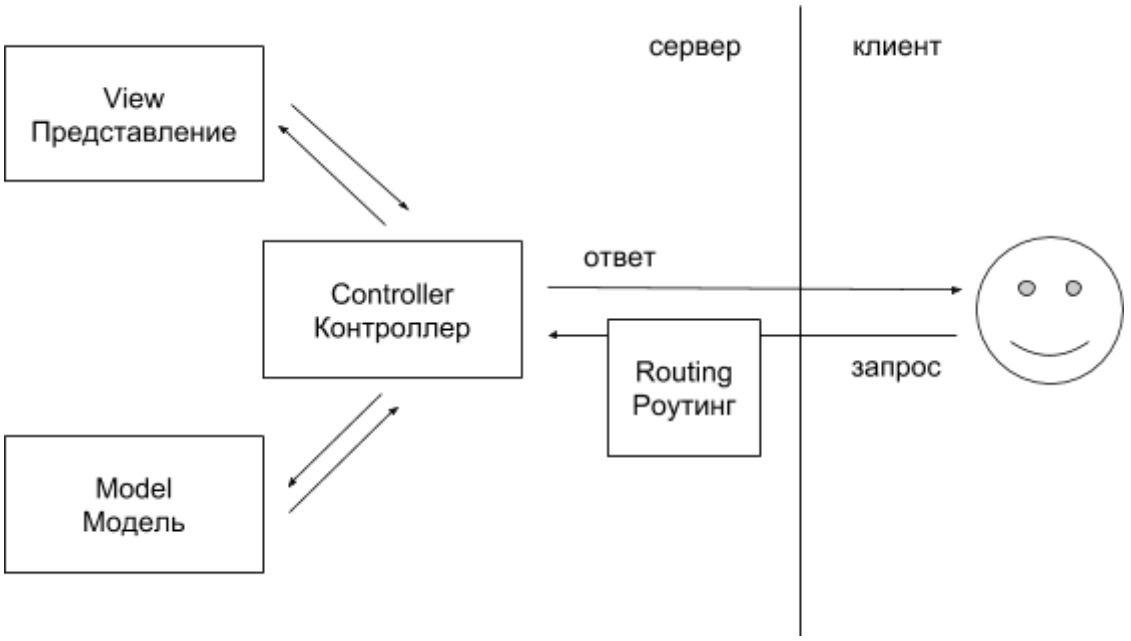
Урок 3

Модели

Базы данных. СУБД PostgreSQL. Модели. Генератор моделей. Миграции. Валидация.

- Базы данных
  - СУБД
  - SQL
  - PostgreSQL
  - Клиентские программы
- ActiveRecord
  - Настройка доступа к базе данных
  - Схема
  - Генератор моделей
  - Миграции
  - Откат миграций
- Консоль Ruby on Rails
- Соглашения
- Валидации
- Остальные модели
  - Сообщения Post
  - Комментарии Comments
- Домашнее задание
- Дополнительные материалы
- Используемая литература

Вспомним схему MVC из первого урока и рассмотрим концепцию модели. Задача моделей − работа с данными приложения.



Модель несет ответственность за долговременное сохранение состояния приложения. Как правило, это достигается сохранением в базу данных. Кроме того, модели выступают в качестве источника данных внутри приложения.

Базы данных

Возможно, вы уже знакомы с базами данных и языком запросов SQL. С помощью Rails можно работать с базой данных, даже не зная SQL. Однако для успешной разработки хороших приложений эти знания необходимы. Нужно понимать, какие запросы делает Rails, чтобы извлечь данные из базы. Без этого не получится написать эффективный код и оптимизировать ресурсоемкие запросы.

Для долговременного хранения информации операционная система использует файлы, которые хранятся на жестком диске. Файлы управляются через системные вызовы ОС. Такое управление имеет ряд ограничений:

- Существует несколько режимов работы с файлами: чтение, запись, перезапись, дозапись.
- Файлы имеют линейную структуру: данные дозаписываются в конец. Можно перемещать файловый указатель в начало или середину файла, но перезаписываться будут старые данные.
- Данные, записываемые в файл, попадают на жесткий диск не сразу, а по мере заполнения буфера.

Это приводит к тому, что при одновременном доступе к файлу нескольких процессов, некоторым из них доступ необходимо блокировать. Это нужно, чтобы один процесс не повредил данные, которые записывает другой процесс. Следовательно, операции над файлом надо организовывать в виде очереди. Часть операций в этой очереди будут конфликтовать, и необходим механизм, разрешающий эти конфликты. Более того, как правило, операции чтения данных более многочисленны, чем операции записи. Поэтому нужны механизмы для оптимизации и повышения эффективности извлечения данных. Иначе придется каждый раз сканировать файл от начала до конца в поисках нужного фрагмента.

СУБД

Сразу после первых компьютеров с долговременной памятью появились системы, которые облегчают работу с файловой системой. Их стали называть *системами управления базами данных* − СУБД.

За прошедшие годы СУБД прошли длительный путь, совершив несколько качественных скачков. Один из наиболее ярких − создание реляционных СУБД. В этих СУБД базы данных состоят из нескольких таблиц. Каждая из них включает несколько столбцов и множество строк. Таблицы связаны друг с другом при помощи внешних ключей. Операции на извлечение данных из одной или нескольких таблиц возвращают промежуточные таблицы. Помимо операций чтения, допускаются вставка, удаление и обновление данных. Такие основные операции называют CRUD-операциями − по первым буквам:

- Create − создание.
- Read − чтение.
- Update − обновление.
- Delete − удаление.

