Лекція 9. Основи Active Record

План.

1. Загальне поняття про Active Record

2. Міграції Active Record

1. Що таке Active Record?

Active Record це M в MVC - модель - яка є шаром в системі, відповідальним за представлення бізнес-логіки і даних. Active Record спрощує створення і використання бізнес-об'єктів, дані яких вимагають персистентного зберігання у базі даних. Сама по собі ця реалізація патерну Active Record є описом системи ORM (Object Relational Mapping).

1.1. Патерн Active Record

Active Record *був описаний Martin Fowler в його книзі Patterns* of Enterprise Application Architecture. У Active Record об'єкти містять і персистентні дані, і поведінка, яка працює з цими даними. Active Record виходить з міркувань, що забезпечення логіки доступу до даних як частини об'єкту покаже користувачам цього об'єкту те, як читати і писати у базу даних.

1.2. Object Relational Mapping (ORM)

Object Relational Mapping (об'єктно-реляційне відображення), що зазвичай згадується як абревіатура ORM, це техніка, що поєднує складні об'єкти застосування з таблицями в системі управління реляційними базами даних. За допомогою ORM, властивості і відношення цих об'єктів застосування можуть бути з легкістю збережені і отримані з бази даних без безпосереднього написання виразів SQL, і, у результаті, з меншим сумарним кодом для доступу у базу даних.

Розуміння того, як працюють реляційні бази даних, має вирішальне значення для розуміння Active Record і Rails в цілому.

1.3. Active Record це фреймворк ORM

Active Record надає нам декілька механізмів, найбільш важливими з яких є можливості для :

- Представлення моделей і їх даних.
- Представлення зв'язків між цими моделями.
- Представлення ієрархій спадкоємства за допомогою пов'язаних моделей.
- Валідації моделей до того, як вони стануть персистентними у базі даних.
- Виконання операцій з базою даних в об'єктно-орієнтованому стилі.
- **Персистентні структури даних** (англ. *persistent data structure*) — це структури даних, які при внесенні в них якихось змін зберігають усі свої попередні стани і доступ до цих станів.

2. Домовленості над конфігурацією в Active Record

При написанні застосування з використанням інших мов програмування або фреймворків часто вимагається писати багато конфігураційного коду. Зокрема, це справедливо для фреймворків ORM. Проте, якщо наслідувати Домовленостям, прийнятим Rails, вам доведеться написати зовсім небагато конфігурацій (а іноді і зовсім не потрібно) при створенні моделей Active Record. Ідея в тому, що у більшості випадків ви настроюєте свої застосування однаковим чином, і цей спосіб має бути способом По замовчуванню. Таким чином, явна конфігурація знадобиться тільки тоді, коли ви не наслідуєте Домовленості з якоїсь причини.

2.1. Домовленості по назвах

По замовчуванню Active Record використовує деякі домовленості по назвах щоб дізнатися, як має бути створений зв'язок між моделями і таблицями бази даних. Rails утворює множину для імен класу, щоб знайти відповідну таблицю бази даних. Так, для класу `Book` слід створити таблицю бази даних з ім'ям **books**. Механізми утворення множини Rails дуже потужні, вони здатні утворювати множину (і однину) як для правильних, так і для неправильних форм слів. При використанні імен класу, створених з двох і більше слів, ім'я класу моделі повинне наслідувати Домовленості Ruby, використовуючи форму CamelCase, тоді як ім'я таблиці повинне містити слова, розділені знаком підкреслення. Приклади:

- Таблиця бази даних - множинна форма із словами, розділеними знаком підкреслення (тобто, book_clubs).
- Клас моделі - однина з першою прописною буквою в кожному слові (тобто, `BookClub`).

Модель / Клас	Таблиця / Схема
`Article`	`articles`
`LineItem`	`line_items`
`Deer`	`deers`
`Mouse`	`mice`
`Person`	`people`

2.2. Домовленості по схемах

Active Record використовує домовленості про назви стовпців в таблицях бази даних, залежно від призначення цих стовпців.

- **Зовнішні ключі** - Ці поля повинні називатися за зразком `singularized_table_name_id` (тобто, `item_id`, `order_id`). Це поля, які шукає Active Record при створенні зв'язків між вашими моделями.
- **Первинні ключі** - По замовчуванню Active Record використовує числовий стовпець з ім'ям `id` як первинний ключ таблиці. Цей стовпець буде автоматично створений при використанні міграцій Active Record для створення таблиць.

Також є деякі опціональні імена стовпців, що додають додаткові особливості для екземплярів Active Record :

- `created_at` - Автоматично будуть встановлені поточна дата і час при первинному створенні запису.
- `updated_at` - Автоматично будуть встановлені поточна дата і час всякий раз, коли оновлюється запис.

- `lock_version` - Додає оптимістичне блокування до моделі.
- `type` - Вказує, що модель використовує Single Table Inheritance.
- `(association_name)_type` - Зберігає тип для поліморфних зв'язків.
- `(table_name)_count` - Використовується для кешування кількості об'єктів, що належать по зв'язку. Наприклад, стовпець `comments_count` в класі `Article`, у якого може бути декілька пов'язаних екземплярів `Comment`, закэширует кількість існуючих коментарів для кожної статті.

Хоча ці імена стовпців опціональні, фактично вони зарезервовані Active Record. Уникайте зарезервованих ключових слів, якщо ви не бажаєте додатковій функціональності. Наприклад, `type` - це зарезервоване слово для визначення таблиці, що використовує спадкоємство з єдиною таблицею (STI). Якщо ви не використовуєте STI, спробуйте використати аналогічне слово, таке як "context", яке також може акуратно описати дані, які ви моделюєте.

