



## Урок 4

# ActiveRecord

Отношения в моделях. Генерация тестовых данных. Операции ActiveRecord. Промежуточные таблицы. Счетчики. Колбеки.

### [Отношения в моделях](#)

[Связь belongs\\_to](#)

[Связь has\\_one](#)

[Связь has\\_many](#)

### [Гем FFaker](#)

### [Заполнение базы данных: seed-файл](#)

[Пользователи](#)

[Сообщения и комментарии](#)

### [Операции ActiveRecord](#)

[Извлечения значений столбца](#)

[Фильтрация where](#)

[Поиск записи. Методы find и find\\_by](#)

[Сортировка order](#)

[Ограничение выборки limit и offset](#)

### [Промежуточные таблицы](#)

[Отношения has\\_many :through](#)

### [Полиморфные отношения](#)

### [Счетчики](#)

### [Колбеки](#)

### [Домашнее задание](#)

[Дополнительные материалы](#)

[Используемая литература](#)

# Отношения в моделях

На прошлом уроке мы подготовили базу данных для хранения сообщений (posts), пользователей (users) и комментариев (comments). Чтобы извлекать из нее, например, все сообщения пользователя или все комментарии к сообщению, нужны связанные таблицы. На уровне базы данных связи создаются внешними ключами, в rails – задаются на уровне моделей.

Ruby on Rails предоставляет шесть типов связей:

- belongs\_to;
- has\_one;
- has\_many;
- has\_many :through;
- has\_one :through;
- has\_and\_belongs\_to\_many.

## Связь belongs\_to

Эта связь устанавливает отношение «один к одному»: один экземпляр объявляющей модели «принадлежит» одному экземпляру другой модели. В нашем случае такая связь организуется между сообщениями и пользователями, а также между комментариями и пользователями/сообщениями. У каждого сообщения может быть только один автор. У каждого комментария – один автор и одно сообщение.

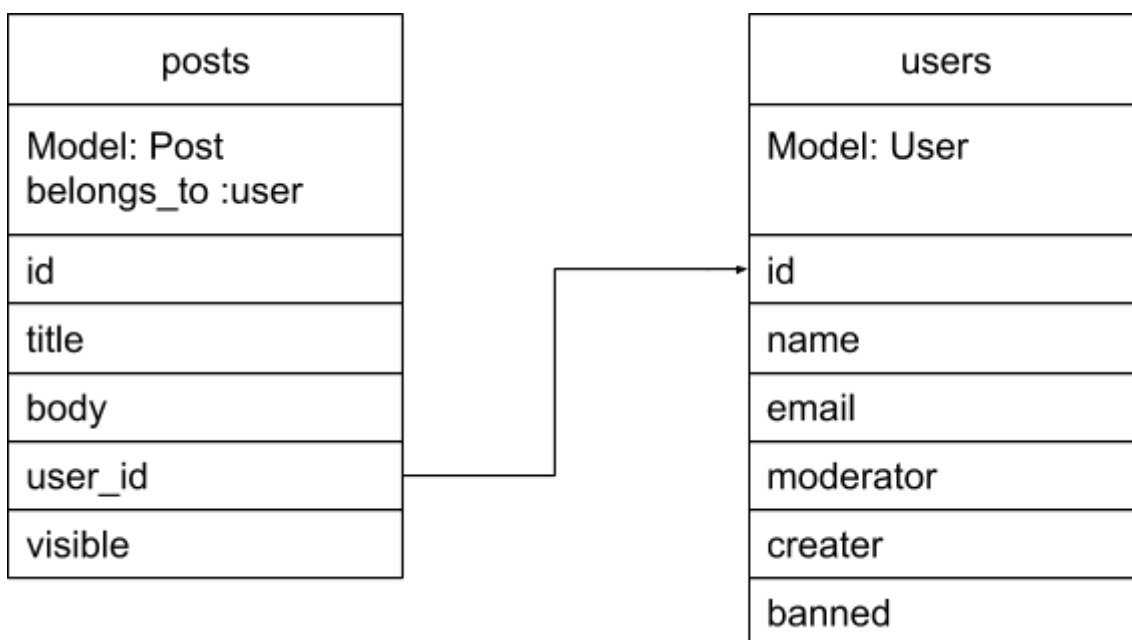


Таблица модели, которая реализует связь belongs\_to, обязана содержать внешний ключ. Например, для случая, приведенного выше, таблица posts должна содержать в своем составе ключ user\_id. Он создается при помощи метода references в миграции.

Связи в Rails создаются в модели при помощи декларативных объявлений. Давайте изменим модели сообщений и комментариев соответствующим образом.

## app/models/post.rb

```
class Post < ApplicationRecord
  belongs_to :user
end
```

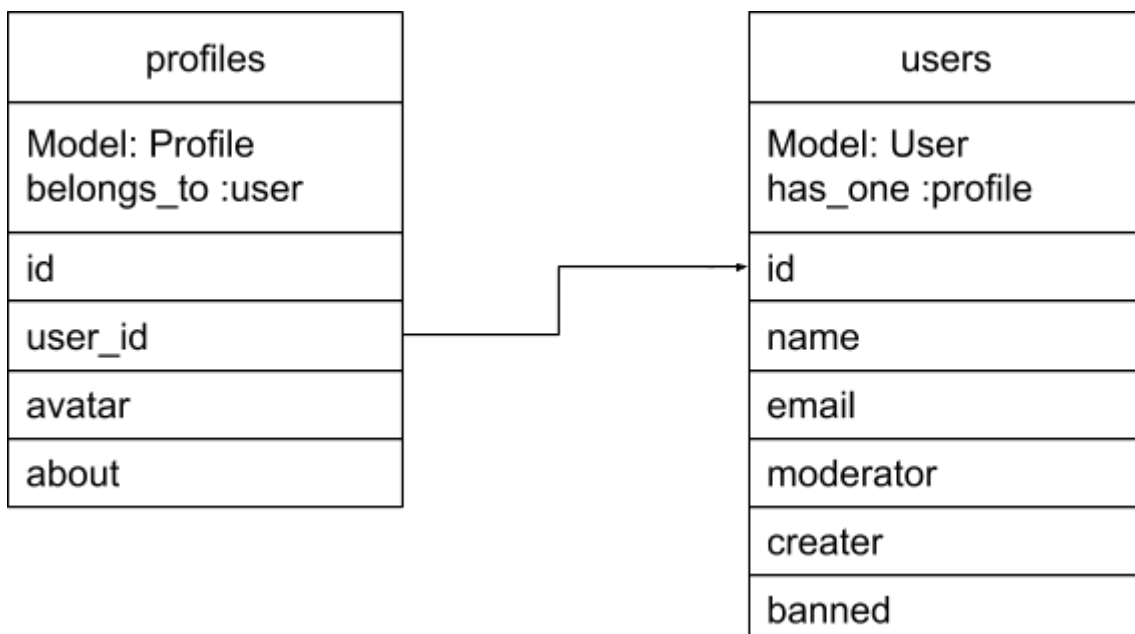
## app/models/comment.rb

```
class Comment < ApplicationRecord
  belongs_to :user
  belongs_to :post
end
```