Идеи, лежащие в основе реляционных баз данных, настолько популярны, что используются в отрасли свыше 30 лет. Как правило, базы данных используют клиент-серверную технологию. СУБД − это сервер, который управляет базами данных, принимает и выполняет запросы. В качестве клиента выступает специализированная программа или программный код. Для доступа к серверу код использует библиотеку, например Ruby on Rails-приложение.

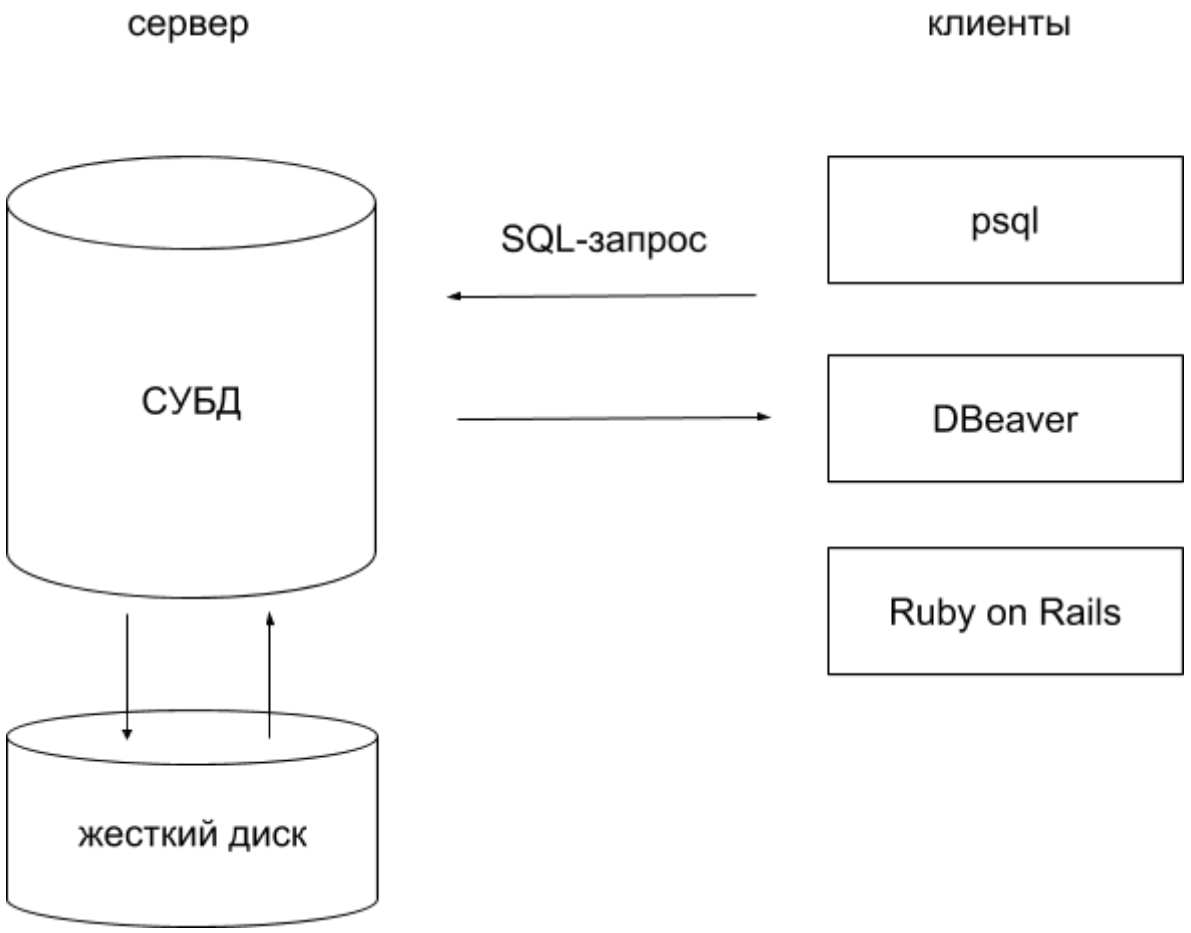
SQL

Благодаря единым принципам построения реляционных баз данных был разработан и стандартизирован единый язык запросов, с помощью которого клиент общается с СУБД. Язык запросов называется SQL − Structured Query Language или язык структурированных запросов.

Для получения информации из базы данных необходимо направить ей запрос, созданный с помощью SQL. Выполненный запрос − это результирующая таблица.

Но в настоящее время SQL − это нечто большее, чем язык запросов. С его помощью осуществляются все операции с данными в СУБД, а именно:

- **Выборка данных** − извлечение информации из базы данных.
- **Организация данных** − определение структуры базы данных и установление отношений между ее элементами.
- **Обработка данных** − добавление, изменение и удаление информации.
- **Управление доступом** − ограничение возможностей пользователей на доступ к некоторым категориям данных и защита данных от несанкционированного доступа.
- **Обеспечение целостности данных** − защита базы данных от разрушения.
- **Управление состоянием СУБД**.



SQL − это специализированный язык программирования. В отличие от языков высокого уровня (PHP, Python, Ruby и т. д.), он не позволяет создать независимую программу, которая работает без помощи других программ или языков программирования. Все запросы выполняются в специализированных либо прикладных программах при помощи библиотек.