3. Створення моделей Active Record

Створювати моделі Active Record дуже просто. Усе, що необхідно зробити, - це створити підклас `ApplicationRecord`, і готово:

```
class Product < ApplicationRecord
end
```

Це створить модель `Product`, зв'язавши її з таблицею `products` у базі даних. Зробивши так, також з'явиться здатність зв'язати стовпці кожного рядка цієї таблиці з атрибутами екземплярів вашої моделі. Припустимо, що таблиця `products` була створена з використанням такого виразу SQL:

```
CREATE TABLE products (
   id int(11) NOT NULL auto_increment,
   name varchar(255),
   PRIMARY KEY (id)
);
```

Вищезгадана схема оголошує таблицю з двома стовпцями: `id` і `name`. Кожен рядок цієї таблиці є певним продуктом з цими двома параметрами. Таким чином, можна написати подібний код:

```
p = Product.new
p.name = "Some Book"
puts p.name # "Some Book"
```

4. Перевизначення угод про назви

Але що, якщо ви наслідуєте іншу угоду по назвах або використовуєте нове застосування Rails із старою базою даних? Не проблема, можна просто перевизначити домовленості по замовчуванню.

`ApplicationRecord` наслідується від `ActiveRecord::Base`, який визначає ряд корисних методів. Можна використати метод `ActiveRecord::Base.table_name=` для вказівки імені таблиці, яка має бути використана :

```
class Product < ApplicationRecord
  self.table_name = "my_products"
end
```

Якщо так зробити, треба вручну визначити ім'я класу, фикстури (my_products.yml, фікстури - це по суті тестові дані), використовуючи метод `set_fixture_class` у визначенні тесту :

```
class ProductTest < ActiveSupport::TestCase
  set_fixture_class my_products: Product
  fixtures :my_products
  ...
end
```

Також можливо перевизначити стовпець, який має бути використаний як первинний ключ таблиці, за допомогою методу `ActiveRecord::Base.primary_key=`:

```
class Product < ApplicationRecord
  self.primary_key = "product_id"
end
```

5. CRUD: Читання і запис даних

CRUD це скорочення для чотирьох дієслів, використовуваних для опису операцій з даними : **Create (**створити), **Read (**прочитати), **Update (**відновити) і **Delete (**видалити). Active Record автоматично створює методи, що дозволяють застосуванню читати і впливати на дані, що зберігаються у своїх таблицях.

5.1. Створення

Об'єкти Active Record можуть бути створені з хеша, блоку або з вручну вказаних після створення атрибутів. Метод `new` поверне новий об'єкт, тоді як `create` поверне об'єкт і збереже його у базу даних.

Наприклад, для моделі `User` з атрибутами `name` і `occupation`, виклик методу `create` створить і збереже новий запис у базу даних :

```
user = User.create(name: "David", occupation: "Code Artist")
```

Використовуючи метод `new`, об'єкт може ініціалізувати без збереження:

```
user = User.new
user.name = "David"
user.occupation = "Code Artist"
```

Виклик `user.save` передасть запис у базу даних.

Нарешті, якщо наданий блок і `create`, і `new` передадуть новий об'єкт в цей блок для ініціалізації:

```
user = User.new do |u|
  u.name = "David"
  u.occupation = "Code Artist"
end
```

5.2. Читання

Active Record надає багатий API для доступу до даних у базі даних. Нижче декілька прикладів різних методів доступу до даних, наданих Active Record.

```
# поверне колекцію з усіма користувачами
users = User.all
```

```
# поверне першого користувача
user = User.first
#поверне першого користувача з ім'ям David
david = User.find_by(name: 'David')
# знайде усіх користувачів з ім'ям David, які Code Artists, і сортує їх по
created_at в зворотному хронологічному порядку
users = User.where(name: 'David ', occupation: 'Code Artist').order(created_at::
desc)
```

5.3. Оновлення

Як тільки об'єкт Active Record буде отриманий, його атрибути можуть бути модифіковані, і він може бути збережений у базу даних.

```
user = User.find_by(name: 'David')
user.name = 'Dave'
user.save
```

Скороченим варіантом для цього є використання хэша з атрибутами, пов'язаними з бажаними значеннями, таким чином:

```
user = User.find_by(name: 'David')
user.update(name: 'Dave')
```

Це найкорисніше, коли необхідно відновити декілька атрибутів за раз. Якщо, з іншого боку, необхідно відновити декілька записів за раз, корисний метод класу `update_all` :

```
User.update_all "max_login_attempts = 3, must_change_password = 'true'"
```

5.4. Видалення

Більше того, після отримання, об'єкт Active Record може бути знищений, що прибере його з бази даних.

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

Якщо необхідно видалити відразу декілька записів, можна використати метод `destroy_all` :

```
# знайти і видалити усіх користувачів з ім'ям David
User.where(name: 'David').destroy_all

# видалити усіх користувачів
User.destroy_all
```

1. Огляд міграцій

Міграції - це зручний спосіб змінювати схему вашої бази даних час від часу. Вони використовують Ruby DSL (Domain Specific Languages). Тому вам не треба писати SQL вручну, дозволяючи вашій схемі бути незалежною від бази даних.

Кожну міграцію можна розглядати як нову 'версію' бази даних. Схема спочатку нічого не містить, а кожна міграція модифікує її, додаючи або прибираючи таблиці, стовпці або записи. Active Record знає, як оновлювати вашу схему з часом, переносячи її з певної точки у минулому в останню версію. Active Record також оновлює ваш файл db/`schema.rb`, щоб він відповідав поточній структурі вашої бази даних.