Такие изменения дают нам возможность извлекать из базы связанные данные. Например, чтобы получить пользователя, создавшего сообщение, достаточно использовать метод `post.user`. Он автоматически делает необходимый запрос к базе данных для извлечения информации о пользователе.

## Связь `has_one`

Данная связь тоже устанавливает соединение «один к одному», но по-другому, и часто используется в паре с `belongs_to`. Эта связь показывает, что каждый экземпляр модели содержит или обладает одним экземпляром другой модели. Допустим, мы вводим отдельную таблицу для профиля пользователя. Тогда каждый профиль «принадлежит» только одному пользователю, и каждый пользователь «владеет» только одним профилем. То есть в модели профиля следует использовать `belongs_to`, а в модели пользователя – `has_one`. Это позволяет нам получать из экземпляра модели пользователя экземпляр профиля с помощью метода `user.profile`.



## Связь `has_many`

Эта связь описывает отношение «один ко многим». Как и `has_one`, она часто работает в паре с `belongs_to`, но используется, когда экземпляр одной модели «владеет» одним или несколькими

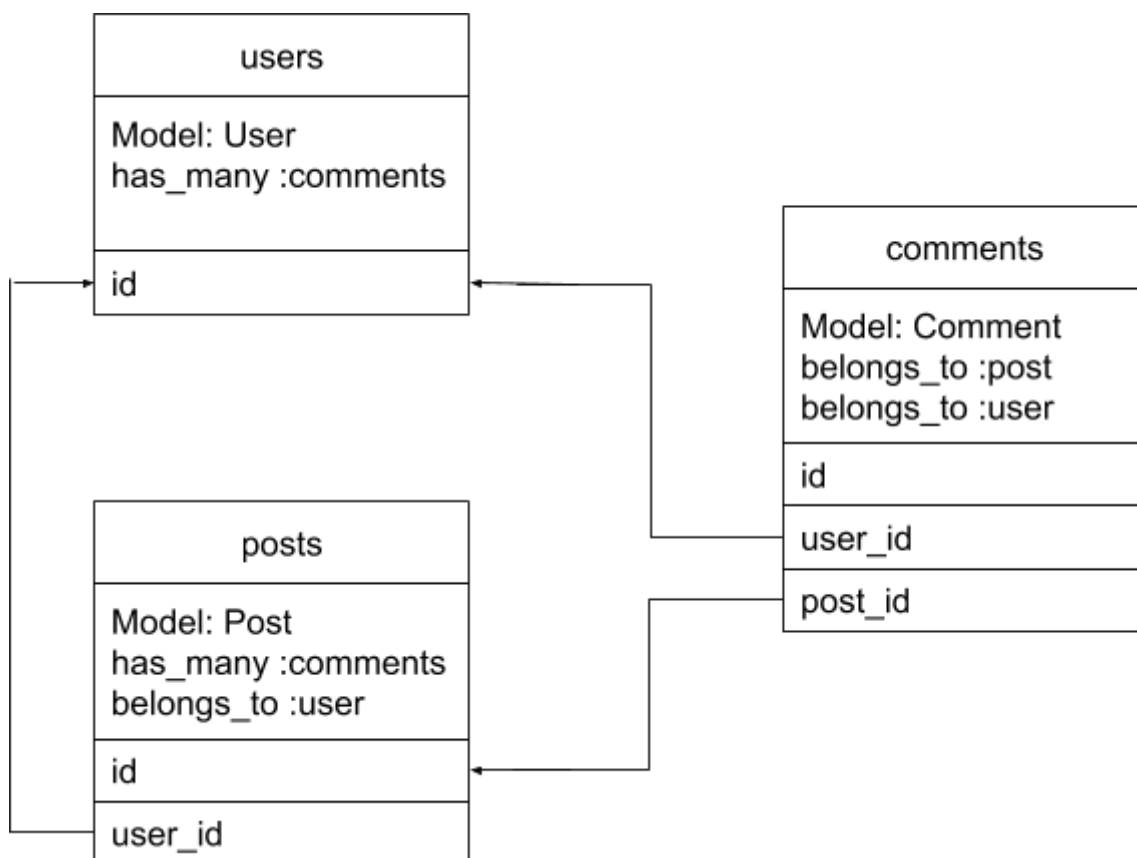
экземплярами другой. В нашем случае это пользователи по отношению к сообщениям и комментариям, и комментарии по отношению к сообщениям.

#### app/models/user.rb

```
class User < ApplicationRecord
  has_many :posts
  has_many :comments
  ...
end
```

#### app/models/post.rb

```
class Post < ApplicationRecord
  belongs_to :user
  has_many :comments
end
```



Связь `has_many` дает нам возможность получить все сообщения пользователя, воспользовавшись методом `user.posts`. Обратите внимание на то, что в `has_many` название модели всегда используется во множественном числе.

# Гем FFaker

Сейчас база не содержит никаких данных. Мы можем заполнить ее вручную, но это утомительно. При разработке тоже иногда необходимо перестроить базу с нуля или очистить таблицу.

Для решения этой задачи часто используется гем FFaker, который генерирует тестовые данные разных типов: параграфы, электронные адреса, имена пользователей и т.п.

Добавим этот гем в файл Gemfile в development- и test-окружение.