Несмотря на то, что SQL строго стандартизирован, существует множество его диалектов. Каждая база данных реализует собственный вариант SQL со своими особенностями и конструкциями, которые недоступны в других базах данных. Это связано с тем, что стандарты SQL появились достаточно поздно. А компании, поставляющие базы данных, существуют давно. Таким компаниям необходимо обеспечивать для клиентов обратную совместимость баз данных со старыми версиями программного обеспечения.

Например, СУБД PostgreSQL значительно расширяет возможности традиционных СУБД за счет наследования таблиц и коллекционных типов данных: массивов, json-объектов, пользовательских типов данных.

Главная особенность реляционных СУБД − хранение данных в *таблицах*, состоящих из строк и столбцов. На пересечении каждого столбца и строки находится только одно значение. Строки и столбцы не равнозначны. У каждого столбца есть имя и тип, например целочисленный столбец id или текстовый столбец name. Столбцы располагаются в определенном порядке, который задается при создании таблицы.

Таблица users		
id	first_name	last_name
1	Максим	Кузнецов
2	Игорь	Симдянов
3	Сергей	Голышев
4	Дмитрий	Котеров
5	Алексей	Костарев

Строка

Столбец

Столбцы определяют структуру таблицы, а строки – количество записей. Как правило, одна база данных содержит несколько таблиц, связанных между собой или независимых друг от друга.

Для связи таблиц используются ключи. Допустим, есть база данных из двух таблиц: users и articles. Таблица users определяет авторов статей, а таблица articles содержит сами статьи. Одному автору или статье соответствует одна строка таблицы.

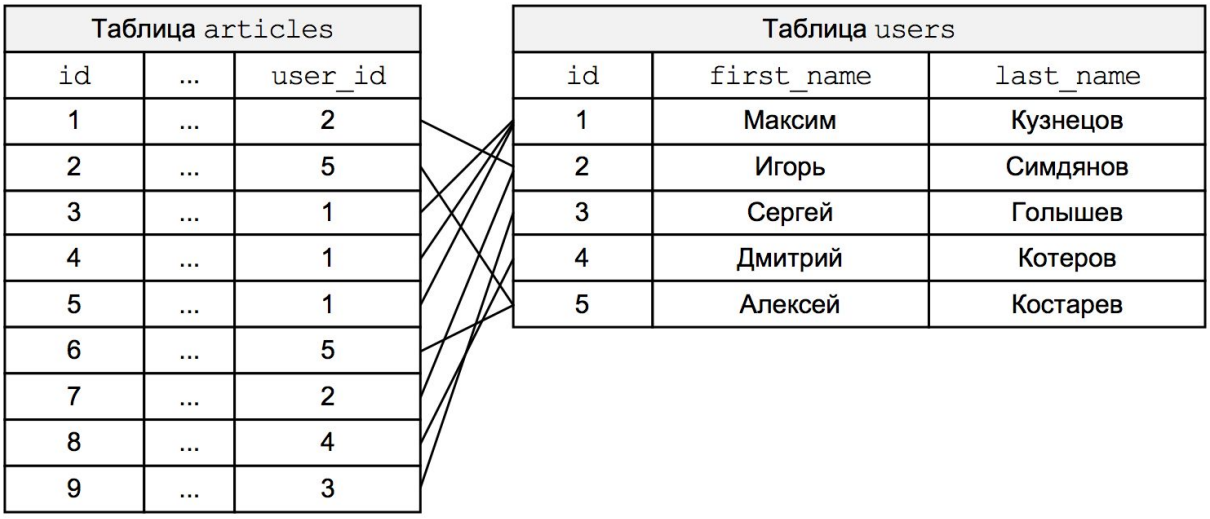


Таблица articles		
id	...	user_id
1	...	2
2	...	5
3	...	1
4	...	1
5	...	1
6	...	5
7	...	2
8	...	4
9	...	3

Таблица users		
id	first_name	last_name
1	Максим	Кузнецов
2	Игорь	Симдянов
3	Сергей	Голышев
4	Дмитрий	Котеров
5	Алексей	Костарев

Последнее поле в каждой строке таблицы articles с именем user_id содержит значение поля в столбце id таблицы users. По этому значению можно однозначно определить автора статьи. Так таблицы связаны друг с другом. Связь условна и может отсутствовать. Проявляется она только в результате специальных запросов. Такая форма организации информации получила название реляционной – связанной отношениями.

PostgreSQL

PostgreSQL задумывалась как СУБД, нарушающая каноны традиционных реляционных баз данных. Главной целью ее разработки было создание объектно-ориентированной СУБД, которая позволит легко взаимодействовать с объектно-ориентированным кодом и осуществлять наследование структур внутри СУБД. PostgreSQL стала предтечей современных NoSQL-баз данных.

Разработчики Ruby on Rails традиционно ориентируются на PostgreSQL. Подавляющее большинство проектов выполняются с использованием этой базы данных. Более того, фреймворк предоставляет широкую поддержку возможностям PostgreSQL.

Клиентские программы

Чтобы понимать, что происходит внутри базы данных, нужно просматривать ее структуру и содержимое таблиц. Делать это в рамках приложения не всегда удобно или возможно. Поэтому при работе с базой данных пригодятся независимые клиенты.

Если вы предпочитаете графический интерфейс, хорошим выбором будет DBeaver. Он реализован для всех основных операционных систем. Загрузить его дистрибутив можно здесь.

