Ruby — интерпретируемый, полностью объектно-ориентированный язык программирования с четкой динамической типизацией. Он сочетает в себе Perl-подобный синтаксис с объектно-ориентированным подходом. Также некоторые черты заимствованы из языков программирования Python, Lisp, Dylan и CLU. Кроссплатформенная реализация интерпретатора языка Ruby распространяется на условиях открытого программного обеспечения. Код, написанный на Ruby, может быть понятен даже человеку, который не разбирается в программировании. На RoR были созданы такие проекты, как Redmine, Twitter, Shopify, Basecamp, GitHub, Kickstarter, Airbnb и другие.

С ростом Node.js популярность Ruby on Rails несколько уменьшилась, но технологические стартапы часто используют RoR благодаря простоте прототипирования. Ruby — 13-й самый популярный язык [в индексе TIOBE](https://www.tiobe.com/tiobe-index/) на апрель 2020 года.

<https://www.ruby-lang.org/ru/about/> - О Ruby

Расширение Ruby файлов - example.rb ---- чтобы вывести в терминале пишем ---------- ruby example.rb

puts "Hello, world!" // puts - сразу ставит новую строку

print "Bye-bye\n"

age = 18 //объвили переменную и вывели

puts(age)

age = 24 //переопределили переменную

puts(age)

Типы переменных в языке Ruby не объявляются очевидно, тем не менее они

присутствуют.

Интерпретатор понимает, что записывается в переменную и на основании этого добавляет тип к этой переменной.

В ходе программы мы можем перезаписывать значение переменной, при этом мы можем указывать новый тип переменной.

Ниже приводим несколько существующих типов данных в языке Ruby:

some = 1 Integer - целые числа;

some = 1.12 Float - числа с плавающей точкой;

some = "Привет" String - строки;

some = true Boolean - принимают только true или же false;

some = nil Nil - пустое значение переменной.

name = 'Hello, everyone!'

puts(name)

\*puts name

Если аргумент один то используем вышеописанный способ, но если аргументов больше чем один то только тогда ты уже передаешь их используя скобки

puts(name, age)

Посмотри code style для ruby и учи людей изначально писать правильный код

text = "Hello, everyone!"

puts(text)

\* '' или "" разницы нет

hapiness = 0.98

person\_has\_car = true

none = nil

\*в переменной ничего нет, она пуста

print(none)

\*ничего не выведет

#3 - Строки и математические действия

Для работы со строками предусмотрено множество методов, что позволяют трансформировать строку. При помощи методов можно посчитать количество символов в строке (length), привести всё в верхний (upcase) или нижний (downcase) регистр и выполнить множество других операций.

words = "Hello, world\n\"Привет\""

word = "!"

puts(words.upcase() + word)

some\_text = " Много пробелов "

puts(some\_text.strip()) //убирает пробелы до и после текста, в середине не убирает

website = "itProger"

puts(website.include? "it") //ищет в строке website символы it выводит либо true, либо false

В Ruby можно выполнять все доступные математические операции. Это такие операции, как:

* вычитание - z = y - x;
* сложение - z = y + x;
* умножение - z = y \* x;
* деление - z = y / x (на ноль делить нельзя);
* остаток от деления - z = y % x;
* возвести в степень z = 2 \*\* 3
* операция z = 3^4 выведет тебе 7:
* если первое число меньше второго, то ^ вызовет сложение, если же второе число меньше, то будет вычитание. если они соответственно равны, будет 0.

Если операция проходит над одной и той же переменной, то запись можно сократить:

z = z + 20

z += 20 # Эта запись аналогична записи выше

Нельзя объединить разные типы данных, используем метод to\_s и переводим число z в строку:

x = 2

y = 11

z = 11 % 2

puts("Результат остатка от деления: " + z.to\_s)

Операция по модулю:

num = -14.65

puts(num.abs())

puts(num.round()) - округлит

Вывод: 14

Можно обращаться к классу Math = снабжает математическими методами:

puts Math.sqrt(144)

Возвращает: 12.0

Можно объединять строки:

words = "Hello, world"

word = "!"

puts (words + word)

Вывод: Hello, world!

# #4 - Получение данных от пользователя

Для получения данных от пользователя необходимо использовать метод gets. Все полученные данные можно записать в переменные, при этом тип данных для переменной будет указан как String (строка).

puts("Введите свое имя: ")

name = gets

puts("Ваше имя: " + name)

\*все полученное gets является строкой даже цифры

puts("Введите свое имя: ")

name = gets

puts("Ваше имя: " + name + ", ты крут!")

* gets - захватывает также <enter> - в итоге вывод:

Ваше имя: Fa

, ты крут!

puts("Введите свое имя: ")

name = gets.chomp() //убирает enter

puts("Ваше имя: " + name + ", ты крут!")