## Gemfile

```
...
group :development, :test do
  gem 'ffaker'
  gem 'byebug', platforms: [:mri, :mingw, :x64_mingw]
  gem 'pry-byebug'
end
...
```

Установим добавленный гем с помощью `bundle install`. При выборе методов гемов, можно ориентироваться на страницу <https://github.com/ffaker/ffaker/blob/master/REFERENCE.md>

```
>> FFaker::Internet.safe_email
=> "tanesha.altenwerth@example.com"
>> FFaker::Internet.user_name
=> "clarence_trantow"
>> User.create name: FFaker::Internet.user_name, email:
FFaker::Internet.safe_email
(0.1ms)  BEGIN
User Exists (0.8ms)  SELECT  1 AS one FROM "users" WHERE "users"."name" = $1
LIMIT $2  [["name", "jennifer"], ["LIMIT", 1]]
SQL (12.1ms)  INSERT INTO "users" ("name", "email", "created_at",
"updated_at") VALUES ($1, $2, $3, $4) RETURNING "id"  [["name", "jennifer"],
["email", "eric_orn@example.org"], ["created_at", "2018-01-08 19:45:02.666028"],
["updated_at", "2018-01-08 19:45:02.666028"]]
(12.3ms)  COMMIT
```

## Заполнение базы данных: seed-файл

Чтобы автоматизировать процесс заполнения базы данных, Ruby on Rails предоставляет специальную команду `rails db:seed`, которая выполняет скрипт `db/seeds.rb`.

## Пользователи

Давайте создадим десять пользователей, заполнив их данные при помощи гема FFaker. Для этого в файле `seed.rb` разместим код:

## seeds.rb

```
User.destroy_all

hash_users = 10.times.map do
  {
    name: FFaker::Internet.user_name[0...16],
    email: FFaker::Internet.safe_email
  }
end

users = User.create! hash_users
users.first(7).each { |u| u.update creator: true }
users.first(2).each { |u| u.update moderator: true }
```













В первой строке при помощи метода `destroy_all` удаляются все существовавшие до этого пользователи. Затем при помощи итераторов `times` и `map` формируется массив из 10 хэшей вида:

```
{ name: "duane_leannon", email: "carlo.reinger@example.net" }
```

Метод `user_name` модуля `FFaker::Internet` формирует случайное имя пользователя, а метод `safe_email` — адрес электронной почты. Затем первые семь пользователей помечаются как `creator` и два — как `moderator`. Для выполнения скрипта запустим команду:

```
rails db:seed
```

После этих действий таблица с пользователями должна выглядеть так:

	 id 	 name 	 email 	 moderator 	 creator 	 banned 
1	1	lynwood.toy	reggie_gleichner@example.org	true	true	false
2	2	karoline	penny_grady@example.org	true	true	false
3	3	ricardo	buster_jones@example.org	false	true	false
4	4	marguerite	trudy@example.org	false	true	false
5	5	tashia.kunze	deon_barrows@example.com	false	true	false
6	6	camille.weber	alva@example.org	false	true	false
7	7	florencio	caleb.hudson@example.org	false	true	false
8	8	joel.funk	ladawn_mosciski@example.net	false	false	false
9	9	nadine.jones	terra@example.org	false	false	false
10	10	rana_parisian	darlena@example.org	false	false	false

## Сообщения и комментарии

Аналогичным образом можно создать сообщения и комментарии.

## seeds.rb

```
User.destroy_all
```

```

Post.destroy_all
Comment.destroy_all

hash_users = 10.times.map do
  {
    name: FFaker::Internet.user_name[0...16],
    email: FFaker::Internet.safe_email
  }
end
users = User.create! hash_users
users.first(7).each { |u| u.update creator: true }
users.first(2).each { |u| u.update moderator: true }

creators = User.where(creator: true)
hash_posts = 20.times.map do
  {
    title: FFaker::HipsterIpsum.paragraph,
    body: FFaker::HipsterIpsum.paragraphs,
    user: creators.sample
  }
end

posts = Post.create! hash_posts
hash_comments = 200.times.map do
  {
    body: FFaker::HipsterIpsum.paragraphs,
    user: users.sample,
    post: posts.sample
  }
end
Comment.create! hash_comments

```

## Операции ActiveRecord

ActiveRecord предоставляет большое количество разнообразных методов, которые фильтруют, сортируют и ограничивают выборку. Рассмотрим наиболее популярные из них. Все операции выполним в rails-консоли, запустить которую можно при помощи команды rails c.

### Извлечения значений столбца

При использовании метода all результирующий массив содержит объекты модели, ориентироваться в которых довольно сложно.

```

>> User.all
  User Load (0.3ms)  SELECT "users".* FROM "users" LIMIT $1  [["LIMIT", 11]]
=> #<ActiveRecord::Relation [#<User id: 58, name: "leana.miller", email:
"juliane@example.org", moderator: false, creator: false, banned: false,
created_at: "2018-01-08 20:29:16", updated_at: "2018-01-08 20:29:16">, #<User
id: 59, name: "elayne", email: "arie@example.net", moderator: false, creator:
false, banned: false, created_at: "2018-01-08 20:29:16", updated_at: "2018-01-08

```



```
20:29:16">, #<User id: 60, name: "donna.hudson", email:
"wally_waelchi@example.org", moderator: false, creator: false, banned: false,
created_at: "2018-01-08 20:29:16", updated_at: "2018-01-08 20:29:16">, #<User
id: 53, name: "floretta", email: "federico@example.net", moderator: false,
creator: true, banned: false, created_at: "2018-01-08 20:29:16", updated_at:
"2018-01-08 20:29:16">, #<User id: 54, name: "ashanti", email:
"yadira_runolfsdottir@example.net", moderator: false, creator: true, banned:
false, created_at: "2018-01-08 20:29:16", updated_at: "2018-01-08 20:29:16">,
#<User id: 55, name: "gilbert", email: "margit@example.org", moderator: false,
creator: true, banned: false, created_at: "2018-01-08 20:29:16", updated_at:
"2018-01-08 20:29:16">, #<User id: 56, name: "masako_aufderhar", email:
"lorea@example.org", moderator: false, creator: true, banned: false,
created_at: "2018-01-08 20:29:16", updated_at: "2018-01-08 20:29:16">, #<User
id: 57, name: "viva_ferry", email: "willie@example.net", moderator: false,
creator: true, banned: false, created_at: "2018-01-08 20:29:16", updated_at:
"2018-01-08 20:29:16">, #<User id: 51, name: "maya", email:
"ophelia@example.com", moderator: true, creator: true, banned: false,
created_at: "2018-01-08 20:29:16", updated_at: "2018-01-08 20:29:16">, #<User
id: 52, name: "sena_aufderhar", email: "keesha.gusikowski@example.com",
moderator: true, creator: true, banned: false, created_at: "2018-01-08
20:29:16", updated_at: "2018-01-08 20:29:16">]>
```

