration qu'on trouve dans Rails :

schema.rb.

```
define_model 'Color' do |model_definition|
  model_definition.table 'color'
end

define_model 'Brick' do |model_definition|
  model_definition.table 'brick'
end

define_model 'Kit' do |model_definition|
  model_definition.table 'kit'
end

define_model 'KitBrick' do |model_definition|
  model_definition.table 'kit_brick'
end
```

La capacité d'utiliser des noms de table par défaut en les déduisant des noms des classes demanderait un peu plus de code sans changer le fonctionnement d'ensemble, du coup je ne vais pas l'intégrer.

Pour générer les modèles je dois commencer par lire le contenu de fichier.

Pour cela je commencer par créer la classe `ModelDefinition` qui contiendra les contenus des modèles tels que définis dans le fichier, en étant passé dans chacun des blocs `define_model`.

generator.rb.

```
class ModelDefinition

  attr_reader :name, :table_name

  # @param name [String]
  def initialize(name)
    @name = name
  end

  # @param [String]
  # @return [void]
  def table(table_name)
    @table_name = table_name
```

```
  end
end
```

Comme le script de génération `generator.rb` des modèles sera lancé de manière indépendante du reste du code, je peux définir la méthode `define_model` de manière globale (dans un script indépendant elle ne risque pas de polluer l'espace de noms), puis de faire un `require_relative` sur le fichier de configuration.

Lorsque le fichier sera chargé, la méthode `define_model` sera ainsi appelée pour chaque bloc du fichier `schema.rb`.

Chaque appel va instancier un `ModelDefinition` avec le nom du modèle, puis le passe en paramètre du bloc.

generator.rb.

```ruby
# @yield [model_definition]
# @yieldparam [ModelDefinition] model_definition
# @yieldreturn [void]
def define_model(model_name, &block)
  puts "Defining model [#{model_name}]"
  model_definition =
    ModelDefinition.new(model_name)
  block.yield(model_definition)
end

require_relative 'schema'
```

Pour pouvoir utiliser ensuite ces `ModelDefinition`, le constructeurs les stockera dans un tableau au fur et à mesure.

generator.rb.

```ruby
class ModelDefinition

  MODELS_DEFINITIONS = []

  attr_reader :name, :table_name

  # @param name [String]
  def initialize(name)
    @name = name
    MODELS_DEFINITIONS << self
  end

  # …
```

Après le chargement du fichier de configuration, `ModelDefinition::MODELS_DEFINITIONS` contiendra la ainsi liste des définitions.

Un template pour générer le fichier

Une fois la configuration chargée je vais m'intéresser à la génération du code.

Comme à l'étape précédente, la première étape est de définir la syntaxe cible qui m'intéresse :

models.rb.

```
class Color

  # @return [String]
  def self.table_name
      'color'
  end
end
```

Chaque modèle est dans une classe, exposant une méthode de classe pour récupérer le nom de la table.

Comme expliqué plus haut, je me sers d'erb pour la génération, voici donc le template de classe correspondant :

models.rb.erb.

```
class <%= model.name %>

  # @return [String]
  def self.table_name
      '<%= model.table_name %>'
  end
end
```

Pour générer le fichier, il faut alors charger ce template, l'appliquer à chacun des définitions qui sont disponibles dans `ModelDefinition::MODELS_DEFINITIONS` et stocker le résultat dans un fichier.

generator.rb.

```
# …

require 'erb'

# Récupère le template
erb = ERB.new(IO.read('models.rb.erb'))

# Applique le template aux modèles
models_code = ModelDefinition::MODELS_DEFINITIONS.
    map do |model|
  # Fait en sorte que le ModelDefinition soit disponible dans le template
  # via la variable `model`
  erb.result_with_hash(model: model)
end
```

```
# Concatène le code des modèles et l'écrit dans un fichier
IO.write(
    'models.rb',
    models_code.
        join("\n\n")
)
```

Le code est alors terminé, il me manque seulement une tâche Rake pour pouvoir l'invoquer. Comme les chemins des fichiers sont tous en dur dans le code, il n'y a pas besoin de le rendre paramétrable :

Rakefile.

```
desc 'Génère les modèles à partir du fichier schema.rb'
task :generate_models do
  require_relative 'generator'
end
```

On peut alors lancer la génération :

```
$ rake generate_models
Defining model [Color]
Defining model [Brick]
Defining model [Kit]
Defining model [KitBrick]
```

Et observer le résultat :

models.rb.

```
class Color

  # @return [String]
  def self.table_name
      'color'
  end
end

class Brick

  # @return [String]
  def self.table_name
      'brick'
  end
end

class Kit
```

```ruby
  # @return [String]
  def self.table_name
      'kit'
  end
end

class KitBrick

  # @return [String]
  def self.table_name
      'kit_brick'
  end
end
```

Pour le moment, tout ce que je peux faire c'est d'instancier les différentes classes :

```ruby
require_relative 'models'
black = Color.new
```

Mais la structure est en place et dans la partie suivante je vais pouvoir m'en servir pour faire mes premières requêtes.

Partie 3 : récupération des champs, insertion et sélection

Après avoir introduit le sujet puis avoir posé les bases de l'outil, je vais ici ajouter ce qu'il manque pour faire les premières requêtes.

Des modèles avec des champs

La première étape va s'intéresser aux champs des modèles.

Il serait possible de les paramétrer via le fichier de configuration de schéma `schema.rb`, mais je vais plutôt utiliser les métadonnées de la base de données, car c'est une bonne occasion de donner un aperçu de leur fonctionnement.