* вывод: Ваше имя: Fa, ты крут!

print("Put first number:")

x = gets.chomp() //11

print("Put second number:")

y = gets.chomp() //423

puts(x + y)

Вывод: 11423

Т.к. gets получает строки, то puts просто соединяет две строки

print("Put first number:")

x = gets.chomp()

print("Put second number:")

y = gets.chomp().to\_i

puts(x.to\_i + y)

* to\_i - приводим к целочисленным; to\_f - приводим к типу float
* # - комментарий

Можно изменить тип данных переменной для последующего взаимодействия с ним. Для изменения используйте следующие методы:

* to\_s - привод значение к типу данных строка;
* to\_i - привод значение к типу данных целое число;
* to\_f - привод значение к типу данных число с плавающей точкой.

Puts, gets, to\_i, to\_s, print и т.д. – это всё методы. Руби конечно позволяет вызывать их как со скобками, так и без. Но хорошим стилем считается либо пользоваться ими только по необходимости, либо ставить их везде.

# #5 - Массивы и ассоциативные массивы

Массивы состоят из множества элементов и позволяют работать с каждым из них.

Массивы позволяют хранить большой объем информации в одном месте. В языке **Ruby** можно найти как минимум два основных типа массивов: обычные массивы и ассоциативные массивы.

**Одномерный массив данных**

Одномерный массив состоит из нескольких элементов, объединенных под одним именем. Чтобы создать такой массив необходимо создать переменную и присвоить ей значение Array.new. Внутри массива можно через запятую записывать различные значения, которые и будут элементами массива. В Ruby - в массиве можно смешивать типы данных, перечислять через запятую int, string, float, boolean etc.

arr = Array[4, 6, 8, 12, true, "Hello", 5.64]

puts(arr)

Массив выше равен массиву ниже, если сразу заполняем массив, то можно не вводить Array:

arr = [4, 6, 8, 12, true, "Hello", 5.64]

puts(arr)

В массивах отсчет начинается с 0, поэтому первый элемент всегда идет по индексу 0, второй по индексу 1 и так далее.

names = Array["Gear", "Marie", "Elein"]

puts(names[1])

puts(names[-1]) - последний элемент массива

puts(names[-2]) - предпоследний элемент массива/ второй элемент с конца массива

puts(names[1, 2]) - Marie и Elein

puts names.reverse()

names = Array["Gear", "Marie", "Elein"]

names[0] = "Peter"

puts(names[0])

Поменяли Gear на Peter

list = Array.new

puts list

* list пустой массив, т.е. состоит из nil значений

list = Array.new

list[0] = 12

list[7] = 36

puts list

puts list.length() #Выводит, что кол-во элементов 8

puts list.include? 12 #Выводит true

* недостающие элементы массива между индексами 0 и 7 автоматически заполняются интерпретатором языка руби значением nil. Массив list выглядит на самом деле так: [12, nil, nil, nil, nil, nil, nil, 36] И это на самом деле очень важно. Потому что если какой-нибудь человек насмотревшись твоих видосов захочет исполнить что-то типа list.each{|x| puts x\*x} то получит ошибку, которую тупо не будет знать как разрешить.

Примеры создания массива:

some = Array.new # Создание пустого массива

some[0] = '1' # Добавление 1 элемента

some[1] = 2 # Добавление 2 элемента

print(some[0]) # Вывод первого элемента

array = Array[1, 5, 2] # Создание массива сразу со значениями

Работать с элементами массива можно точно также как с переменными. Мы можем их выводить или же устанавливать для них новые значения.

**Ассоциативные массивы**

Ассоциативный массив отличается от обычного лишь индексами. Здесь вместо чисел можно подставить, так называемый **ключ**, который будет отвечать за определённый элемент в массиве. Название для ключей может быть каким-угодно.

* В качестве ключей ассоциативного массива можно использовать любые типы. Например, другие ассоциативные массивы, строки, числа, символы или просто обьекты любых классов. Не три перечисленных типа данных – String, Integer (FixNum) и Symbol (да-да, это не "такая вот строка, а именно символ – специальный тип данных в руби), а все имеющиеся в руби типы данных могут быть как ключами, так и значениями в ассоциативных массивах (хэшах – Hash).

Пример массива:

some = {

"Ключ" => "Значение"

}

countries = {

  "RU" => "Russia",

  1 => 1.01,

  :UA => "Ukraine"

}

puts countries["RU"]

puts countries[1]

puts countries[:UA]

ВЫВОД:

Russia

1.01

Ukraine

# #6 - Методы и оператор return

Метод - это функция внутри класса. А это самая обычная функция или процедура (функция возвращает значение, а процедура возвращает управление).

Функции можно назвать небольшими подпрограммами, куда можно вынести повторяющийся код и обращаться к нему, когда это будет нужно. Функции значительно облегчают построение программ, так как нам не надо копировать однотипный код множество раз, а можно просто воспользоваться одной общей функцией. Функция - метод класса main.