Поэтому для анализа выборки часто прибегают к методу `pluck`, который позволяет выбрать лишь несколько столбцов. Имена столбцов передаются в виде списка символов:

```
>> User.pluck(:name)
(0.5ms) SELECT "users"."name" FROM "users"
=> ["leana.miller", "elayne", "donna.hudson", "floretta", "ashanti", "gilbert",
"masako_aufderhar", "viva_ferry", "maya", "sena_aufderhar"]
>> User.pluck(:id, :name)
(0.5ms) SELECT "users"."id", "users"."name" FROM "users"
=> [[58, "leana.miller"], [59, "elayne"], [60, "donna.hudson"], [53,
"floretta"], [54, "ashanti"], [55, "gilbert"], [56, "masako_aufderhar"], [57,
"viva_ferry"], [51, "maya"], [52, "sena_aufderhar"]]
```

## Фильтрация where

По умолчанию методы `all` или `pluck` извлекают абсолютно все строки таблицы. Чтобы ограничить выборку, отфильтруем результаты поиска методом `where`.

```
>> User.pluck(:name, :moderator)
(0.4ms) SELECT "users"."name", "users"."moderator" FROM "users"
=> [["leana.miller", false], ["elayne", false], ["donna.hudson", false],
["floretta", false], ["ashanti", false], ["gilbert", false],
["masako_aufderhar", false], ["viva_ferry", false], ["maya", true],
["sena_aufderhar", true]]
>> User.where(moderator: true).pluck(:name)
(0.3ms) SELECT "users"."name" FROM "users" WHERE "users"."moderator" = $1
[["moderator", "t"]]
```

```
=> ["maya", "sena_aufderhar"]
```

В качестве фильтруемых могут выступать не только скалярные значения, но и массивы:

```
>> User.where(name: ['viva_ferry', 'maya', 'sena_aufderhar']).pluck(:id)
(0.6ms)  SELECT "users"."id" FROM "users" WHERE "users"."name" IN
('viva_ferry', 'maya', 'sena_aufderhar')
=> [57, 51, 52]
```

Применять метод `where` можно не только к моделям, но и к отношениям. Для демонстрации извлечем первое сообщение и его автора:

```
>> post = Post.first
Post Load (0.6ms)  SELECT "posts".* FROM "posts" ORDER BY "posts"."id" ASC
LIMIT $1  [["LIMIT", 1]]
=> #<Post id: 1, title: "Yr chambray skateboard hoodie you probably haven't...",
body: "[\"You probably haven't heard of them Wayfarers iro...", user_id: 53,
visible: false, created_at: "2018-01-08 20:29:16", updated_at: "2018-01-08
20:29:16">
>> user = post.user
User Load (0.5ms)  SELECT "users".* FROM "users" WHERE "users"."id" = $1
LIMIT $2  [["id", 53], ["LIMIT", 1]]
=> #<User id: 53, name: "floretta", email: "federico@example.net", moderator:
false, creator: true, banned: false, created_at: "2018-01-08 20:29:16",
updated_at: "2018-01-08 20:29:16">
>> user.name
=> "floretta"
```

Как видно, метод `user` возвращает не просто `id` пользователя, а экземпляр модели пользователя. С ним можно работать как с любым другим, например, посмотреть комментарии от имени пользователя.

```
>> user.comments
Comment Load (7.8ms)  SELECT "comments".* FROM "comments" WHERE
"comments"."user_id" = $1 LIMIT $2  [["user_id", 53], ["LIMIT", 11]]
=> #<ActiveRecord::Associations::CollectionProxy [#<Comment id: 4, body:
"[\"Chambray raw denim leggings banh mi synth wolf e...", user_id: 53, post_id:
10, visible: false, created_at: "2018-01-08 20:29:16", updated_at: "2018-01-08
20:29:16">, #<Comment id: 24, body: "[\"Butcher etsy artisan tofu Shoreditch
retro keffi...", user_id: 53, post_id: 8, visible: false, created_at:
"2018-01-08 20:29:16", updated_at: "2018-01-08 20:29:16">, #<Comment id: 36,
body: "[\"Dreamcatcher freegan twee leggings trust fund. C...", user_id: 53,
post_id: 1, visible: false, created_at: "2018-01-08 20:29:16", updated_at:
"2018-01-08 20:29:16">, #<Comment id: 41, body: "[\"Artisan mlkshk party 8-bit
tumblr Carles cardiga...", user_id: 53, post_id: 17, visible: false, created_at:
"2018-01-08 20:29:16", updated_at: "2018-01-08 20:29:16">, #<Comment id: 45,
body: "[\"Lo-fi banh mi artisan craft beer readymade synth...", user_id: 53,
post_id: 4, visible: false, created_at: "2018-01-08 20:29:16", updated_at:
"2018-01-08 20:29:16">, #<Comment id: 69, body: "[\"Messenger bag irony before
```

```
they sold out Rerry R...", user_id: 53, post_id: 9, visible: false, created_at:
"2018-01-08 20:29:17", updated_at: "2018-01-08 20:29:17">, #<Comment id: 70,
body: "[\"Tumblr fap brunch party twee artisan aesthetic. ...\", user_id: 53,
post_id: 3, visible: false, created_at: "2018-01-08 20:29:17", updated_at:
"2018-01-08 20:29:17">, #<Comment id: 87, body: "[\"Leggings party cliché
single-origin coffee Rerry...", user_id: 53, post_id: 10, visible: false,
created_at: "2018-01-08 20:29:17", updated_at: "2018-01-08 20:29:17">, ...]>
```

Если нас интересует количество комментариев, для их подсчета можно воспользоваться методами `count` или `size`.

```
>> user.comments.count
(0.5ms)  SELECT COUNT(*) FROM "comments" WHERE "comments"."user_id" = $1
[["user_id", 53]]
=> 18
```