Ось приклад міграції :

```
class CreateProducts < ActiveRecord::Migration[5.0]
  def change
    create_table :products do |t|
      t.string :name
      t.text :description

      t.timestamps
    end
  end
end
```

Ця міграція додає таблицю `products` із строковим стовпцем name і текстовим стовпцем description. Первинний ключ, названий id, також буде неявно доданий по замовчуванню, оскільки це первинний ключ по замовчуванню для усіх моделей Active Record. Макрос timestamps додає два стовпці, created_at і updated_at. Ці спеціальні стовпці автоматично управляються Active Record, якщо існують.

Відмітьте, що ми визначили зміну, яка повинна відбуватися при русі вперед в часі. До запуску цієї міграції таблиці немає. Після - таблиця існуватиме. Active Record також знає, як відмінити цю міграцію: якщо ми відкочуємо цю міграцію, він видалить таблицю.

У базах даних, таких, що підтримують транзакції з виразами, що змінюють схему, міграції обертаються в транзакцію. Якщо база даних це не підтримує, і міграція провалюється, частини, які пройшли успішно, не відкотять назад. Вам треба провести відкат вручну.

Деякі запити не можуть бути запущені в транзакції. Якщо ваш адаптер підтримує транзакції DDL, можна використати `disable_ddl_transaction!` для їх відключення для окремої міграції.

Якщо хочете міграцію для чогось, що Active Record не знає, як обернути, ви можете використати `reversible`:

```
class ChangeProductsPrice < ActiveRecord::Migration[5.0]
  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
```

З іншого боку, можна використати `up` і `down` замість `change`:

```
class ChangeProductsPrice < ActiveRecord::Migration[5.0]
  def up
    change_table :products do |t|
      t.change :price, :string
    end
  end

  def down
    change_table :products do |t|
      t.change :price, :integer
    end
  end
end
```

2. Створення міграції

2.1. Створення автономної міграції

Міграції зберігаються як файли в директорії `db/migrate`, один файл на кожен клас. Ім'я файлу має вигляд `YYYYMMDDHHMMSS_create_products.rb`, це означає, що часова мітка UTC ідентифікує міграцію, потім йде знак підкреслення, потім йде ім'я міграції, де слова розділені підкресленнями. Ім'я класу міграції містить буквену частину назви файлу, але вже у форматі CamelCase (тобто слова пишуться разом, кожне слово розпочинається з великої букви). Наприклад, `20190206120000_create_products.rb` повинен визначати клас `CreateProducts`, а `20190206120001_add_details_to_products.rb` повинен визначати `AddDetailsToProducts`. Rails використовує цю мітку, щоб визначити, яка міграція має бути запущена і в якому порядку, так що якщо ви копіюєте міграції з іншого застосування або генеруєте файл самі, будьте пильніші.

Звичайно, обчислення часових міток не забавне, тому Active Record надає генератор для управління цим:

```
$bin/rails generate migration AddPartNumberToProducts
```

Це створить порожню, але правильно названу міграцію:

```
class AddPartNumberToProducts < ActiveRecord::Migration[5.0]
  def change
  end
end
```

Якщо ім'я міграції має форму "AddXXXToYYY" або "RemoveXXXFromYYY" і далі слідує перелік імен стовпців і їх типів, то в міграції будуть створені відповідні вирази `add_column` і `remove_column`.

```
$bin/rails generate migration AddPartNumberToProducts part_number:string
```

згенерує

```
class AddPartNumberToProducts < ActiveRecord::Migration[5.0]
  def change
    add_column :products, :part_number, :string
  end
end
```

Якщо ви хочете додати індекс на новий стовпець, ви можете зробити це так

```
$ bin/rails generate migration AddPartNumberToProducts part_number:string:index
```

згенерує

```
class AddPartNumberToProducts < ActiveRecord::Migration[5.0]
  def change
    add_column :products, :part_number, :string
    add_index :products, :part_number
  end
end
```

Так само можна згенерувати міграцію для видалення стовпця з командного рядка:

```
$ bin/rails generate migration RemovePartNumberFromProducts part_number:string
```

генерує

```
class RemovePartNumberFromProducts < ActiveRecord::Migration[5.0]
  def change
    remove_column :products, :part_number, :string
  end
end
```

Ви не обмежені одним згенерованим стовпцем. Наприклад:

```
$ bin/rails generate migration AddDetailsToProducts part_number:string
price:decimal
```

генерує

```
class AddDetailsToProducts < ActiveRecord::Migration[5.0]
  def change
    add_column :products, :part_number, :string
    add_column :products, :price, :decimal
  end
end
```

Якщо ім'я міграції має форму "CreateXXX" і потім слідує список імен і типів стовпців, то буде згенерована міграція, що генерує таблицю XXX з перерахованими стовпцями. Наприклад:

```
$ bin/rails generate migration CreateProducts name:string part_number:string
```

генерує

```
class CreateProducts < ActiveRecord::Migration[5.0]
  def change
    create_table :products do |t|
      t.string :name
      t.string :part_number
    end
  end
end
```

Як завжди, те, що було згенеровано, є усього лише стартовою точкою. Ви можете додавати і прибирати рядки, як вважаєте потрібним, відредагувавши файл `db/migrate/YYYYMMDDHHMMSS_add_details_to_products.rb`.