В составе PostgreSQL есть консольный клиент psql:

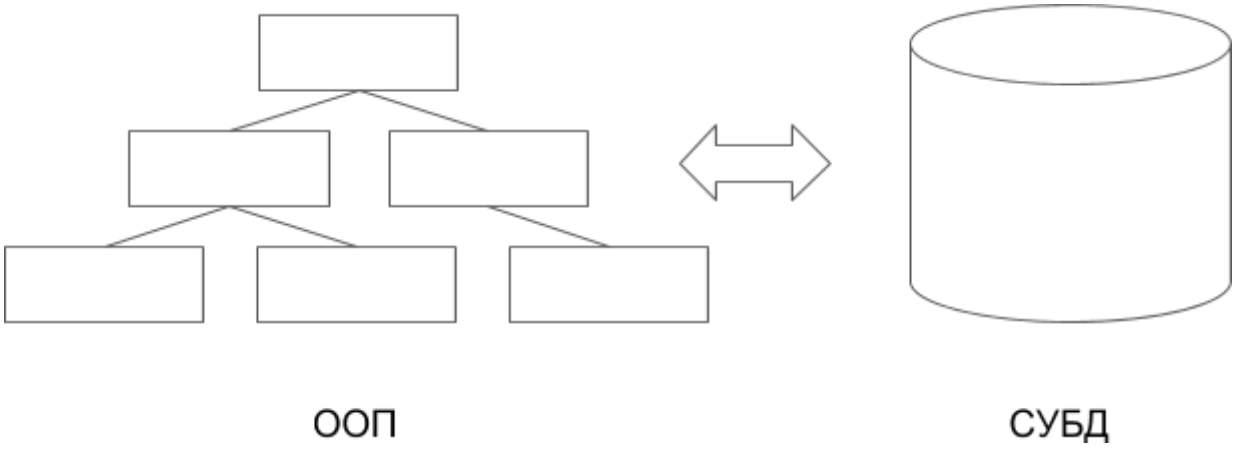
```
psql
# SELECT CURRENT_USER;
 current_user
--------------
 i.simdyanov
(1 row)

# \q
```

Вызывают консольный клиент командой:

```
rails dbconsole
```

ActiveRecord



Реляционная теория баз данных была заложена в 80-х годах прошлого столетия, и в ее основе лежат множества. Объектно-ориентированное программирование развивалось параллельно, и в его основе лежат деревья и графы. При сохранении объектов в базе данных приходится преобразовывать их в записи с жесткой структурой, а при извлечении − снова восстанавливать из них объекты. Практически

все исследователи отмечают, что соединение классических реляционных баз данных и объектно-ориентированных систем – крайне болезненная операция. Она приводит к потере преимуществ либо баз данных, либо объектно-ориентированных систем.

Есть два решения этой проблемы:

- Изменение традиционных реляционных баз данных, например введение наследования таблиц в PostgreSQL или появление документоориентированных NoSQL, как MongoDB.
- Создание сложных систем отражения объектов на реляционные базы данных.

Такие системы получили название ORM (Object-Relational Mapping) и призваны скрыть сложность сохранения объекта в базу данных. Есть два наиболее известных паттерна для решения этой задачи: ActiveRecord и DataMapper. В Ruby on Rails распространение получил паттерн ActiveRecord в виде одноименного гема. Размер гема ActiveRecord составляет 20 000 строк. Это та цена, которую приходится платить за решение сложной задачи.

Согласно паттерну ActiveRecord, таблице в базе данных соответствует класс модели, строкам таблицы – объекты класса модели, а каждому из столбцов – атрибуты модели. В Ruby on Rails таблица в базе данных является первичной. Именно по ней строится модель, а не наоборот, как, например, в Django.

Настройка доступа к базе данных

Чтобы взаимодействовать с базой данных, необходимо настроить соединение с ней в конфигурационном файле config/database.yml:

config/database.yml

```
default: &default
  adapter: postgresql
  encoding: unicode
  pool: <%= ENV.fetch("RAILS_MAX_THREADS") { 5 } %>

development:
  <<: *default
  database: blog_development

test:
  <<: *default
  database: blog_test

production:
  <<: *default
  database: blog_production
  username: blog
  password: <%= ENV['BLOG_DATABASE_PASSWORD'] %>
```

Файл разбит на секции по количеству поддерживаемых окружений: development, test, production. В каждое из окружений добавлена секция default, чтобы исключить дублирование кода. В ней указывается адаптер для соединения с базой данных postgresql. Вместо postgresql могут быть mysql, sqlite3, oracle, sqlserver. При создании нового приложения командой rails new в качестве адаптера по умолчанию используется sqlite3. Это файловая база данных, используемая во встраиваемых

решениях. Благодаря тому, что мы использовали параметр --database, конфигурационный файл config/database.yml ориентирован на PostgreSQL.

```
rails new my_app --database=postgresql
```

Ключ encoding (см. config/database.yml) задает в качестве кодировки по умолчанию UTF-8. Другая кодировка в новых приложениях использоваться не будет.

Параметр pool задает количество соединений с базой данных на процесс. В классическом MRI варианте Ruby выбирать этот параметр стоит осторожно. Дело в том, что в Ruby не реализована полноценная многопоточность: в рамках процесса могут работать несколько потоков, но в каждый момент времени работает только один из них. Поэтому количество соединений на процесс не следует устанавливать избыточным. Особенно это актуально, когда используется fork-сервер вроде Unicorn, который может запускать десятки процессов. Произведение параметра pool и количества процессов дает то количество соединений, которое установит Ruby on Rails с базой данных. Ситуация усугубится, если используется несколько серверов с Ruby on Rails-приложением. Количество соединений, которые может поддерживать сервер PostgreSQL, величина конечная и устанавливается в размерах от 100 до 500.