## Поиск записи. Методы `find` и `find_by`

Следует помнить, что данные, которые извлекаются методами `all` или `where`, всегда представляют собой массив, даже если в нем лишь один элемент. Поэтому обратиться к нему мы можем только после извлечения элемента при помощи квадратных скобок:

```
>> Post.where(id: 1)
Post Load (0.4ms)  SELECT "posts".* FROM "posts" WHERE "posts"."id" = $1
LIMIT $2 [["id", 1], ["LIMIT", 11]]
=> #<ActiveRecord::Relation [#<Post id: 1, title: "Yr chambray skateboard hoodie
you probably haven't...", body: "[\"You probably haven't heard of them Wayfarers
iro...", user_id: 53, visible: false, created_at: "2018-01-08 20:29:16",
updated_at: "2018-01-08 20:29:16">]>
bundle :038 > Post.where(id: 1).title
NoMethodError: Post Load (0.4ms)  SELECT "posts".* FROM "posts" WHERE
"posts"."id" = $1 LIMIT $2 [["id", 1], ["LIMIT", 11]]
undefined method `title' for #<Post::ActiveRecord_Relation:0x00007f9ce446f798>
from (irb):38
>> Post.where(id: 1)[0].title
Post Load (0.3ms)  SELECT "posts".* FROM "posts" WHERE "posts"."id" = $1
[["id", 1]]
=> "Yr chambray skateboard hoodie you probably haven't heard of them
single-origin coffee freegan next level. Portland jean shorts beard
single-origin coffee trust fund. Cred high life seitan +1 keffiyeh whatever
tattooed cliché Banksy."
```

`ActiveRecord` предоставляет несколько удобных методов, которые возвращают сразу объект модели (не массив). Например, это методы `first` и `last`, которые возвращают первую и последнюю запись в выборке. Однако наиболее популярным является метод `find`, который позволяет извлечь запись по ее первичному ключу. Например:

```
>> p = Post.find(1)
```

```

Post Load (0.5ms)  SELECT  "posts".* FROM "posts" WHERE "posts"."id" = $1
LIMIT $2  [["id", 1], ["LIMIT", 1]]
=> #<Post id: 1, title: "Yr chambray skateboard hoodie you probably haven't...",
body: "[\"You probably haven't heard of them Wayfarers iro...", user_id: 53,
visible: false, created_at: "2018-01-08 20:29:16", updated_at: "2018-01-08
20:29:16">
>> p.title
=> "Yr chambray skateboard hoodie you probably haven't heard of them
single-origin coffee freegan next level. Portland jean shorts beard
single-origin coffee trust fund. Cred high life seitan +1 keffiyeh whatever
tattooed cliché Banksy."

```

Для извлечения объекта модели при помощи значения, отличного от первичного ключа, удобно использовать метод `find_by`. Поиск по имени пользователя выглядит так:

```

>> User.find_by(name: 'maya')
User Load (0.4ms)  SELECT  "users".* FROM "users" WHERE "users"."name" = $1
LIMIT $2  [["name", "maya"], ["LIMIT", 1]]
=> #<User id: 51, name: "maya", email: "ophelia@example.com", moderator: true,
creator: true, banned: false, created_at: "2018-01-08 20:29:16", updated_at:
"2018-01-08 20:29:16">

```

## Сортировка order

Записи в таблице хранятся без определенного порядка. Отсортировать выборку можно при помощи метода `order`:

```

>> User.pluck(:name)
(0.5ms)  SELECT "users"."name" FROM "users"
=> ["leana.miller", "elayne", "donna.hudson", "floretta", "ashanti", "gilbert",
"masako_aufderhar", "viva_ferry", "maya", "sena_aufderhar"]
>> User.order(:name).pluck(:name)
(0.9ms)  SELECT "users"."name" FROM "users" ORDER BY "users"."name" ASC
=> ["ashanti", "donna.hudson", "elayne", "floretta", "gilbert", "leana.miller",
"masako_aufderhar", "maya", "sena_aufderhar", "viva_ferry"]
>> User.order(name: :desc).pluck(:name)
(0.5ms)  SELECT "users"."name" FROM "users" ORDER BY "users"."name" DESC
=> ["viva_ferry", "sena_aufderhar", "maya", "masako_aufderhar", "leana.miller",
"gilbert", "floretta", "elayne", "donna.hudson", "ashanti"]

```

Для сортировки в обратном порядке используется хэш вида `{name: :desc}`.

## Ограничение выборки limit и offset

Для ограничения выборки по количеству строк используется метод `limit`, который позволяет задать количество извлекаемых записей. Метод `offset` позволяет задать позицию, начиная с которой должны извлекаться записи:

```
>> User.pluck(:name)
(0.5ms) SELECT "users"."name" FROM "users"
=> ["leana.miller", "elayne", "donna.hudson", "floretta", "ashanti", "gilbert",
"masako_aufderhar", "viva_ferry", "maya", "sena_aufderhar"]
>> User.limit(3).pluck(:name)
(0.3ms) SELECT "users"."name" FROM "users" LIMIT $1 [{"LIMIT", 3}]
=> ["leana.miller", "elayne", "donna.hudson"]
>> User.offset(3).limit(3).pluck(:name)
(0.3ms) SELECT "users"."name" FROM "users" LIMIT $1 OFFSET $2 [{"LIMIT",
3}, [{"OFFSET", 3}]]
=> ["floretta", "ashanti", "gilbert"]
```

## Промежуточные таблицы

Предыдущие варианты связей предполагали отношения «один к одному» или «один ко многим». Однако на практике часто встречается отношение «многие ко многим». Например, каждый пользователь может прокомментировать несколько сообщений, а сами сообщения могут иметь комментарии нескольких пользователей.