Також генератор приймає такий тип стовпця, як `references` (чи його псевдонім `belongs_to`). Наприклад

```
$ bin/rails generate migration AddUserRefToProducts user:references
```

генерує

```
class AddUserRefToProducts < ActiveRecord::Migration[5.0]
  def change
    add_reference :products, :user, foreign_key: true
  end
end
```

Ця міграція створить стовпець `user_id` і відповідний індекс. Більше опцій для `add_reference` описані в документації API.

Існує також генератор, який проводитиме об'єднання таблиць, якщо `JoinTable` є частиною назви.

Наприклад

```
$ bin/rails generate migration CreateJoinTableCustomerProduct customer product
```

Згенерує наступну міграцію:

```
class CreateJoinTableCustomerProduct < ActiveRecord::Migration[5.0]
  def change
    create_join_table :customers, :products do |t|
      # t.index [:customer_id, :product_id]
      # t.index [:product_id, :customer_id]
    end
  end
end
```

2.2. Генератори моделі

Генератори моделі і скаффолда створять міграції, відповідні для створення нової моделі. Міграція міститиме інструкції для створення відповідної таблиці. Якщо ви повідомите Rails, які стовпці ви хочете, то вирази для додавання цих стовпців також будуть створені. Наприклад, запуск:

```
$ bin/rails generate model Product name:string description:text
```

створить міграцію, яка виглядає так

```
class CreateProducts < ActiveRecord::Migration[5.0]
  def change
    create_table :products do |t|
      t.string :name
      t.text :description

      t.timestamps
    end
  end
end
```

Можна визначити скільки завгодно пар ім'я_стовпця/тип.

2.3. Передача модифікаторів

Деякі часто використовувані модифікатори типу можуть бути передані безпосередньо в командному рядку. Вони вказуються у фігурних дужках і йдуть за типом поля :

Приміром, запуск:

```
$bin/rails generate migration AddDetailsToProducts 'price:decimal{5,2}'
supplier:references{polymorphic}
```

створить міграцію, яка виглядає як ця :

```
class AddDetailsToProducts < ActiveRecord::Migration[5.0]
  def change
    add_column :products, :price, :decimal, precision: 5, scale: 2
    add_reference :products, :supplier, polymorphic: true
  end
end
```

Щоб дізнатися про подробиці, зверніть увагу на повідомлення генератора, що виводяться.

3. Написання міграції

Як тільки ви створили свою міграцію, використовуючи один з генераторів, прийшов час попрацювати!

3.1. Створення таблиці

Метод `create_table` один з найфундаментальніших, але у більшості випадків, він буде згенерований для вас генератором моделі або скаффолда. Звичайне використання таке

```
create_table :products do |t|
  t.string :name
end
```

Це створить таблицю `products` із стовпцем name (і, як обговорювалося вище, стовпцем id, що мається на увазі).

По замовчуванню `create_table` створить первинний ключ, названий `id`. Ви можете змінити ім'я первинного ключа за допомогою опції: `primary_key` (не забудьте також відновити відповідну модель), або, якщо ви взагалі не хочете первинний ключ, можна вказати опцію `id :false`. Якщо треба передати базі даних специфічні опції, ви можете помістити фрагмент `SQL` в опцію `:options`. Наприклад:

```
create_table :products, options: "ENGINE=BLACKHOLE" do |t|
  t.string :name, null: false
end
```

додасть `ENGINE=BLACKHOLE` до SQL вираженню, використовуваному для створення таблиці.

Також можна передати опцію: `comment` з будь-яким описом для таблиці, який буде збережено в самій базі даних, і може бути проглянуто за допомогою інструментів адміністрування бази даних, таких як MySQL Workbench або PgAdmin III. Дуже рекомендується залишати коментарі в міграціях для застосувань з великими базами даних, оскільки це допомагає зрозуміти модель даних і згенерувати документацію. Нині коментарі підтримують тільки адаптери MySQL і PostgreSQL.

3.2. Створення сполучної таблиці

Міграційний метод `create_join_table` створює сполучну таблицю HABTM (has and belongs to many, багато до багатьом). Звичайне використання буде таким:

```
create_join_table :products, :categories
```

що створить таблицю `categories_products` з двома стовпцями на ім'я `category_id` і `product_id`. У цих стовпців є опція: `null`, встановлена в `false` По замовчуванню. Це може бути перевизначено опцією: `column_options`:

```
create_join_table :products, :categories, column_options: { null: true }
```

По замовчуванню, ім'я сполучної таблиці виходить як з'єднання перших двох аргументів, переданих в create_join_table, в алфавітному порядку. Щоб настроїти ім'я таблиці, передайте опцію: `table_name`:

```
create_join_table :products, :categories, table_name: :categorization
```

створює таблицю `categorization`.

По замовчуванню `create_join_table` створить два стовпці без опцій, але можна визначити ці опції з використанням опції: `column_options`. Наприклад

```
create_join_table :products, :categories, column_options: { null: true }
```

створить `product_id` і `category_id` з опцією: `null` рівною `true`.

`create_join_table` також приймає блок, який можна використати для додавання індексів (які По замовчуванню не створюються) або додаткових стовпців :

```
create_join_table :products, :categories do |t|
  t.index :product_id
  t.index :category_id
end
```

3.3. Зміна таблиць

Близький родич `create_table` це `change_table`, використовуваний для зміни існуючих таблиць. Він використовується подібно `create_table`, але у об'єкту, що передається у блок, більше методів. Наприклад:

```
change_table :products do |t|
  t.remove :description, :name
  t.string :part_number
  t.index :part_number
  t.rename :upccode, :upc_code
end
```

видаляє стовпці `description` і `name`, створює строковий стовпець `part_number` і додає індекс на нього. Нарешті, він перейменовує стовпець `upccode`.