En effet, même si tous les ORMs n'utilisent pas les métadonnées, beaucoup le font il est donc important de comprendre comment cela fonctionne et de réaliser que ça n'est pas si compliqué.

Même si cela ne fait pas partie du standard SQL, la majorité des bases de données de ce type fournissent des moyens d'accéder aux métadonnées comme les tables, les index...

Suivant les systèmes on pourra utiliser du SQL — en interrogeant des tables particulières — ou des commandes spécifiques.

Avec SQLite, l'accès aux métadonnées se fait via des commandes PRAGMA.

```
$ sqlite3 orm-ruby.sqlite
sqlite> .header on -- Affiche les noms de colonnes
sqlite> pragma table_info('color'); -- Infos sur la table `color`
```

```
cid|name|type|notnull|dflt_value|pk
0|id|INTEGER|1||1
1|name|TEXT|1||0
```

Pour chaque colonne `table_info` nous donne son identifiant, son nom, son type, si elle est nullable, son éventuelle valeur par défaut et si il s'agit de la clé primaire de la table.

Avec la gem sqlite3, la méthode `SQLite3::Database#table_info` fait l'appel à la pragma et renvoie le résultat sous une forme accessible.

Pour ajouter les colonnes disponibles dans chaque table, commençons par définir une classe `ColumnDefinition` qui contiendra les informations qui nous intéressent, à savoir le nom et le type de la colonne.

generator.rb.

```ruby
class ColumnDefinition

  attr_reader :name, :type

  # @param name [String]
  # @param type [String]
  def initialize(name, type)
    @name = name
    @type = type
  end

end
```

Ensuite, je dois récupérer les colonnes de chaque table. L'endroit le plus simple est de le faire juste avant de créer les classes de modèles, alors qu'on est déjà en train d'itérer sur chaque modèle, on pourra alors passer la liste des colonnes au template.

Mais pour commencer, il faut se connecter à la base de données. Je vais utiliser une base en local avec un fichier stocké dans le répertoire du projet :

generator.rb.

```ruby
DATABASE = SQLite3::Database.new('orm-ruby.sqlite')
```

puis :

generator.rb.

```ruby
models_code = ModelDefinition::MODELS_DEFINITIONS.map do |model|
  # Liste les colonnes de la table correspondante
  columns_definitions = DATABASE.table_info(model.table_name).collect do |column_info|
    column_name = column_info['name']
    column_type = column_info['type']
    ColumnDefinition.new(column_name, column_type)
```

```
  end
  erb.result_with_hash(model: model, columns_definitions: columns_definitions)
end
```

Dans le template, `columns_definitions` contient alors la liste des `ColumnDefinition` prête à l'emploi.

Presque prêt à l'emploi car il reste une subtilité : `table_info` renvoie les types SQL des colonnes (ou plus précisément le type SQLite) comme `TEXT`, pour pouvoir l'utiliser dans le code je dois le transformer en types Ruby comme `String`.

Pour se faire, nous allons utiliser une `Hash` pour faire la conversion entre les deux.

generator.rb.

```
SQLITE_TYPE_TO_RUBY_CLASS = {
    'INTEGER' => 'Integer',
    'TEXT' => 'String'
}
```

Elle ne contient que les types que l'on va rencontrer dans notre exemple, il sera ensuite possible de l'enrichir en fonction des besoins.

En modifiant le code pour utiliser `SQLITE_TYPE_TO_RUBY_CLASS`, cela donne :

generator.rb.

```
models_code = ModelDefinition::MODELS_DEFINITIONS.map do |model|
  # Liste les colonnes de la table correspondante
  columns_definitions = DATABASE.table_info(model.table_name).collect do |column_info|
    column_name = column_info['name']
    sql_type = column_info['type']
    # Transforme le type SQL en type Ruby
    ruby_type = SQLITE_TYPE_TO_RUBY_CLASS[sql_type]
    ColumnDefinition.new(column_name, ruby_type)
  end
  erb.result_with_hash(model: model, columns_definitions: columns_definitions)
end
```

Avec le nom et le type de chaque colonne, je vais pouvoir générer les getters et les setters en itérant sur les `ColumnDefinition` dans le template :

models.erb.rb.

```
class <%= model.name %>

  <% columns_definitions.each do |column_definition| %>
  <% column_name = column_definition.name %>
  <% column_type = column_definition.type %>
  # @return [<%= column_type %>]
  def <%= column_name %>
```

```
    @<%= column_name %>
  end

  # @param <%= column_name %> [<%= column_type%>]
  # @return [void]
  def <%= column_name %>=(<%= column_name %>)
    @<%= column_name %> = <%= column_name %>
  end
  <% end %>

end
```

Ce qui donne ce résultat :

models.rb.

```
class Model

  # @return [Integer]
  def id
    @id
  end

  # @param id [Integer]
  # @return [void]
  def id=(id)
    @id = id
  end

  # @return [String]
  def name
    @name
  end

  # @param name [String]
  # @return [void]
  def name=(name)
    @name = name
  end
end
# …
```

Ce qui permet d'écrire :

```
require_relative 'models'
```

```ruby
black = Color.new
black.name = 'Black'
```

On peut voir ici l'intérêt de la génération de code à froid : on peut facilement consulter les méthodes disponibles avec leurs informations de type. Avec un IDE on peut même disposer de l'autocomplétion.

En cas d'évolution d'un modèle, l'évolution sera visible dans les classes générées.