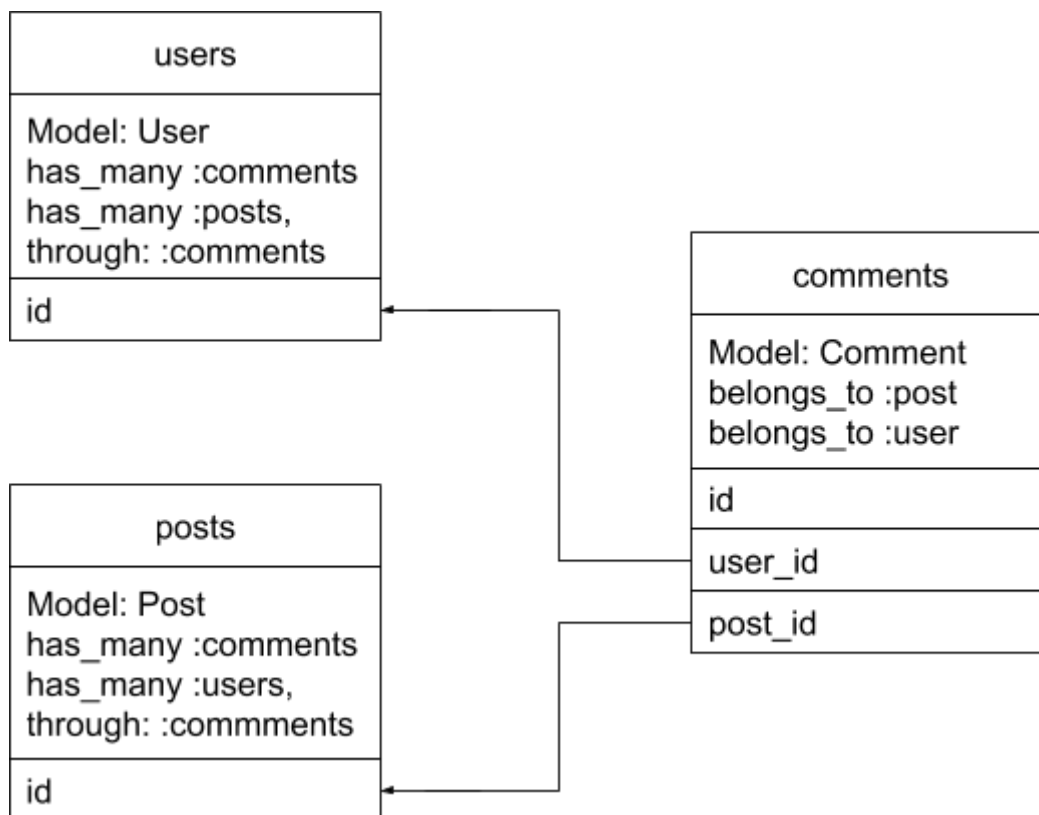
Для реализации отношения «многие ко многим» в реляционных базах данных используется промежуточная таблица. В Ruby on Rails существует три специальных отношения, которые реализуют этот подход:

- **Has\_and\_belongs\_to\_many** – две таблицы с моделями связываются через промежуточную таблицу, для которой не предусмотрено модели.
- **Has\_many :through** – две таблицы связываются через промежуточную таблицу. Для всех трех таблиц предусмотрены модели.
- **Has\_one :through** – две таблицы связываются через промежуточную таблицу, в которой вместо отношения «многие ко многим» организуется связь «один ко многим».

Отношение `has_and_belongs_to_many` – исторически первая конструкция. В большинстве случаев вместо нее лучше использовать более гибкую ассоциацию `has_many :through`. Отношение `has_one :through`, полностью равно по возможностям `has_many :through`, однако реализует связь «один ко многим».

### Отношение `has_many :through`

Добавим в модель пользователя список прокомментированных им сообщений (`posts`), а в модель сообщений – список пользователей (`users`), оставивших комментарии.



Чтобы добиться корректных связей, поправим модель сообщений:

**app/models/post.rb**

```
class Post < ApplicationRecord
  belongs_to :user
  has_many :comments
  has_many :users, through: :comments
end
```

Belongs\_to создает связь с пользователем, создавшим сообщение. Has\_many создает связь со списком пользователей, прокомментировавших сообщение. При помощи ключа through указывается промежуточная таблица comments. Так как внутри промежуточной таблицы предусмотрены внешние ключи user\_id и comment\_id, Ruby on Rails может установить связь между текущим сообщением и таблицей users.

```
>> p = Post.find(5)
Post Load (0.5ms) SELECT "posts".* FROM "posts" WHERE "posts"."id" = $1
LIMIT $2 [["id", 5], ["LIMIT", 1]]
=> #<Post id: 5, title: "Brooklyn food truck vegan twee trust fund vinyl
Co...", body: "[\"Bicycle rights raw denim messenger bag iPhone si...", user_id:
53, visible: false, created_at: "2018-01-08 20:29:16", updated_at: "2018-01-08
20:29:16">
>> p.user.name
User Load (0.4ms) SELECT "users".* FROM "users" WHERE "users"."id" = $1
LIMIT $2 [["id", 53], ["LIMIT", 1]]
=> "floretta"
>> p.users.pluck(:name)
```

```
(0.5ms) SELECT "users"."name" FROM "users" INNER JOIN "comments" ON
"users"."id" = "comments"."user_id" WHERE "comments"."post_id" = $1
[["post_id", 5]]
=> ["masako_aufderhar", "donna.hudson", "viva_ferry", "ashanti",
"sena_aufderhar", "masako_aufderhar", "ashanti", "viva_ferry", "gilbert",
"donna.hudson", "donna.hudson", "gilbert"]
```

Однако стоит нам изменить название связи, например вместо users использовать commentators, как мы вынуждены указать источник данных при помощи ключевого слова source.

#### app/models/post.rb

```
class Post < ApplicationRecord
  belongs_to :user
  has_many :comments
  has_many :commentators, through: :comments, source: :user
end
```

Чем более нестандартно называются внешние ключи, имена классов моделей, тем больше усилий нужно, чтобы связать таблицы. Поэтому, чтобы сохранить компактность и читаемость кода, важно следовать соглашениям, принятым в Ruby on Rails.

Пока связь «многие ко многим» построена только в одну сторону. Чтобы иметь возможность извлечь список статей, прокомментированных каждым пользователем, следует добавить связь has\_many :commented\_posts через промежуточную таблицу comments.

#### app/models/user.rb

```
class User < ApplicationRecord
  has_many :posts
  has_many :comments
  has_many :commented_posts, through: :comments, source: :post

  validates :name, presence: true
  validates :name, length: { maximum: 16, minimum: 2 }
  validates :name, uniqueness: true
end
```