3.4. Зміна стовпців

Подібно `remove_column` і `add_column`, Rails надає міграційний метод `change_column`.

```
change_column :products, :part_number, :text
```

Він міняє тип стовпця `part_number` в таблиці `products` на: `text`. Відмітьте, що команда `change_column` — необратима.

Окрім `change_column`, методи `change_column_null` і `change_column_default` використовуються щоб змінити обмеження не-null або значення стовпця По замовчуванню.

```
change_column_null :products, :name, false
change_column_default :products, :approved, from: true, to: false
```

Це настроїть поле: `name` в products бути `NOT NULL` стовпцем і змінить значення По замовчуванню для поля: `approved` з true на false.

Також можна написати попередню міграцію `change_column_default` як `change_column_default: products,: approved, false`, але, на відміну від попереднього прикладу, це зробило б вашу міграцію безповоротною.

3.5. Модифікатори стовпця

Модифікатори стовпця можуть бути застосовані при створенні або зміні стовпця :

- `limit` Встановлює максимальний розмір полів `string/text/binary/integer`.
- `precision` Визначає точність для полів `decimal`, що визначає загальну кількість цифр в числі.
- `scale` Визначає масштаб для полів `decimal`, що визначає кількість цифр після коми.
- `polymorphic` Додає стовпець `type` для зв'язків `belongs_to`.
- `null` Дозволяє або забороняє значення `NULL` в стовпці.
- `default` Дозволяє встановити значення По замовчуванню для стовпця. Відмітьте, що якщо ви використовуєте динамічне значення (таке як дату), значення По замовчуванню буде вичислено лише один раз (тобто на дату, коли міграція буде застосована).
- `index` Додає індекс для стовпця.
- `comment` Додає коментар для стовпця.

Деякі адаптери можуть підтримувати додаткові опції; за подробицями звернетеся до документації API конкретних адаптерів.

За допомогою командного рядка не можна вказати `null` і `default`.

3.6. Зовнішні ключі

Хоча це і не потрібно, ви можете захотіти додати обмеження зовнішнього ключа для забезпечення посилальної цілісності.

```
add_foreign_key :articles, :authors
```

Це додасть новий зовнішній ключ до стовпця `author_id` таблиці `articles`. Ключ посилається на стовпець `id` таблиці `authors`. Якщо імена стовпців не можуть бути проведені з імен таблиць, можна використати опції: `column` і: `primary_key`.

Rails згенерує ім'я для кожного зовнішнього ключа, що починається з `fk_rails_` плюс 10 символів, які детерміновано генеруються на основі `from_table` і `column`. Також є опція: `name`, якщо хочете вказати інше ім'я.

Active Record підтримує зовнішні ключі тільки для окремих стовпців. Щоб використати складені зовнішні ключі, потрібно `execute` і `structure.sql`. Дивіться Вивантаження схеми

Прибрати зовнішній ключ також просто:

```
# дозволимо Active Record з'ясувати ім'я стовпця
remove_foreign_key: accounts,: branches

# приберемо зовнішній ключ для певного стовпця
remove_foreign_key: accounts, column:: owner_id

# приберемо зовнішній ключ по імені
remove_foreign_key: accounts, name:: special_fk_name
```

3.7. Коли хелперів недостатньо

Якщо хелперів, наданих Active Record, недостатньо, можна використати метод `execute` для виконання довільного SQL :

```
Product.connection.execute("UPDATE products SET price = 'free' WHERE 1=1")
```

Більше подробиць і прикладів окремих методів міститься в документації по API. Зокрема, документація для ActiveRecord::ConnectionAdapters::SchemaStatements `(який` забезпечує методи, доступні в методах up, down і change), ActiveRecord::ConnectionAdapters::TableDefinition (який забезпечує методи, доступні у об'єкту, переданого `методом create_`table) і ActiveRecord::ConnectionAdapters::Table (який забезпечує методи, доступні у об'єкту, переданого методом change_table).

3.8. Використання методу `change`

Метод `change` це основний метод написання міграцій. Він працює у більшості випадків, коли Active Record знає, як обернути міграцію автоматично. На даний момент метод `change` підтримує тільки ці визначення міграції :

- add_column
- add_foreign_key
- add_index
- add_reference
- add_timestamps
- change_column_default (необхідно вказати опції: from і: to)
- change_column_null
- create_join_table
- create_table
- disable_extension
- drop_join_table
- drop_table (необхідно вказати блок)
- enable_extension
- remove_column (необхідно вказати тип)
- remove_foreign_key (необхідно вказати другу таблицю)
- remove_index
- remove_reference
- remove_timestamps
- rename_column
- rename_index
- rename_table

`change_table` також є оборотним, поки блок не викликає `change`, `change_default` або `remove`.

`remove_column` обратима, якщо надати тип стовпця третім аргументом. Також надайте опції оригінального стовпця, інакше Rails не зможе в точності перестворювати цей стовпець при відкаті:

```
remove_column: posts,: slug,: string, null: false, default: '', index: true
```

Якщо ви потребуєте використання інших методів, слід використати `reversible` або писати методи `up` і `down` замість методу `change`.