Je ne l'utilise pas dans mon exemple, mais l'information de nullabilité des colonnes peut servir pour renseigner la nullabilité des paramètres ou des retours des méthodes.

L'insertion

Une fois qu'on a la liste des champs et qu'il est possible de leur attribuer des valeurs, il est temps de pouvoir insérer ces données dans la base, en ajoutant une méthode `insert` aux modèles.

Pour cela il faut générer ce type de requêtes :

```sql
INSERT INTO table_name
  (column_name_1, column_name_2)
  values (column_value_1, column_value_2)
```

Pour partager le code entre les modèles, je vais ajouter une classe `Model` qui sera parente des classes de modèles.

model.rb.

```ruby
# @abstract
class Model
end
```

Je la marque comme abstraite avec `@abstract` pour indiquer qu'elle n'est pas utilisable directement mais qu'on doit passer par les classes dérivées.

Pour générer les requêtes d'insertion, je vais avoir besoin du nom de la table et de la liste des colonnes de chaque modèle. Pour cela je vais ajouter des méthodes de classes pour récupérer les valeurs.

Je les déclare dans la classe parente :

model.rb.

```ruby
# @abstract
class Model

  # Méthode à implémenter dans les sous-classes
  # @abstract
  # @return [String]
  def self.table_name
    raise NotImplementedError
  end
```

```ruby
  # Méthode à implémenter dans les sous-classes
  # @abstract
  # @return [Array<String>]
  def self.columns
    raise NotImplementedError
  end
end
```

Puis je les ajoute au template de modèle, avec la déclaration de l'héritage :

models.erb.rb.

```
class <%= model.name %> < Model
  # @return [String]
  def self.table_name
      '<%= model.table_name %>'
  end

  # @return [Array<String>]
  def self.columns
      <%= columns_definitions.map do |column_definition|
        column_definition.name
      end %>
  end
end
```

Ce qui donne, après avoir relancé la génération avec la commande `rake generate_models` :

models.rb.

```ruby
class Color < Model

  # @return [String]
  def self.table_name
      'color'
  end

  # @return [Array<String>]
  def self.columns
      ["id", "name"]
  end

  # …
end
```

Avec ces méthodes je peux générer la requête, en ajoutant une connexion à la base pour pouvoir l'exécuter.

Pour la requête je vais utiliser la méthode `SQLite3::Database#execute`, qui permet de passer les valeurs

des colonnes en paramètre plutôt que de les mettre dans le corps de la requête, ce qui donnera ce genre d'appel :

```
DATABASE.execute('INSERT INTO color (name) values (?)', ['Black'])
```

Cette syntaxe permet d'éviter d'avoir à se préoccuper du format à utiliser pour passer les valeurs à la base, cela simplifie le code et évite d'introduire des risques de sécurité en cas de problème d'échappement.

Dans notre cas les valeurs des `id` des modèles ne doivent pas être insérées car elles sont gérées par la base, c'est pour cela que les colonnes `id` sont déclarées en `AUTOINCREMENT`. Cela simplifie le code et fournit une garantie d'unicité dans le cas d'une base SQL standard.

La manière de s'y prendre n'est pas standardisée et dépend donc de la base de données. Il y a deux grandes approches : soit les valeurs sont retournées par la requête d'insertion, ou une requête spécifique permet de récupérer les `id` des valeurs qu'on vient d'insérer.

SQLite utilise la deuxième solution via `last_insert_rowid()`.

model.rb.

```
require 'sqlite3'

# @abstract
class Model

  # Connection à la base pour executer les requêtes
  DATABASE = SQLite3::Database.new('orm-ruby.sqlite')

  # @return [void]
  def insert
    # Liste des noms de colonnes sans la colonne id
    # car les valeurs des ids sont gérées par la base
    columns_names_except_id = self.class.columns.
        select { |column| column != 'id' }

    # Noms des colonnes échappées pour éviter
    # les problèmes avec des guillemets et d'autres symboles
    quoted_columns_names_except_id = columns_names_except_id.
        map { |column_name| SQLite3::Database.quote(column_name) }

    # Valeurs des colonnes à part la colonne 'id'
    columns_values_except_id = columns_names_except_id.
        map { |column_name| self.send(column_name) }

    # Les requêtes vont ressembler à
    # INSERT INTO table_name
    #   (column_name_1, column_name_2)
```

```
    #   VALUES (?, ?)
    DATABASE.execute(
        "INSERT INTO #{SQLite3::Database.quote(self.class.table_name)} " +
            "(#{quoted_columns_names_except_id.join(', ')}) " +
            "VALUES (#{Array.new(columns_names_except_id.length, '?').join(', ')})",
        columns_values_except_id
    )

    # Définit la valeur  du champ `id` du modèle
    # en récupérant la valeur attribuée par la base
    self.id = DATABASE.last_insert_row_id
  end
end
```

Les méthodes `table_name` et `columns` étant implémentées dans chaque classe de modèle, utiliser `self.class.table_name` et `self.class.columns` dans la classe parente `Model` appellera bien la méthode spécifique de chaque modèle plutôt que les méthodes de la classe `Model`.

Avec ce code, on peut enfin insérer les données :

script.rb.

```
require_relative 'model'
require_relative 'models'

black = Color.new
black.name = 'Black'
black.insert

brick = Brick.new
brick.color_id = black.id
brick.name = 'Awesome brick'
brick.description = 'This brick is awesome'
brick.insert
```

On peut vérifier dans la base que tout s'est bien passé :

```
$bundle exec ruby script.rb
$ sqlite3 orm-ruby.sqlite

sqlite> select * from color;

1|Black

sqlite> select * from brick;

1|Awesome brick|This brick is awesome|1
```

La récupération

Maintenant que je peux insérer des données, je vais pouvoir m'intéresser à leur récupération.