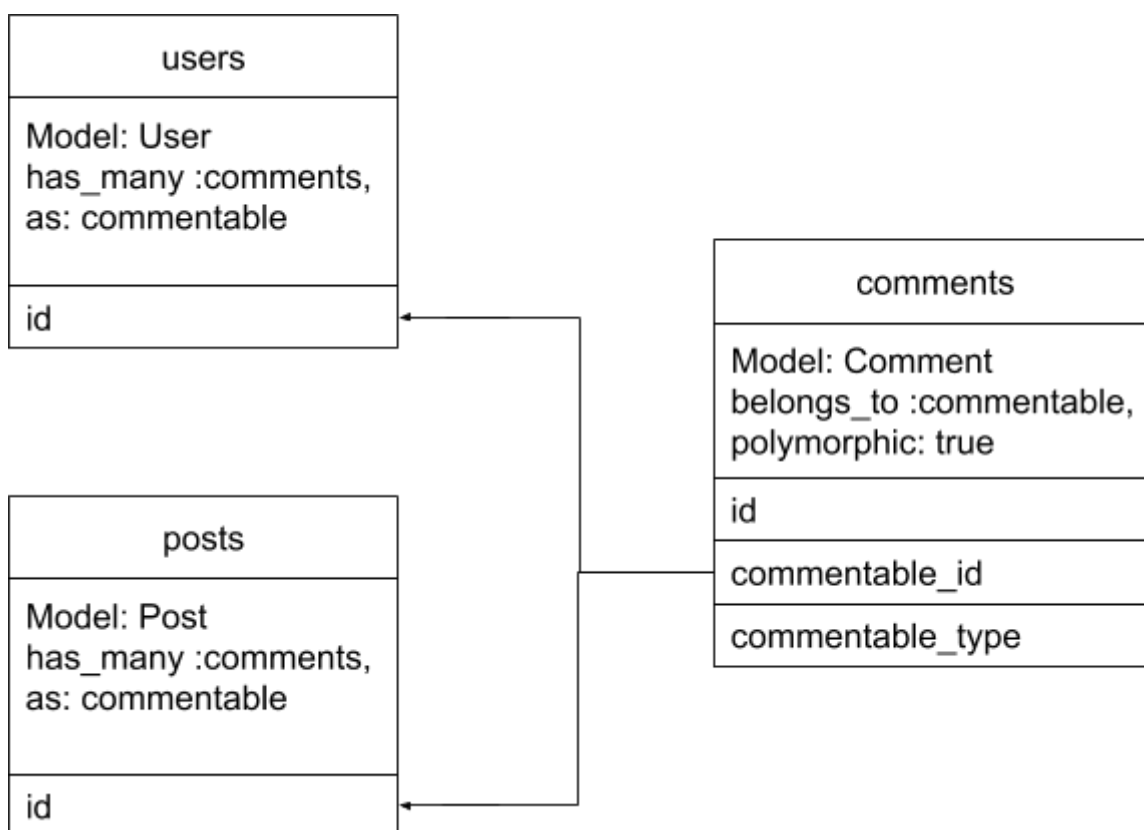
Теперь можно извлечь список сообщений, которые пользователь прокомментировал:

```
>> u = User.find(56)
  User Load (0.5ms) SELECT "users".* FROM "users" WHERE "users"."id" = $1
LIMIT $2 [["id", 56], ["LIMIT", 1]]
=> #<User id: 56, name: "masako_aufderhar", email: "lolean@example.org",
moderator: false, creator: true, banned: false, created_at: "2018-01-08
20:29:16", updated_at: "2018-01-08 20:29:16">
>> u.commented_posts.size
  (0.7ms) SELECT COUNT(*) FROM "posts" INNER JOIN "comments" ON "posts"."id" =
"comments"."post_id" WHERE "comments"."user_id" = $1 [["user_id", 56]]
```

## Полиморфные отношения

Полиморфные связи предназначены для того, чтобы одной моделью обслуживать несколько однотипных задач. Например, мы хотим, чтобы в приложении была возможность комментировать сообщения и страницы пользователей. Нет смысла заводить под эти комментарии отдельные таблицы и модели и обслуживать их отдельными контроллерами и представлениями. Более экономно и правильно использовать одну таблицу `comments` для комментирования как сообщений, так и страниц пользователей.

Для этого предназначены полиморфные связи, которые организуются при помощи двух столбцов, например `commentable_id` и `commentable_type`. Первый столбец содержит внешний ключ комментируемой сущности, второй – тип.



Для демонстрации такого типа отношений преобразим таблицу `comments`. Создадим миграцию:

```
rails g migration AddCommentableToComments commentable:references{polymorphic}
```

Генератор создаст миграцию:

```
class AddCommentableToComments < ActiveRecord::Migration[5.1]
  def change
```



```
    add_reference :comments, :commentable, polymorphic: true
  end
end
```

Добавим в миграцию удаление старого внешнего ключа `post_id`, так как теперь его роль будет играть `commentable_id`:

```
class AddCommentableToComments < ActiveRecord::Migration[5.1]
  def change
    add_reference :comments, :commentable, polymorphic: true
    remove_reference :comments, :post
  end
end
```

После выполнения миграции, необходимо внести изменения в модели, установив полиморфную связь:

#### **app/models/comment.rb**

```
class Comment < ApplicationRecord
  belongs_to :commentable, polymorphic: true
end
```

#### **app/models/post.rb**

```
class Post < ApplicationRecord
  belongs_to :user
  has_many :comments, as: :commentable
  has_many :commentators, through: :comments, source: :user
end
```

#### **app/models/user.rb**

```
class User < ApplicationRecord
  has_many :posts
  has_many :comments
  has_many :commented_posts, through: :comments, source: :commentable,
source_type: :Post
  has_many :commented_users, through: :comments, source: :commentable,
source_type: :User

  validates :name, presence: true
  validates :name, length: { maximum: 16, minimum: 2 }
  validates :name, uniqueness: true
end
```

Изменение структуры моделей требует внести изменения в seed-файл, относящийся к комментариям:

**db/seed.rb**

```
User.destroy_all
Post.destroy_all
Comment.destroy_all
...
hash_commentaries = 200.times.map do
  commentable = ((rand(2) == 1) ? posts : users).sample
  {
    body: FFaker::HipsterIpsum.paragraphs,
    user: users.sample,
    commentable_id: commentable.id,
    commentable_type: commentable.class.to_s
  }
end
Comment.create! hash_commentaries
```

Теперь можно выполнить команду пересоздания базы данных:

```
rails db:drop db:create db:migrate db:seed
```

То же самое одной командой:

```
rails db:reset
```

## Счетчики

Часто возникает необходимость в подсчете количества связанных отношением записей. Для этих целей можно использовать метод `size` или `count`. Однако операция подсчета довольно трудоемка. Поэтому ассоциации предоставляют возможность кэшировать результаты в специальных столбцах-счетчиках с суффиксом `_count`. Если при этом связь `belongs_to` имеет параметр `counter_cache`, ActiveRecord самостоятельно позаботится об увеличении или уменьшении значения счетчика при создании или удалении записи.