Параметр database (см. config/database.yml) позволяет установить название базы данных, username − имя, и password − пароль. При локальной разработке зачастую это не требуется, так как PostgreSQL использует имя текущего пользователя и осуществляет соединение без пароля.

Обратите внимание, что в продакшн-окружении пароль устанавливается через переменную окружения. Она извлекается через предопределенную константу-хэш ENV. Это очень распространенный подход, который позволяет передавать код сторонним разработчикам и выкладывать его в свободный доступ. При этом вся критическая информация (пароли, токены) выставляются при помощи переменных окружений, которые не покидают сервера.

Чтобы создать базу данных, есть rake-команда:

```
rails db:create
```

Эта команда создала две базы данных: для development- и test-окружений. Для удаления базы данных есть команда:

```
rails db:drop
```

Допустимо выполнение нескольких rake-задач в рамках одного вызова команды rails:

```
rails db:drop db:create
```

Схема

Перед созданием таблиц и написанием кода необходимо понять, какие данные надо хранить.

В нашем блоге должны быть сообщения (статьи), пользователи и комментарии.

У каждого сообщения должны быть автор, название и содержание. Сообщения можно удалять и восстанавливать.

У каждого комментария должны быть автор, содержание, указание, к какому сообщению относится комментарий, а также время создания комментария. Комментарии можно удалять и восстанавливать.

У каждого пользователя есть уникальный логин и адрес электронной почты.

По умолчанию пользователь может только оставлять комментарии под сообщениями. Помимо этого, у него могут быть права на добавление статей и модерирование. Модератор имеет право забанить пользователя.

Генератор моделей

Для создания моделей Ruby on Rails предоставляет специальный генератор **rails g model**:

```
rails generate model NAME [field[:type][:index] field[:type][:index]] [options]
```

В качестве NAME указывается имя модели в snake_case-режиме. Квадратные скобки обозначают необязательные элементы. **Field** – это название столбца в таблице, **:type** – тип столбца (:string, :text, :boolean, :integer, :datetime). Элемент **:index** указывает на необходимость индексирования столбца. Индекс – это копия столбца, которая поддерживается СУБД в отсортированном состоянии. Поэтому поиск по нему исключительно быстр. Тем не менее, большое количество индексов приводит к замедлению операций вставки, обновления и удаления. Поэтому их добавление должно быть взвешенным шагом, доказанным при помощи команды explain.

Вернемся к пользователям. У каждого пользователя есть уникальный логин строкового типа name. Добавим адрес электронной почты. Еще нам потребуются флаги, разрешающие пользователю быть модератором и автором, а также флаг, который обозначает, забанен пользователь или нет.

Обратите внимание, что пока у пользователя нет пароля и возможности зарегистрироваться. Это тема шестого занятия. Генератор, который создает пользователя с указанными выше параметрами, выглядит так:

```
rails g model user name:string:uniq email:string:uniq moderator:boolean
creator:boolean banned:boolean
      invoke  active_record
      createdb/migrate/20180108092508_create_users.rb
      createapp/models/user.rb
```

Rails создал миграцию и саму модель. Модель не заработает до тех пор, пока не будет выполнена миграция. И пока по миграции не будет создана таблица в базе данных. До того, как миграция будет выполнена, ее можно модифицировать.

Миграции

Миграции описывают изменения, которые необходимо провести со структурой базы данных: создание новых таблиц, переименование полей, перенос данных из одной таблицы в другую и прочее.

Миграции располагаются в папке db/migrate. Каждая миграция предваряется префиксом с датой ее создания. Это предотвращает коллизии, когда два разработчика создают миграции с одинаковым названием. Кроме того, дата позволяет отсортировать миграции в хронологическом порядке. Это

важно при выполнении их на свежей базе данных, например при выводе сайта в продакшн. При выполнении миграций маркеры даты регистрируются в специальной таблице schema_migrations. По ней Ruby on Rails понимает, была выполнена миграция или нет.

Миграция представляет собой класс, который наследуется от ActiveRecord::Migration. Название класса не может быть произвольным. Во-первых, оно должно быть уникальным, а во-вторых – коррелировать с частью названия файла миграции после префикса с датой.

Внутри класса должен быть определен метод **change**, который выполняет операции с базой данных. Эти операции оформляются в виде DSL-языка программирования. То есть это обычные методы и блоки языка Ruby. Однако за счет того, что опускаются скобки, создание, изменение и манипуляция таблицами выглядят как декларативные объявления.

Методов, обслуживающих миграции, очень много. Познакомиться с ними лучше по документации. Опишем лишь те, с которыми непосредственно столкнемся.

Метод **create_table** создает таблицу, имя которой указывается в виде символа :users. В блоке, который принимает метод, указывается состав будущей таблицы.

```
class CreateUsers < ActiveRecord::Migration[5.1]
  def change
    create_table :users do |t|
      t.string :name
      t.string :email
      t.boolean :moderator
      t.boolean :creator
      t.boolean :banned

      t.timestamps
    end
    add_index :users, :name, unique: true
    add_index :users, :email, unique: true
  end
end
```

В таблицу автоматически добавляется первичный ключ с именем id. Можно запретить это делать или задать другое имя. Однако присутствия ключа :id требуют Rails-соглашения. Наличие первичного ключа с именем id существенно сокращает объем кода, необходимого для связывания таблиц.