Je commence par m'occuper de la récupération de l'intégralité des données d'une table en ajoutant une méthode de classe `all` aux modèles.

Cela permettra des appels du type :

```
Color.all
```

En SQL cela donne ce type de requêtes :

```
SELECT column_name_1, column_name_2
  FROM table_name
```

Les noms de la table et des colonnes sont à disposition pour construire la requête.

Une fois les valeurs récupérées, pour chaque ligne trouvée il faut créer une instance de la classe du modèle et attribuer leurs valeurs aux différents champs.

Les noms des attributs étant les mêmes que ceux des colonnes, pour chaque colonne `nom_de_colonne`, j'appellerait le setter `nom_de_colonne=` via la méthode `send` qui permet d'appeler une méthode dynamiquement à partir de son nom.

À l'inverse du cas précédent, il nous faudra également récupérer la valeur de la colonne `id`.

model.rb.

```
class Model
  # @return [Array]
  def self.all
    quoted_columns_names = columns.
        map { |column_name| SQLite3::Database.quote(column_name) }

    # Les requêtes vont ressembler à
    # SELECT column_name_1, column_name_2
    #   FROM table_name
    DATABASE.execute(
        "SELECT #{quoted_columns_names.join(', ')} " +
            "FROM #{SQLite3::Database.quote(table_name)}"
    ).map do |result_row|
      # Instancie l'objet de la classe du modèle
      model_instance = self.new
      # Pour chaque colonne
      columns.each_with_index do |column, column_index|
        # On récupère la valeur
        column_value = result_row[column_index]
        # On stocke la valeur dans l'attribue correspondant
        model_instance.send("#{column}=", colonne_value)
```

```
      end
      model_instance
    end
  end
end
```

Je peux alors récupérer des données :

script.rb.

```
require_relative 'model'
require_relative 'models'

black = Color.new
black.name = 'Black'
black.insert

puts 'Les couleurs'
Color.all.each do |color|
  puts "  #{color.id} : #{color.name}"
end

brick = Brick.new
brick.color_id = black.id
brick.name = 'Awesome brick'
brick.description = 'This brick is awesome'
brick.insert

puts 'Les briques'
Brick.all.each do |brick|
  puts "  #{brick.id} : #{brick.name}, #{brick.description}, #{brick.color_id}"
  puts brick.id
  puts brick.name
  puts brick.description
  puts brick.color_id
end
```

```
$ bundle exec ruby script.rb
Les couleurs
  1 : Black
Les briques
  1 : Awesome brick, This brick is awesome, 1
```

Et la suppression

Pour terminer, après l'insertion et la récupération il est temps de supprimer des données.

Dans le standard SQL, il existe une commande `TRUNCATE table_name` qui supprime le contenu d'une table.

Malheureusement elle n'est pas disponible dans SQLite, je vais donc devoir utiliser la requête SQL :

```
DELETE FROM table_name
```

Je vais tout de même nommer ma méthode `truncate` pour qu'elle corresponde à la commande SQL standard, même si l'implémentation SQLite utilise pas cette commande.

On a ici un exemple où l'ORM doit assurer la compatibilité entre les systèmes de bases de données. Si ce cas est assez simple, il permet de comprendre la manière dont les choses pourraient être mises en œuvre : une méthode de base qui utiliserait la commande `truncate` et une classe spécifique à SQLite qui utiliserait la requête `delete`.

Le code résultant est assez court et s'inspire des méthodes existantes :

model.rb.

```ruby
class Model
  # @return [void]
  def self.truncate
    DATABASE.execute("DELETE FROM #{SQLite3::Database.quote(table_name)}")
  end
end
```

On peut alors la tester

script.rb.

```ruby
require_relative 'model'
require_relative 'models'

Brick.truncate
Color.truncate

puts '# Les couleurs'
Color.all.each do |color|
  puts "  #{color.id} : #{color.name}"
end

puts 'Les briques'
Brick.all.each do |brick|
  puts "  #{brick.id} : #{brick.name}, #{brick.description}, #{brick.color_id}"
  puts brick.id
  puts brick.name
  puts brick.description
  puts brick.color_id
end
```

```
$ bundle exec ruby script.rb
Les couleurs
Les briques
```

Je ne vais pas les détailler ici mais pour les suppressions de données il faut générer des requêtes `DELETE FROM table_name WHERE ID = ?` et leurs passer l'id de l'instance à supprimer et pour les mises à jour s'inspirer des requêtes d'insertion pour obtenir des requêtes du type `UPDATE table_name SET column_name_1 = ?, column_name_2 = ? WHERE id = ?`.

Toutes les requêtes vues ici s'appuient sur l'hypothèse d'un identifiant technique présent dans toutes les tables, ce qui est la pratique généralement conseillée en SQL. Prendre en compte les autres types d'identifiants demande de rendre paramétrable cette partie des requêtes.

C'est tout pour le moment, dans la partie suivante je vais enrichir les méthodes de récupération pour pouvoir ajouter des filtres et trier les données.

Partie 4 : filtrage et ordre

Après avoir introduit le sujet, avoir posé les bases de l'outil puis avoir ajouté la génération des requêtes, je vais m'occuper ici du filtrage et de l'ordre des résultats dans les requêtes de sélection.