Попробуем учесть количество сообщений, которые оставляют пользователи, и сохранить их в специальном столбце `comments_count`. Для этого добавим его в таблицу `users` при помощи генератора:

```
rails g migration AddCommentsCountToUsers comments_count:integer
invoke active_record
createdb/migrate/20180114184833_add_comments_count_to_users.rb
```

В результате создается миграция, которую следует применить к базе данных при помощи команды `rails db:migrate`.

```
class AddCommentsCountToUsers < ActiveRecord::Migration[5.1]
  def change
    add_column :users, :comments_count, :integer
  end
end
```

В модель Comment в вызов belongs\_to :user следует добавить параметр counter\_cache: true:

```
class Comment < ApplicationRecord
  belongs_to :user, counter_cache: true
  belongs_to :commentable, polymorphic: true
end
```

Если выполним обновление базы данных при помощи rails db:reset или rails db:seed, убедимся, что счетчики comments\_count содержат количество оставленных пользователями комментариев.

```
>> User.pluck(:name, :comments_count)
(0.5ms) SELECT "users"."name", "users"."comments_count" FROM "users"
=> [{"jeanene_schumm", 21}, {"leonore_marquard", 21}, {"altha_lubowitz", 23},
["syble", 14], ["carylon", 18], ["mikel_fahey", 24], ["katia.smitham", 24],
["solange", 21], ["kathy_rosenbaum", 17], ["donte.kub", 17]]
```

Добавление новых комментариев пользователем приводит к автоматическому увеличению счетчика.

## Колбеки

Колбеки – это методы, которые вызываются до или после определенной операции с объектом. Например при создании объекта, сохранении, обновлении, удалении, валидации или загрузки из базы данных.

Для демонстрации колбеков в модель User добавим два вызова: before\_destroy и after\_destroy, которые будут срабатывать непосредственно перед и после удаления пользователя.

**app/models/user.rb**

```
class User < ApplicationRecord
  before_destroy :log_before_destory
  after_destroy :log_after_destory
  ...
  private

  def log_before_destory
    Rails.logger.info "##### Собираемся удалить пользователя #{@name} #####"
  end
  def log_after_destory
    Rails.logger.info "##### Пользователь #{@name} удален #####"

```

Первая попытка удалить пользователя завершится неудачей. Колбек `before_destroy` срабатывает, а до `after_destroy` дело не доходит.

```
>> u = User.find(1)
   User Load (0.4ms)  SELECT  "users".* FROM "users" WHERE "users"."id" = $1
LIMIT $2  [["id", 1], ["LIMIT", 1]]
=> #<User id: 1, name: "donte.kub", email: "celinda_daugherty@example.net",
moderator: true, creator: true, banned: false, created_at: "2018-01-14
18:53:36", updated_at: "2018-01-14 18:53:37", comments_count: 17>
>> u.destroy
   (0.2ms)  BEGIN
##### Собираемся удалить пользователя #####
   SQL (3.7ms)  DELETE FROM "users" WHERE "users"."id" = $1  [["id", 1]]
   (0.2ms)  ROLLBACK
ActiveRecord::InvalidForeignKey: PG::ForeignKeyViolation: ERROR:  update or
delete on table "users" violates foreign key constraint "fk_rails_03de2dc08c" on
table "comments"
DETAIL:  Key (id)=(1) is still referenced from table "comments".
: DELETE FROM "users" WHERE "users"."id" = $1
from (irb):3
```

Ошибка заключается в том, что при удалении пользователя остаются его комментарии и сообщения. Чтобы удаление проходило корректно, в модели `User` следует снабдить ассоциации `has_many :comments` и `has_many :posts` параметром `dependent: :destroy`. Он обеспечит каскадное удаление комментариев и сообщений пользователя.

#### `app/models/user.rb`

```
class User < ApplicationRecord
  before_destroy :log_before_destroy
  after_destroy :log_after_destroy
  ...
  has_many :posts, dependent: :destroy
  has_many :comments, dependent: :destroy
  ...
  private

  def log_before_destroy
    Rails.logger.info "##### Собираемся удалить пользователя #{@name} #####"
  end
  def log_after_destroy
    Rails.logger.info "##### Пользователь #{@name} удален #####"
  end
end
```

После этого удаление происходит корректно, и оба колбека работают успешно:

```
>> u = User.find(1)
```

```

User Load (0.5ms)  SELECT "users".* FROM "users" WHERE "users"."id" = $1 LIMIT
$2 [["id", 1], ["LIMIT", 1]]
>> u.destroy
      (0.2ms)  BEGIN
##### Собираемся удалить пользователя #####
      Post Load (0.4ms)  SELECT "posts".* FROM "posts" WHERE "posts"."user_id" = $1
[["user_id", 1]]
      SQL (0.4ms)  DELETE FROM "posts" WHERE "posts"."id" = $1  [["id", 11]]
      Comment Load (0.8ms)  SELECT "comments".* FROM "comments" WHERE
"comments"."user_id" = $1  [["user_id", 1]]
      SQL (0.3ms)  DELETE FROM "comments" WHERE "comments"."id" = $1  [["id", 7]]
      SQL (0.2ms)  DELETE FROM "comments" WHERE "comments"."id" = $1  [["id", 21]]
      SQL (0.3ms)  DELETE FROM "comments" WHERE "comments"."id" = $1  [["id", 189]]
      SQL (0.9ms)  DELETE FROM "users" WHERE "users"."id" = $1  [["id", 1]]
##### Пользователь удален #####
      (0.6ms)  COMMIT

```

## Домашнее задание

1. Сформируйте ActiveRecord-запрос для получения всех постов, написанных модераторами.
2. \* На страницах в META-информации часто выводятся ключевые слова (keywords), описание страницы (description) и заголовок страницы (title). Познакомьтесь с HTML-тэгами, которые обеспечивают вывод этой информации. Кроме того, подготовьте таблицу Seo и соответствующую модель Seo, которая обеспечит сохранение этой информации. Свяжите модель Seo с сообщениями и пользователями полиморфной связью.

«\*» задание повышенной сложности (по желанию).

## Дополнительные материалы

1. [Ген ffaker](#).
2. [Связи в rails](#).
3. [Миграции](#).

## Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. [Связи в rails](#).
2. Майкл Хартл. Ruby on Rails для начинающих. Изучаем разработку веб-приложений на основе Rails.
3. Ruby on Rails [Guides](#).
4. Obie Fernandez. The Rails 5 Way.
5. Джеймс Р. Грофф, Пол Н. Вайнберг, Эндрю Дж. Оппель. SQL. Полное руководство.
6. Билл Карвин. Программирование баз данных SQL. Типичные ошибки и их устранение.