Метод **timestamps** является особым объявлением и добавляет в таблицу два поля: created_at и updated_at. При создании записи в оба поля помещается время создания в UTC-формате. При последующем обновлении записи автоматически обновляется и время в поле updated_at. Так что всегда известно время создания и последнее время обновления записи в таблице.

В конце миграции методы **add_index** в таблице users для столбцов name и email создают уникальные индексы. Они предотвращают появление в таблице записей с повторяющимися именами и электронными адресами.

Код миграции, которую создал генератор, можно усовершенствовать, так как это обычный ruby-код. Например, булевые поля по умолчанию можно при помощи ключа :default установить в false, а конструкции add_index заменить на параметры :index метода **string**.

```
class CreateUsers < ActiveRecord::Migration[5.1]
```

```
  def change
    create_table :users do |t|
      t.string :name, index: { unique: true }
      t.string :email, index: { unique: true }
      t.boolean :moderator, default: false
      t.boolean :creator, default: false
      t.boolean :banned, default: false

      t.timestamps
    end
  end
end
```

Чтобы сократить повторы в вызовах, воспользуемся методом **with_opitions**:

```
class CreateUsers < ActiveRecord::Migration[5.1]
  def change
    create_table :users do |t|
      t.with_options index: { unique: true } do
        string :name
        string :email
      end
      t.with_options default: false do
        boolean :moderator
        boolean :creator
        boolean :banned
      end

      t.timestamps
    end
  end
end
```

Для выполнения миграции выполним команду rails db:migrate:

```
rails db:migrate
== 20180108092508 CreateUsers: migrating ======================================
-- create_table(:users)
   -> 0.0145s
== 20180108092508 CreateUsers: migrated (0.0146s) =============================
```

В базе данных появилась таблица для хранения пользователей. Убедимся в ее наличии, посмотрев на то, что получилось, с помощью клиента базы данных (psql, dbeaver и др.). Воспользуемся для этого консольным клиентом psql и сразу после его загрузки окажемся в базе данных пользователя. Но чтобы выполнять запросы, придется переключиться на базу данных приложения blog_development. Загрузить список всех баз данных можно при помощи команды **\l**. Для переключения на базу данных нужна команда \c blog_development:

```
psql
# \c blog_development
```

Чтобы получить список таблиц текущей базы данных, воспользуемся командой **\dt:**

```
# \dt
                 List of relations
 Schema |          Name          | Type  |    Owner
--------+------------------------+-------+-------------
 public | ar_internal_metadata | table | i.simdyanov
 public | schema_migrations    | table | i.simdyanov
 public | users                | table | i.simdyanov
(3 rows)
```

Чтобы ознакомиться со структурой одной из таблиц – командой **\d:**

```
# \d users
                                    Table "public.users"
   Column   |            Type                |                    Modifiers
------------+--------------------------------+-----------------------------------
---------------
 id         | bigint                         | not null default
nextval('users_id_seq'::regclass)
 name       | character varying              |
 email      | character varying              |
 moderator  | boolean                        | default false
 creator    | boolean                        | default false
 banned  | boolean                      | default false
 created_at | timestamp without time zone | not null
 updated_at | timestamp without time zone | not null
Indexes:
      "users_pkey" PRIMARY KEY, btree (id)
      "index_users_on_email" UNIQUE, btree (email)
      "index_users_on_name" UNIQUE, btree (name)
```

Помимо этого, в таблице schema_migrations появилась новая запись с датой и временем миграции. Если теперь снова выполнить команду rails db:migrate, миграция с таким префиксом-датой выполняться не будет.

```
SELECT * FROM schema_migrations;
      version
----------------
 20180108092508
(1 row)
```

Откат миграций

Миграции можно проводить как в одну сторону, так и в другую. Миграцию базы из одного состояния в другое можно безболезненно откатить назад, если она не удаляла какие-либо данные. В простых случаях Rails сам знает, как создавать и откатывать миграции. Например, если нужно создать дополнительные таблицы или колонки, такие вещи просто записывают в миграции внутри метода change. Чтобы откатить миграцию, есть команда:

```
rails db:rollback
```

Чтобы откатить и снова выполнить исправленную миграцию, используется комбинация команд:

```
rails db:rollback db:migrate
```

Есть также отдельная команда, которая откатывает и выполняет миграции:

```
rails db:migrate:redo
```

Если необходимо реализовать нестандартное поведение, используют блок reversible. С его помощью можно явно указать операции, которые выполняются при команде migrate и команде rollback. Для этого предназначены методы **up** и **down** — для миграции и отката, соответственно.

```
class CreateUsers < ActiveRecord::Migration[5.1]
  def up
    create_table :users do |t|
      t.with_options index: { unique: true } do
        string :name
        string :email
      end
      t.with_options default: false do
        boolean :moderator
        boolean :creator
        boolean :banned
      end

      t.timestamps
    end
  end

  def down
    raise ActiveRecord::IrreversibleMigration
  end
end
```

В этом примере в блоке down возбуждается исключительная ситуация, тем самым запрещая выполнение команды rails db:rollback.