3.9. Використання `reversible`

Комплексна міграція може включати процеси, які Active Record не знає як обернути. Ви можете використати `reversible`, щоб вказати що робити коли запускається міграція і коли вона вимагає відкату. Наприклад:

```
class ExampleMigration < ActiveRecord::Migration
  def change
    create_table: distributors do |t|
      t.string: zipcode
    end

    reversible do |dir|
      dir.up do
        # додамо обмеження CHECK
        execute <<- SQL
          ALTER TABLE distributors
            ADD CONSTRAINT zipchk
              CHECK (char_length(zipcode) = 5) NO INHERIT;
        SQL
      end
      dir.down do
        execute <<- SQL
          ALTER TABLE distributors
            DROP CONSTRAINT zipchk
        SQL
      end
    end

    add_column: users,: home_page_url,: string
    rename_column: users,: email,: email_address
  end
end
```

Використання `reversible` гарантує, що інструкції виконаються в правильному порядку. Якщо попередній приклад міграції відкочується, `down` блок почне виконаються після того, як стовпець `home_page_url` буде видалений і перед перш ніж станеться видалення таблиці `distributors`.

Іноді міграція робитиме те, що просто безповоротно; наприклад, вона може знищити деякі дані. У таких випадках, ви можете викликати `ActiveRecord::IrreversibleMigration` у вашому `down` блоці. Якщо хто-небудь спробує відмінити вашу міграцію, буде відображена помилка, що це не може бути виконано.

3.10. Використання методів `up/down`

Ви також можете використати старий стиль міграцій використовуючи `up` і `down` методи, замість `change`. Метод `up` повинен описувати зміни, які необхідно внести в схему, а метод `down` міграції повинен обертати зміни, внесені методом `up`. Іншими словами, схема бази даних повинна залишитися незмінною після виконання `up`, а потім `down`. Наприклад, якщо створити таблицю в методі `up`, її слід видалити в методі `down`. Розумно проводити відміну змін в повністю протилежному порядку тому, в якому вони зроблені в методі `up`. Тоді приклад з розділу про `reversible` буде еквівалентний:

```
class ExampleMigration < ActiveRecord::Migration[5.0]
  def up
    create_table: distributors do |t|
      t.string: zipcode
    end

    #додаємо обмеження CHECK
    execute <<- SQL
      ALTER TABLE distributors
        ADD CONSTRAINT zipchk
        CHECK (char_length(zipcode) = 5);
    SQL

    add_column: users,: home_page_url,: string
    rename_column: users,: email,: email_address
  end

  def down
    rename_column: users,: email_address,: email
    remove_column: users,: home_page_url

    execute <<- SQL
      ALTER TABLE distributors
        DROP CONSTRAINT zipchk
    SQL

    drop_table: distributors
  end
end
```

Якщо ваша міграція не обратима вам слід викликати `ActiveRecord::IrreversibleMigration` з вашого методу `down`. Якщо хто-небудь спробує відмінити вашу міграцію, буде відображена помилка, що це не може бути виконано.

3.11. Повернення до попередніх міграцій

Ви можете використати можливість Active Record, щоб відкотити міграції за допомогою `revert` методу:

```
require_relative '20121212123456_example_migration'

class FixupExampleMigration < ActiveRecord::Migration[5.0]
  def change
    revert ExampleMigration

    create_table(: apples) do |t|
      t.string: variety
    end
  end
end
```

Метод `revert` також може приймати блок. Це може бути корисно для відкату вибраної частини попередніх міграцій. Для прикладу, давайте уявимо, що `ExampleMigration` закоммичена, а пізніше ми вирішили, що було б краще використати валідації Active Record, замість обмеження `CHECK`, для перевірки zipcode.

```
class DontUseConstraintForZipcodeValidationMigration <
ActiveRecord::Migration[5.0]
  def change
    revert do
      reversible do |dir|
        dir.up do
          # додамо обмеження CHECK
          execute <<- SQL
            ALTER TABLE distributors
              ADD CONSTRAINT zipchk
                CHECK (char_length(zipcode) = 5);
          SQL
        end
        dir.down do
          execute <<- SQL
            ALTER TABLE distributors
              DROP CONSTRAINT zipchk
          SQL
        end
      end

      # The rest of the migration was ok
    end
  end
end
```

Подібна міграція також може бути написана без використання `revert`, але це б привело до ще декількох кроків: зміна порядку (дотримання) `create table` і `reversible`, заміна `create_table` на `drop_table` і зрештою зміна `up` на `down` і навпаки. Про усе це вже потурбувався revert.

Якщо необхідно додати обмеження `CHECK`, як у вищезгаданих прикладах, треба використати `structure.sql` в якості методу для вивантаження. Дивіться Вивантаження схеми.

4. Запуск міграцій

Rails надає ряд завдань bin/rails для запуску певних наборів міграцій.

Найперша міграція, що відноситься до завдання bin/rails, яку використовуватимемо, це `rails db: migrate`. У своїй основній формі вона усього лише запускає метод `change` або `up` для усіх міграцій, які ще не були запущені. Якщо таких міграцій немає, вона виходить. Вона запустить ці міграції в порядку, заснованому на даті міграції.

Зверніть увагу, що запуск завдання `db: migrate` також викликає завдання `db : schema: dump`, яка оновлює ваш файл `db/schema.rb` відповідно до структури вашої бази даних.

Якщо ви визначите цільову версію, Active Record запустить необхідні міграції (методи up, down або change), поки не досягне необхідної версії. Версія це числовий префікс у файлу міграції. Наприклад, щоб мігрувати до версії 20190206120000, запустіть:

```
$ bin/rails db: migrate VERSION=20190206120000
```

Якщо версія 20190206120000 більше поточної версії (тобто міграція вперед) це запустить метод `change` (чи `up`) для усіх міграцій до і включаючи 20190206120000, але не виконає які-небудь пізні міграції. Якщо міграція назад, це запустить метод `down` для усіх міграцій до, але не включаючи, 20190206120000.