À la fin de l'article, il sera possible d'utiliser ce genre d'appels :

```
Color.where('name = ?', 'Black').first
Color.where('name = ?', 'b*').order_by('name desc').all
```

L'approche "builder"

Ce type d'API utilise souvent le design pattern "builder" qui consiste à stocker les différent paramètres d'une requête dans des objets et les utiliser au moment où la requête est exécutée.

Ainsi la commande

```
Color.where('name = ?', 'b*').order_by('name desc').all
```

peut se décomposer en

```
query_parameters_1 = Color.where('name = ?', 'b*')
query_parameters_2 = query_parameters_1.order_by('name desc')
result = query_parameters_2.all
```

Le premier appel `Color.where` construit un objet qui contient la condition `where`, appeler `order_by` sur cet objet construit un autre objet qui contient la condition `where` ainsi que l'ordre de tri du `order_by`.

Finalement appeler `all` sur le dernier objet utilise tous les paramètres stockés jusque là pour construire la requête.

Quand on enchaîne plusieurs appels comme dans

```
Color.where('name = ?', 'b*').order_by('name desc').all
```

le code construit donc une suite d'objets, un à chaque appel.

La classe `QueryParameters`

La classe `QueryParameters` va donc contenir les différents types de paramètres qu'on peut utiliser dans une requêtes. Dans notre exemple je vais couvrir le cas des filtres, de l'ordre et du nombre d'éléments remontés, mais dans un ORM plus complet il peut s'agit de l'ensemble de la grammaire SQL : `group by`, `distinct`, pouvoir choisir quels champs sont remontés...

Voici le début de la classe avec ses attributs :

model.rb.

```
class QueryParameters

  # @param [Class] model_class
  def initialize(model_class)
    @model_class = model_class
    @wheres = []
    @order_bys = []
  end
end
```

En plus des paramètres pour les `where` et les `order by`, elle a besoin de connaître la classe du modèle à traiter pour avoir accès au nom de la table et aux champs et pour savoir quels objets instancier avec les résultats.

Comme décrit plus haut, appeler `where` ou `order_by` construit une nouvelle instance de `QueryParameters` en ajoutant le paramètre correspondant et la renvoie.

La méthode `order_by` prend un seul paramètre qui est une chaîne de caractère qui contient l'information de tri comme `'name desc'`. Ils sont stockés sous la forme d'un tableau dans l'attribut `@order_bys`.

La méthode `where` prend un nombre variable de paramètres, le premier est une chaîne de caractère qui contient l'expression SQL comme `'name = ?'` et les éventuels paramètres suivants contiennent les valeurs à utiliser comme `'Black'` et auront donc des types variés (chaîne de caractères, nombre entier ou à virgule flottante). Ils sont stockés sous la forme d'un tableau de table de hachages (`Hash`) dans l'attribut `@wheres`,
.

model.rb.

```
class QueryParameters
  # …

  # @param expression [String]
  # @param parameters [Array]
  # @return [QueryParameters]
  def where(expression, *parameters)
    new_query_parameters = self.dup
    new_query_parameters.wheres << {
      expression: expression,
```

```
      parameters: parameters
    }
    new_query_parameters
  end

  # @param order_by [String]
  # @return [QueryParameters]
  def order_by(order_by)
    new_query_parameters = self.dup
    new_query_parameters.order_bys << order_by
    new_query_parameters
  end
end
```

Les méthodes du modèle

Cela permet de construire un `QueryParameters` à partir d'un `QueryParameters` existant, mais il faut aussi pouvoir construire le premier `QueryParameters` à partir du modèle, par exemple quand on appelle `Color.where('name = ?', 'Black')`.

Cela signifie ajouter ces mêmes méthodes aux modèles, qui initialisent le `QueryParameters` avec la classe du modèle et appellent les méthodes correspondantes sur le `QueryParameters`.

model.rb.

```
class Model
  # @return [QueryParameters]
  def self.query_parameters
    QueryParameters.new(self)
  end

  # @param expression [String]
  # @param parameters [Array]
  # @return [QueryParameters]
  def self.where(expression, *parameters)
    query_parameters.where(expression, *parameters)
  end

  # @param order_by [String]
  # @return [QueryParameters]
  def self.order_by(order_by)
    query_parameters.order_by(order_by)
  end
end
```

Avec cela je peux appeler `Color.where('name = ?', 'b*').order_by('name desc')` et récupérer un

`QueryParameters` avec les bonnes données, reste ensuite à s'occuper de la génération de la requête.

La requête

La méthode existante `Model#all` construit ce genre de requêtes :

```
SELECT column_name_1, column_name_2
  FROM table_name
```

Avec les nouveaux paramètres, cela va donner :

```
SELECT column_name_1, column_name_2
  FROM table_name
  WHERE column_A = ? AND column_B < ?
  ORDER BY column_X asc, column_Y desc
```

Pour les `where` et `order by` la logique est la même : s'il existe au moins un paramétre de ce type, ajouter la clause en concaténants les éléments séparés par des `AND` ou des virgules et pour le `where` il faut ensuite passer les valeurs à la requête sous forme d'un tableau contenant l'ensemble des éléments dans le bon ordre.

La partie finale de la méthode qui instancie et renseigne les modèles est reprise de la méthode `Model#all`.