Консоль Ruby on Rails

Ruby on Rails предоставляет специальный интерактивный режим для тестирования и экспериментов. Для входа в него предназначена команда rails console или, сокращенно, rails c. Она открывает интерактивную консоль, поведение которой аналогично утилите irb (интерактивный Ruby).

```
rails c
bundle :001 > 2 + 2
4
```

Отличие от irb в том, что в случае Rails-консоли полностью подгружается окружение приложения. Это предоставляет доступ ко всем классам, например к моделям.

```
>> User.columns.map(&:name)
=> ["id", "name", "email", "moderator", "creator", "banned", "created_at",
"updated_at"]
```

В файле app/models/user.rb объявлен класс User, наследник класса ApplicationRecord. Это и есть модель. Обратимся из интерактивной консоли к этой модели и получим все доступные записи. Введем команду User.all, которая возвращает список всех записей модели User, обернутых в классы модели. Так как у нас не создано ни одного пользователя, как результат получаем пустой список.

```
>> User.all
  User Load (13.9ms)  SELECT  "users".* FROM "users" LIMIT $1  [["LIMIT", 11]]
=> #<ActiveRecord::Relation []>
```

При выполнении методов моделей Ruby on Rails выводит SQL-запрос, который отправляется серверу базы данных, а также время на выполнение этого запроса в микросекундах. Получить SQL-запрос можно и самостоятельно, воспользовавшись методом **to_sql**:

```
>> User.all.to_sql
=> "SELECT \"users\".* FROM \"users\""
```

Для инициализации пользователя используется команда **User.new <attributes>**, а для его сохранения в базу данных – метод **save**.

```
>> u = User.new name: 'test', email: 'test@test.ru'
=> #<User id: nil, name: "test", email: "test@test.ru", moderator: false,
creator: false, banned: false, created_at: nil, updated_at: nil>
>> u.save
   (0.2ms)  BEGIN
  SQL (7.6ms)  INSERT INTO "users" ("name", "email", "created_at", "updated_at")
VALUES ($1, $2, $3, $4) RETURNING "id"  [["name", "test"], ["email",
"test@test.ru"], ["created_at", "2018-01-08 11:34:48.033396"], ["updated_at",
"2018-01-08 11:34:48.033396"]]
   (12.5ms)  COMMIT
```

Конструктор new создает объект класса User в оперативной памяти. Вызов метода save формирует и выполняет SQL-запрос на вставку новой записи в таблицу users. Эти две операции можно объединить, воспользовавшись методом **User.create <attributes>**. Он инициализирует и сразу сохраняет пользователя.

```
>> u = User.create name: 'test_moderator', email: 'moderator@test.ru',
moderator: true
   (0.2ms)  BEGIN
  SQL (0.5ms)  INSERT INTO "users" ("name", "email", "moderator", "created_at",
"updated_at") VALUES ($1, $2, $3, $4, $5) RETURNING "id"  [["name",
"test_moderator"], ["email", "moderator@test.ru"], ["moderator", "t"],
["created_at", "2018-01-08 11:41:58.837652"], ["updated_at", "2018-01-08
11:41:58.837652"]]
   (6.3ms)  COMMIT
=> #<User id: 2, name: "test_moderator", email: "moderator@test.ru", moderator:
true, creator: false, banned: false, created_at: "2018-01-08 11:41:58",
updated_at: "2018-01-08 11:41:58">
```

Допустимо обновлять данные пользователя при помощи метода **update**:

```
>> u = User.find_by(name: 'test')
  User Load (0.4ms)  SELECT  "users".* FROM "users" WHERE "users"."name" = $1
LIMIT $2  [["name", "test"], ["LIMIT", 1]]
=> #<User id: 1, name: "test", email: "test@test.ru", moderator: false, creator:
false, banned: false, created_at: "2018-01-08 11:34:48", updated_at: "2018-01-08
11:34:48">
>> u.update(name: 'hello')
   (0.2ms)  BEGIN
  SQL (2.8ms)  UPDATE "users" SET "name" = $1, "updated_at" = $2 WHERE
"users"."id" = $3  [["name", "hello"], ["updated_at", "2018-01-08
11:58:58.436932"], ["id", 1]]
   (0.3ms)  COMMIT
```

Или удалять их при помощи метода **destroy**:

```
>> u.destroy
   (0.2ms)  BEGIN
  SQL (0.5ms)  DELETE FROM "users" WHERE "users"."id" = $1  [["id", 1]]
   (6.5ms)  COMMIT
```

Соглашения

SQL-команды, выполняющие CRUD-операции, в консоли и логах подсвечены разными цветами. Поэтому, даже не зная SQL, можно примерно понимать, какие операции выполняются:

- *Темно-синий* – извлечение данных из таблицы при помощи SELECT-запроса.

- *Зеленый* – вставка данных в таблицу при помощи INSERT-запроса.
- *Желтый* – обновления данных таблицы при помощи UPDATE-запроса.
- *Красный* – удаление записи при помощи DELETE-запроса.

Обратите внимание и на имена таблиц и классов моделей. Классы моделей – это всегда константы в CameCase-режиме в **единственном числе**, а названия таблиц записываются в snake_case-режиме во **множественном числе**. Если ошибиться в имени, модель не сможет обнаружить таблицу в базе данных. Поэтому на начальных этапах обучения лучше прибегать к генераторам, которые позволяют избегать таких ошибок.