4.1. Відкат

Звичайне завдання - це відкотити останню міграцію. Наприклад, ви зробили помилку і хочете виправити її. Можна відстежити версію попередньої міграції і провести міграцію до неї, але можна поступити простіше, запустивши:

```
$bin/rails db: rollback
```

Це поверне ситуацію до останньої міграції, або обернувши метод `change`, або запустивши метод `down`. Якщо треба відмінити декілька міграцій, можна вказати параметр `STEP` :

```
$bin/rails db:rollback STEP=3
```

станеться відкат на 3 останніх міграції.

Завдання `db : migrate: redo` це ярлик для виконання відкату, а потім запуску міграції знову. Так само, як і із завданням `db : rollback` можна вказати параметр `STEP`, якщо треба працювати більш ніж з однією версією, наприклад:

```
$bin/rails db:migrate:redo STEP=3
```

Жодне з цих завдань bin/rails не може зробити нічого такого, чого не можна було б зробити з `db: migrate`. Вони просто зручніші, оскільки вам не треба явно вказувати версію міграції, до якої треба мігрувати.

4.2. Установка бази даних

Завдання `rails db:setup` створить базу даних, завантажить схему і ініціалізує її за допомогою даних seed.

4.3. Скидання бази даних

Завдання `rails db:reset` видалить базу даних і встановить її наново. Функціонально це еквівалентне `rails db:drop db:setup`.

Це не те ж саме, що запуск усіх міграцій. Воно використовує тільки поточний вміст файлу `db/schema.rb` або `db/structure.sql`. Якщо міграція не може відкотити, `rails db: reset` може не допомогти вам. Детальніше про вивантаження схеми дивіться розділ Вивантаження схеми.

4.4. Запуск певних міграцій

Якщо вам треба запустити певну міграцію (up або down), завдання `db : migrate: up` і `db : migrate: down` зроблять це. Просто визначте відповідний варіант і у відповідної міграції буде викликаний метод `change`, `up` або `down`, наприклад:

```
$ bin/rails db: migrate: up VERSION=20190206120000
```

запустить метод `up` у міграції 20190206120000. Це завдання спершу перевірить, чи була міграція вже виконана, і нічого робити не буде, якщо Active Record вважає, що вона вже була запущена.

4.5. Запуск міграцій в різних середовищах

По замовчуванню запуск `bin/rails db : migrate` запуститься в оточенні `development`. Для запуску міграцій в іншому оточенні, його можна вказати, використовуючи змінну середовища `RAILS_ENV` при запуску команди. Наприклад, для запуску міграцій в середовищі `test`, слід запустити:

```
$ bin/rails db: migrate RAILS_ENV=test
```

4.6. Зміна виведення результату запущених міграцій

По замовчуванню міграції говорять нам тільки те, що вони роблять, і скільки часу це зайняло. Міграція, що створює таблицю і додає індекс, видасть щось на кшталт цього:

```
==  CreateProducts: migrating ================================================
-- create_table(: products)
   -> 0.0028s
==  CreateProducts: migrated (0.0028s)=======================================
```

Деякі методи в міграціях дозволяють вам усе це контролювати:

Метод	Призначення
suppress_messages	Приймає блок як аргумент і забороняє будь-який вивід, згенерований цим блоком.
say	Приймає повідомлення як аргумент і виводить його як є. Може бути переданий другий булевий аргумент для вказівки, потрібний відступ або ні.
say_with_time	Виводить текст разом з тривалістю виконання блоку. Якщо блок повертає число, передбачається, що ця кількість рядків, що торкнулися.

Наприклад, ця міграція:

```
class CreateProducts < ActiveRecord::Migration[5.0]
  def change
    suppress_messages do
      create_table: products do |t|
        t.string: name
        t.text: description
        t.timestamps
      end
    end

    say "Created a table"

    suppress_messages {add_index: products,: name}
    say "and an index"!, true

    say_with_time 'Waiting for a while' do
      sleep 10
      250
    end
  end
end
```

згенерує наступний результат

```
==  CreateProducts: migrating ================================================
-- Created a table
   -> and an index!
-- Waiting for a while
   -> 10.0013s
   -> 250 rows
==  CreateProducts: migrated (10.0054s)=======================================
```

Якщо хочете, щоб Active Record нічого не виводив, запуск `rails db : migrate VERBOSE=false` заборонить будь-який вивід.

5. Зміна існуючих міграцій

Періодично ви робитимете помилки при написанні міграції. Якщо ви вже запустили міграцію, ви не зможете просто відредагувати міграцію і запустити її знову: Rails порахує, що він вже виконував міграцію, і нічого не зробить при запуску `rails db : migrate`. Ви повинні відкотити міграцію (наприклад, за допомогою `bin/rails db : rollback`), відредагувати міграцію і потім запустити `rails db: migrate` для запуску виправленої версії.

В цілому, редагування існуючих міграцій не хороша ідея. Ви створите додаткову роботу собі і своїм колегам, і викличете море головного болю, якщо існуюча версія міграції вже була запущена в production. Замість цього, слід написати нову міграцію, що виконує необхідні зміни. Редагування тільки що згенеровані міграції, яка ще не була закоммичена в систему контролю версій (чи, хоч би, не пішла далі за вашу робочу машину) відносно нешкідливо.

Метод `revert` може бути дуже корисним при написанні нової міграції для повернення попередньої в цілому або який те частини (дивіться Повернення до попередніх міграцій

6. Вивантаження схеми

6.1. Для чого потрібні файли схеми?