C'est un peu fastidieux mais pas si long que ça :

model.rb.

```
class QueryParameters
  # …

  # @return [Array]
  def all
    quoted_columns_names = @model_class.columns.
        map { |column_name| SQLite3::Database.quote(column_name) }

    if @wheres.empty?
      where_clause = ' '
      where_params = []
    else
      where_content = @wheres.map do |where|
        where[:expression]
      end.join(' AND ')
      where_clause = "WHERE #{where_content} "
      where_params = @wheres.map { |where| where[:parameters] }.flatten
    end

    if @order_bys.empty?
      order_by_clause = ''
```

```
    else
      order_by_clause = "ORDER BY #{@order_bys.join(', ')} "
    end

    # Les requêtes vont ressembler à
    # SELECT column_name_1, column_name_2
    #   FROM table_name
    #   WHERE column_A = ? AND column_B < ?
    #   ORDER BY column_X asc, column_Y desc
    DATABASE.execute(
        "SELECT #{quoted_columns_names.join(', ')} " +
            "FROM #{@model_class.quoted_table_name} " +
            where_clause +
            order_by_clause,
        where_params
    ).map do |result_row|
      # Construit les instances du modèle
      model_instance = @model_class.new
      @model_class.columns.each_with_index do |column, column_index|
        model_instance.send("#{column}=", result_row[column_index])
      end
      model_instance
    end
  end

end
```

Ne me reste plus qu'à remplacer l'implémentation de `Model#all` existante par un appel à cette nouvelle méthode, pour pouvoir récupérer tous les éléments d'un modèle.

model.rb.

```
class Model
  # …

  # @return [Array]
  def self.all
    query_parameters.all
  end
end
```

C'est le moment de tester :

script.rb.

```
require_relative 'model'
require_relative 'models'
```

```
Brick.truncate
Color.truncate

black = Color.new
black.name = 'Black'
black.insert

yellow = Color.new
yellow.name = 'Yellow'
yellow.insert

brick = Brick.new
brick.color_id = black.id
brick.name = 'Awesome brick'
brick.description = 'This brick is awesome'
brick.insert

puts '# All colors'
Color.all.each do |color|
  puts color.id
  puts color.name
end

puts '# All Bricks'
Brick.all.each do |brick|
  puts brick.id
  puts brick.name
  puts brick.description
  puts brick.color_id
end

puts '# Black color'
Color.where('name = ?', 'Black').all.each do |color|
  puts color.id
  puts color.name
end

puts '# Colors by name'
Color.order_by('name desc').all.each do |color|
  puts color.id
  puts color.name
end
```

```
$ bundle exec ruby script.rb
# All colors
73
Black
74
Yellow
# All Bricks
55
Awesome brick
This brick is awesome
73
# Black color
73
Black
# Colors by name
74
Yellow
73
Black
```

Limiter les résultats

Pour terminer cet article, je vais encore ajouter un cas, celui de la clause `limit` qui permet de limiter le nombre de résultats à récupérer en spécifiant un entier.

Au lieu de stocker les différentes valeurs comme pour `where` et `order by`, on ne conserve qu'une valeur. On pourrait aussi envisager de lever une exception si une valeur a déjà été spécifiée plus tôt dans la chaîne des `QueryParameters`.

`limit` est le plus souvent utilisé indirectement quand on veut récupérer une seule valeur, sous la forme d'une méthode `first` qui spécifie le `limit` à 1, puis renvoie le premier élément du tableau de résultat.

Cette méthode me sera utile dans l'article suivant pour les requêtes de relations.

model.rb.

```ruby
class QueryParameters
  attr_writer :limit
  attr_reader :wheres, :order_bys, :limit

  # @param model_class [Class]
  def initialize(model_class)
    @model_class = model_class
    @wheres = []
    @order_bys = []
    @limit = nil
  end
```

```
  # @param limit [Integer]
  # @return [Model::QueryParameters]
  def limit(limit)
    new_query_parameters = self.dup
    new_query_parameters.limit = limit
    new_query_parameters
  end

    # @return [Array]
  def all
    # …
    if @limit.nil?
      limit_clause = ' '
    else
      limit_clause = "LIMIT #{@limit} "
    end

    # Les requêtes vont ressembler à
    # SELECT column_name_1, column_name_2
    #   FROM table_name
    #   WHERE column_A = ? AND column_B < ?
    #   ORDER BY column_X asc, column_Y desc
    #   LIMIT 10
    DATABASE.execute(
        "SELECT #{quoted_columns_names.join(', ')} " +
            "FROM #{@model_class.quoted_table_name} " +
            where_clause +
            order_by_clause +
            limit_clause,
        where_params
    ).map do |result_row|
    #
  end

  def first
    limit(1).all.first
  end
end
```

Reste encore à ajouter les méthodes sur le modèle qui font la délégation à `QueryParameters` :

model.rb.

```
class Model
```

```
  # @param limit [Integer]
  # @return [Model::QueryParameters]
  def self.limit(limit)
    query_parameters.limit(limit)
  end

  # @return [Object]
  def self.first
    query_parameters.first
  end
end
```

C'est tout pour le moment, dans l'article suivant j'ajouterai une gestion minimale des relations entre objets permettant de parcourir une grappe de dépendances.

Partie 5 : relations

Après avoir introduit le sujet, avoir posé les bases de l'outil puis avoir ajouté la génération des requêtes et enfin m'être occupé du filtrage des requêtes, je vais m'occuper ici des relations entre objets.