Валидации

При успешном создании пользователя с помощью метода save, метод вернет значение true. Если записать значение в базу данных не удалось, результатом функции станет false. Метод create при успешной записи вернет объект только что созданного экземпляра модели, в противном случае — false. Также существуют аналогичные методы **save!** и **create!**. Они отличаются тем, что в случае ошибки выкинут Exception.

Создадим пользователя с уже занятым именем:

```
>> u = User.create name: 'test', email: 'test@test.ru'
   (0.2ms)  BEGIN
  SQL (14.3ms)  INSERT INTO "users" ("name", "email", "created_at",
"updated_at") VALUES ($1, $2, $3, $4) RETURNING "id"  [["name", "test"],
["email", "test@test.ru"], ["created_at", "2018-01-08 11:44:47.224431"],
["updated_at", "2018-01-08 11:44:47.224431"]]
   (0.3ms)  ROLLBACK
ActiveRecord::RecordNotUnique: PG::UniqueViolation: ERROR:  duplicate key value
violates unique constraint "index_users_on_name"
DETAIL:  Key (name)=(test) already exists.
```

База данных отвергла добавление новой записи, вернув сообщение об ошибке. Чтобы не тревожить лишний раз базу данных, можно не пропускать запрос на создание записи на стороне Rails. Для этого существуют валидации.

Валидации прописываются в классах моделей. Например, можно ограничить имя пользователя длиной от 2 до 16 символов и проверить уникальность имени.

app/models/user.rb

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

Красота Ruby и Rails как раз в том, что исходный код выглядит почти как естественный текст на английском языке. Легко догадаться, что означает каждое правило. Первое правило подтверждает, что атрибут name присутствует. Второе говорит, что длина атрибута message лежит в пределах от 2 до 16 символов. Третье указывает на уникальность.

В консоли действует старый вариант модели User. Чтобы появился новый, необходимо либо выйти и снова зайти в консоль, либо перезагрузить консоль при помощи метода **reload!**:

```
reload!
```

В интерактивной консоли создадим нового пользователя с заведомо неверным именем:

```
>> u = User.new name: '1', email: 'hello@world.ru'
=> #<User id: nil, name: "1", email: "hello@world.ru", moderator: false,
creator: false, banned: false, created_at: nil, updated_at: nil>
>> u.valid?
  User Exists (0.8ms)  SELECT  1 AS one FROM "users" WHERE "users"."name" = $1
LIMIT $2  [["name", "1"], ["LIMIT", 1]]
=> false
>> u.errors
=> #<ActiveModel::Errors:0x00007fd07245d3b0 @base=#<User id: nil, name: "1",
email: "hello@world.ru", moderator: false, creator: false, banned: false,
created_at: nil, updated_at: nil>, @messages={:name=>["is too short (minimum is
2 characters)"]}, @details={:name=>[{:error=>:too_short, :count=>2}]}>
```

Чтобы проверить только что инициализированную модель на валидность, нужно вызвать от нее метод **valid?**. В зависимости от результата проверки он вернет истину или ложь. Если проверка завершилась неудачей, список ошибок можно посмотреть, вызвав от объекта модели метод **errors**. Конкретные сообщения ошибок можно получить, вызвав методы **.errors.full_messages**.

Остальные модели

Осталось создать модели для сообщений (Post) и комментариев (Comment). Сообщения ссылаются на пользователей, а комментарии – и на пользователей, и на сообщения. В реляционных базах данных для решения такой задачи принято создавать внешний ключ, то есть колонку с названием <таблица_назначения>_id. Ruby on Rails умеет делать это автоматически: при генерации можно указать специальный тип references.

Сообщения (Post)

Создадим модель сообщений со ссылкой на пользователя, заголовком, телом и маркером доступности на отображение:

```
rails g model post title:string:uniq body:text user:references visible:boolean
invoke  active_record
  create    db/migrate/20180108122919_create_posts.rb
  create    app/models/post.rb
```

Отредактируем миграцию:

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

```
    create_table :posts do |t|
      t.string :title, index: { unique: true }
      t.text :body
      t.references :user, foreign_key: true
      t.boolean :visible, default: false

      t.timestamps
    end
  end
end
```

Комментарии (Comments)

Аналогично создадим модель комментариев:

```
rails g model comment body:text user:references post:references visible:boolean
 invoke  active_record
  create    db/migrate/20180108124655_create_comments.rb
  create    app/models/comment.rb
```

```
class CreateComments < ActiveRecord::Migration[5.1]
  def change
    create_table :comments do |t|
      t.text :body
      t.references :user, foreign_key: true
      t.references :post, foreign_key: true
      t.boolean :visible, default: false

      t.timestamps
    end
  end
end
```

Остается выполнить команду rails db:migrate, чтобы создать новые таблицы в базе данных.

Домашнее задание

1. Напишите валидации для моделей сообщений (Post) и комментариев (Comment).
2. Установите и настройте клиент для управления базой данных, например DBeaver.
3. Создайте rake-задачу, которая заполняет таблицы пользователей, сообщений и комментариев тестовыми данными.
4. *Добавьте таблицу (marks) для хранения оценок пользователей (users) к сообщениям (posts).

   «*» Задание повышенной сложности (по желанию).

Дополнительные материалы

1. Документация по валидаторам.
2. Документация по регулярным выражениям.
3. Основные понятия реляционных баз данных.
4. Правила Кодда.
5. Документация PostgreSQL на русском языке.
6. GUI-клиент для доступа к базам данных DBeaver.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Документация PostgreSQL на русском языке.
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.
7. Джеффри Фридл. Регулярные выражения.