Міграції, якими б не були вони потужними, не є авторитетним джерелом для вашої схеми бази даних. Це роль дістається або файлу `db/schema.rb`, або файлу SQL, які генерує Active Record при дослідженні бази даних. Вони розроблені не для редагування, вони усього лише відбивають поточний стан бази даних.

Не треба (це може привести до помилки) розгортати новий екземпляр застосування, шляхом відтворення усієї історії міграцій. Набагато простіше і швидше завантажити у базу даних опис поточної схеми.

Наприклад, як створюється тестова база даних: поточна робоча база даних вивантажується (чи в `db/schema.rb`, або в `db/structure.sql`), а потім завантажується в тестову базу даних.

Файли схеми також корисні, якщо хочете швидко поглянути, які атрибути є у об'єкту Active Record. Ця інформація не міститься в коді моделі і часто розмазала по декількох міграціях, але зібрана воєдино у файлі схеми. Є гем annotate_models, який автоматично додає і оновлює коментарі на початку кожної з моделей, що становлять схему, якщо хочете таку функціональність.

6.2. Типи вивантажень схеми

Є два способи вивантажити схему. Вони встановлюються в `config/environment.rb` у властивості `config.active_record.schema_format`, яке може бути або: `sql`, або: `ruby`.

Якщо вибрано: `ruby`, тоді схема зберігається в `db/schema.rb`. Подивившись в цей файл, можна побачити, що він дуже схожий на одну велику міграцію:

```
ActiveRecord::Schema.define(version: 20190206171750) do
  create_table "authors", force: true do |t|
    t.string   "name"
    t.datetime "created_at"
    t.datetime "updated_at"
  end

  create_table "products", force: true do |t|
    t.string   "name"
    t.text     "description"
    t.datetime "created_at"
    t.datetime "updated_at"
    t.string  "part_number"
  end
end
```

У багатьох випадках цього вистачає. Цей файл створюється за допомогою перевірки бази даних і описує свою структуру, використовуючи create_table, add_index і так далі. Оскільки він не залежить від типу бази даних, він може бути завантажений у будь-яку базу даних, підтримувану Active Record. Це дуже корисно, якщо Ви поширюєте застосування, яке може бути запущене на різних базах даних.

`db/schema.rb` не може описати специфічні елементи бази даних, такі як тригери, послідовності, процедури, що зберігаються, обмеження `CHECK` і так далі. Відмітьте, в той час, як в міграціях ви можете виконати довільні вирази SQL, ці вирази не зможуть бути відтворені вивантажувачем схеми. Якщо ви використовуєте подібні особливості, треба встановити формат схеми : `sql`.

Замість використання вивантажувача схеми Active Records, структура бази даних буде вивантажена за допомогою інструменту, призначеного для цієї бази даних (за допомогою завдання rails `db : structure: dump`) в `db/structure.sql`. Наприклад, для PostgreSQL використовується утиліта `pg_dump`. Для MySQL і MariaDB цей файл міститиме результат `SHOW CREATE TABLE` для різних таблиць.

Завантаження таких схем це просто виконання виразів SQL, що містяться в них. За визначенням створиться точна копія структури бази даних. Використання формату : `sql` схеми, проте, запобігає завантаженню схеми в СУБД іншу, чим використовувалася при її створенні.

6.3. Вивантаження схем і контроль початкового коду

Оскільки вивантаження схем це авторитетне джерело для вашої схеми бази даних, дуже рекомендовано включати їх в контроль початкового коду.

`db/schema.rb` містить число поточної версії бази даних. Це забезпечує виникаючі конфлікти у разі злиття двох гілок, кожна з яких зачіпала схему. Коли таке станеться, виправіть конфлікти вручну, залишивши найбільше число версії.

7. Active Record і посилальна цілісність

Спосіб Active Record вимагає, щоб логіка була в моделях, а не у базі даних. За великим рахунком, функції, такі як тригери або обмеження, які переносять частину логіки назад у базу даних, не використовуються активно.

Валідації, такі як `validates: foreign_key, uniqueness: true`, це один із способів, яким ваші моделі можуть дотримуватися посилальної цілісності. Опція: `dependent` в зв'язках дозволяє моделям автоматично знищувати дочірні об'єкти при знищенні батька. Подібно до усього, що працює на рівні застосування, це не може гарантувати посилальної цілісності, таким чином хтось може додати ще і зовнішні ключі як обмежувачі посилальної цілісності у базі даних.

Хоча Active Record не надає яких-небудь інструментів для роботи безпосередньо з цими функціями, метод `execute` може використовуватися для виконання довільного SQL.

8. Міграції і сіди

Основним призначенням міграції Rails є запуск команд, що послідовно модифікують схему. Міграції також можуть бути використані для додавання або модифікування даних. Це корисно для існуючої бази даних, яку не можна видалити і перестворювати, наприклад для бази даних на production.

```
class AddInitialProducts < ActiveRecord::Migration[5.0]
  def up
    5.times do |i|
      Product.create(name: "Product ##{i}"description: "A product".)
    end
  end

  def down
    Product.delete_all
  end
end
```

Щоб додати первинні дані у базу даних після створення, в Rails є вбудована особливість 'seeds ', яка робить процес швидким і простим. Це особливо корисно при частому перезавантаженні бази даних в середовищах розробки і тестування. Цією особливістю легко почати користуватися: просто заповните `db/seeds.rb` деяким кодом Ruby і запустіть `rails db: seed`:

```
5.times do |i|
  Product.create(name: "Product ##{i}"description: "A product".)
end
```

В основному, це чистіший спосіб настроїти базу даних для порожнього застосування.