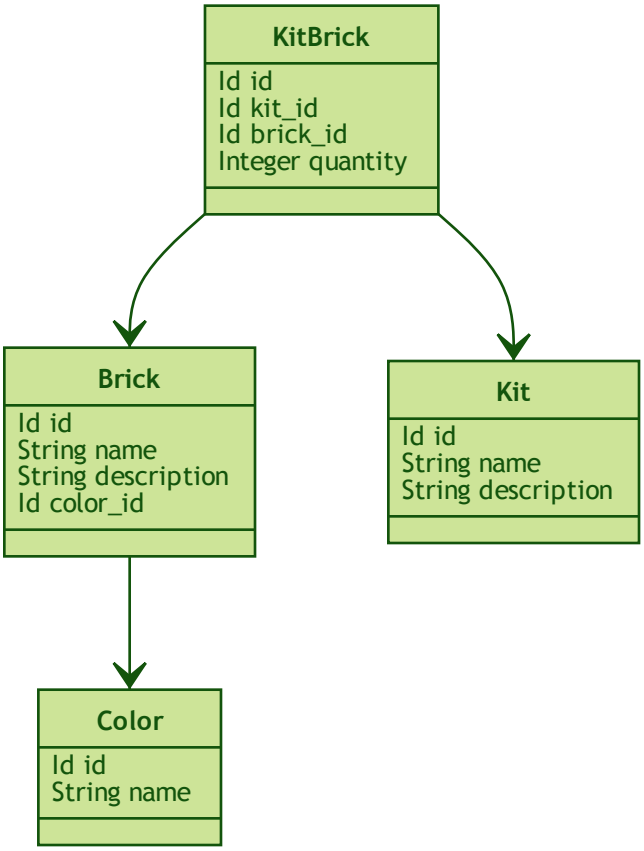
Les ORM permettent en général de définir quatre types de relations : `one-to-many`, `many-to-one`, `one-to-one` et `many-to-many`.

Je vais couvrir les deux premiers (`one-to-many`, `many-to-one`) parce que dans mon expérience on les rencontre beaucoup plus souvent que les deux autres, et que leur implémentation est la plus simple.

Le modèle

Les articles utilisent le modèle d'un jeu de construction qui fournit des relations one-to-many et many-to-one.

Ce jeu de construction est composé des différents types de briques (`Brick`), qui sont chacun d'une certaine couleur (`Color`). Des modèles à construire (`Kit`) sont constitués d'un ensemble de types de briques chacun présent un certain nombre de briques (tant de briques d'une sorte, tant de briques d'une autre sorte), la relation modèle - type de brique étant modélisé par un `KitBrick`.



Le `many-to-one`

Extension du DSL

Le DSL que j'avais défini permettait de définir les noms des tables et des classes de modèles. Il est temps de l'étendre pour y ajouter des informations sur les relations.

Je vais commencer par le `many-to-one`, en introduisant une méthode `has_one`. Pour définir une relation de ce type j'ai besoin du nom de la classe à lier, du nom de la colonne qui contient l'id et du nom de l'attribut que je veux ajouter à la classe de modèle.

schema.rb.

```
define_model 'Color' do |model_definition|
  model_definition.table 'color'
end

define_model 'Brick' do |model_definition|
  model_definition.table 'brick'
  model_definition.has_one(
      attribute_name: 'color',
```

```
      model_class: 'Color',
      column_name: 'color_id'
  )
end

define_model 'Kit' do |model_definition|
  model_definition.table 'kit'
end

define_model 'KitBricks' do |model_definition|
  model_definition.table 'kit_brick'
  model_definition.has_one(
      attribute_name: 'kit',
      model_class: 'Kit',
      column_name: 'kit_id'
  )
  model_definition.has_one(
      attribute_name: 'brick',
      model_class: 'Brick',
      column_name: 'brick_id'
  )
end
```

Que dois-je obtenir ?

Maintenant comment gérer les relations ?

Je vais prendre pour exemple la relation entre brique et couleur :

```
define_model 'Brick' do |model_definition|
  model_definition.table 'brick'
  model_definition.has_one(
      attribute_name: 'color',
      model_class: 'Color',
      column_name: 'color_id'
  )
end
```

Quand je veux récupérer la couleur d'une brique, je dois charger la couleur qui correspond à la valeur de la colonne `color_id`, disponible dans le modèle par la méthode `color_id`. En SQL cela donnerait quelque chose comme :

```
SELECT color_column_name_1, color_column_name_1
  FROM color
  WHERE id = ?
```

Avec les méthodes définies dans l'article précédent cela donne

```
Color.where('id = ?', brick.color_id).first
```

Cette couleur doit être accessible à travers un getter nommé `color` :

```
class Brick < Model
  # @return [Color]
  def color
    Color.where('id = ?', color_id).first
  end
end
```

Pour le setter `color=`, il n'y a pas de besoin de SQL : je peut me contenter de stocker la valeur de l'attribut `color_id`, la valeur sera alors sauvegardée en même temps que les autres attributs de l'objet. Si juste ensuite je refais un appel à `color`, la requête récupérera la couleur correspond au nouvel `color_id`.

```
class Brick < Model
  # @param color [Color]
  # @return [void]
  def color=(color)
    @color_id = color.id
  end
end
```

Maintenant je sais quel code je veux obtenir.

Implémentation

Pour y arriver, je commence par implémenter la méthode `has_one` dans le DSL pour qu'elle stocke les informations qu'on lui passe.

generator.rb.

```
class ModelDefinition

  MODELS_DEFINITIONS = []

  attr_reader :name, :table_name, :has_ones

  # @param name [String]
  def initialize(name)
    @name = name
    @has_ones = []
    MODELS_DEFINITIONS << self
  end

  # …

  # @param attribute_name [String]
```

```
  # @param model_class [String]
  # @param column_name [String]
  # @return [void]
  def has_one(attribute_name:, model_class:, column_name:)
    @has_ones << {
        attribute_name: attribute_name,
        model_class: model_class,
        column_name: column_name
    }
  end
end
```

Pour le template je retranscris le code auquel j'avais abouti plus haut en utilisant les différentes valeurs :

models.rb.erb.

```
  <% model.has_ones.each do |has_one| %>
  # @return [<%= has_one[:model_class] %>]
  def <%= has_one[:attribute_name] %>
    <%= has_one[:model_class] %>.where('id = ?', <%= has_one[:column_name] %>).first
  end

  # @param <%= has_one[:attribute_name] %> [<%= has_one[:model_class] %>]
  # @return [void]
  def <%= has_one[:attribute_name] %>=(<%= has_one[:attribute_name] %>)
    @<%= has_one[:column_name] %> = <%= has_one[:attribute_name] %>.id
  end
  <% end %>
```

On peut alors tester que cela fonctionne :

script.rb.

```
require_relative 'model'
require_relative 'models'

black = Color.new
black.name = 'Black'
black.insert

brick = Brick.new
brick.color = black
brick.name = 'Awesome brick'
brick.description = 'This brick is awesome'
brick.insert

puts brick.color.name
```

```
$ bundle exec ruby script.rb
Black
```

L'exemple d'ORM que je décris ici ne gère pas de cache, ce qui signifie que chaque appel de `brick.color` va générer une nouvelle requête SQL.

Le `one-to-many`

La mise en œuvre du `one-to-many` est très similaire.

Je commence par définir la syntaxe dans le DSL avec une méthode `has_many`.

schema.rb.

```
define_model 'Color' do |model_definition|
  model_definition.table 'color'
  model_definition.has_many(
      attribute_name: 'bricks',
      model_class: 'Brick',
      column_name: 'color_id'
  )
end

define_model 'Brick' do |model_definition|
  model_definition.table 'brick'
  model_definition.has_one(
      attribute_name: 'color',
      model_class: 'Color',
      column_name: 'color_id'
  )
  model_definition.has_many(
      attribute_name: 'kit_brick',
      model_class: 'KitBricks',
      column_name: 'brick_id'
  )
end

define_model 'Kit' do |model_definition|
  model_definition.table 'kit'
  model_definition.has_many(
      attribute_name: 'kit_brick',
      model_class: 'KitBricks',
      column_name: 'kit_id'
  )
end
```

```
define_model 'KitBricks' do |model_definition|
  model_definition.table 'kit_brick'
  model_definition.has_one(
      attribute_name: 'kit',
      model_class: 'Kit',
      column_name: 'kit_id'
  )
  model_definition.has_one(
      attribute_name: 'brick',
      model_class: 'Brick',
      column_name: 'brick_id'
  )
end
```

Qui devrait générer ce type de code :

models.rb.

```
class Color < Model

  # @return [Array<Brick>]
  def bricks
    Brick.where('color_id = ?', id).all
  end

end
```

Je ne vais pas définir le setter car il est assez rare, en général ce type de modification se fait plutôt de l'autre côté de la relation.

J'ajouter la nouvelle méthode `has_many` au DSL :

generator.rb.

```
class ModelDefinition

  MODELS_DEFINITIONS = []

  attr_reader :name, :table_name, :has_ones, :has_manys

  # @param name [String]
  def initialize(name)
    @name = name
    @has_ones = []
    @has_manys = []
    MODELS_DEFINITIONS << self
  end
```

```
  # …

  def has_many(attribute_name:, model_class:, column_name:)
    @has_manys << {
        attribute_name: attribute_name,
        model_class: model_class,
        column_name: column_name
    }
  end
end
```

Et pour terminer, le template :

models.rb.erb.

```
  <% model.has_manys.each do |has_many| %>
  # @return [Array<<%= has_many[:model_class] %>>]
  def <%= has_many[:attribute_name] %>
    <%= has_many[:model_class] %>.where('<%= has_many[:column_name] %> = ?', id).all
  end
  <% end %>
```

Ce qui donne :

script.rb.

```
require_relative 'model'
require_relative 'models'

black = Color.new
black.name = 'Black'
black.insert

brick = Brick.new
brick.color = black
brick.name = 'Awesome brick'
brick.description = 'This brick is awesome'
brick.insert

puts black.bricks.length
puts black.bricks.first.name
```

```
$ bundle exec ruby script.rb
1
Awesome brick
```

Pour finir

Et voila ! À ce stade j'ai la base d'un ORM minimal.

Le code se trouve à https://github.com/archiloque/orm-ruby.

Il manque quelques éléments pour qu'il soit vraiment utile, par exemple la gestion des `UPDATE` et de la suppression unitaire (plutôt que de vider toute une table avec `truncate`), mais une implémentation minimale s'appuierait beaucoup à ce qui a déjà été fait sans introduire de nouvelles idées.

J'espère que ces articles ont pu vous donner une aperçu du fonctionnement de ce type d'outils et les ont rendus moins mystérieux.

S'ils vous donne des idées pour coder votre propre ORM d'une manière différente, lancez-vous, tant que vous restez raisonnable dans vos ambitions, notamment celle de l'utiliser en production.

Si d'autres éléments vous semblent compliqués, contactez-moi et j'ajouterai peut-être ce contenu dans un article supplémentaire.