В Ruby функции создаются при помощи ключевого слова def. Каждая функция может иметь какие-либо параметры или же не иметь их вовсе. Функции способны что-либо возвращать в ходе выполнения кода, если это требуется.

def some\_test # Функция без параметров

x = 23

return x # Функция возвращает значение переменной

end

def some\_test\_2(a, b, c) # Функция с 3 параметрами

return a \* b \* c # Функция возвращает результат умножение чисел

end

def print\_something(word, prefix) # Функция с 2 параметрами

print(prefix + "-" + word)

# Функция ничего не возвращает

end

Функции могут возвращать другие функции, тем самым вызывая их. Чтобы обратиться к функции необходимо прописать её название и передать параметры, если таковы имеются:

res = some\_test\_2(2, 67, 12)

В примере выше результат выполнения функции будет помещен в переменную res. Далее с переменной можно работать как с обычным значением в программе.

def sayHello //объявили функцию, без параметров

  puts "Hello, world!" //она выводит Hello, world!

end

sayHello //вызываем функцию

sayHello //вызываем функцию

sayHello //вызываем функцию

Добавили параметры:

def saywHello(word, num)

  puts "Hello, world!"

  puts ("Your word: " + word + " and your num: " + num.to\_s)

end

saywHello("Heey", 89)

Можно добавить значение по умолчанию, тогда не передав 1 параметр, не возникнет ошибки - просто подставится значение по умолчанию:

def saywHello(word, num=2)

  puts "Hello, world!"

  puts ("Your word: " + word + " and your num: " + num.to\_s)

end

saywHello("Heey")

def saywHello(word="Ruby", num=2)

  puts "Hello, world!"

  puts ("Your word: " + word + " and your num: " + num.to\_s)

end

saywHello

Функция/метод может возвращать значение:

def summa(x, y)

  return x + y, 70

end

res = summa(6, 5)

puts res

ВЫВОД: 11

70

puts res[0]

ВЫВОД: 11

puts res[1]

ВЫВОД: 70

# #7 - Условные операторы

В языке Ruby существует две конструкции позволяющих осуществлять проверку.

## Конструкция if - else

За счёт if else можно проверить одно или несколько условий и в случае их успешной проверки будет выполнен один, иначе другой.

Рассмотрим пример:

a = 2

b = 10

if a == b # Если a будет равным b, тогда будет выполнен код

# Здесь код что будет выполнен

elsif a <= b # Если а будет меньшим или равным b

# и при этом предыдущее условие не было выполнено,

# то здесь обработается код

else

# Код здесь сработает лишь в том случае, если другие условия не были выполнены

end

Вы можете прописать структуру лишь с одним условием if, и дописать в неё сколько-угодно вложенных условий elsif.

Внутри каждого из условий можно прописывать другие конструкции if-else, что будут проверять новые выражения.

Если необходимо проверить несколько условий в одном операторе, то можно воспользоваться логическим «и» или же логическим «или»:

if a != b and a > b

# Код будет выполнен, если и первое, и второе условие окажутся верными

end

if a < b or a == b

# Код будет выполнен, если или первое, или второе условие окажется верным

end

isSmall = true

if isSmall //isSmall == true //!isSmall same isSmall != true

  puts("OK")

end

## След пример:

x = 12

y = 3

if x > y

  puts("Bigger")

  puts("!")

end

isSmall = true

if isSmall and x !=8

  puts("OK")

end

x = 23

y = 3

if x < y

  puts("Smaller")

  puts("!")

elsif x == y

  puts("same")

elsif x == 23

  puts("x = 5")

  if x > 12

    puts("123")

  end

elsif x == 3

  puts("x = 3")

elsif x == 6

  puts("x = 6")

else

  puts("Bigger")

end

isSmall = true

if isSmall and x !=8

  puts("OK")

end

Существует еще такая форма записи условия: [действие] if [условие]:

day = '1'

puts "Monday" if day == '1'

## Конструкция case

Конструкция case обладает более удобным форматом для проверки множественных условий на совпадение значения. В конструкцию записывается переменная, что проверяется, а также значения на которые происходит проверка.

Пример оператора:

x = 23

case x # Проверяем переменную x

when 1 # Если переменная будет равна 1, то сработает код ниже

# Может быть несколько строк, а не только одна

print("Переменная равна 1")

when 56 # Если переменная будет равна 56, то здесь сработает код

# Может быть несколько строк, а не только одна

print("Переменная равна 1")

# По аналогии таких проверок может быть сколько-угодно.

# Также можно добавить проверку, которая сработает в случае

# когда все остальные проверки не сработают:

else

print("Что-то другое")

end

end

def getDayWeek(day) //тут использовали string, но можно и с int day

  nameOfDay = ""

  case day

  when "1"

    nameOfDay = "Monday"

  when "2"

    nameOfDay = "Tuesday"

  when "3"

    nameOfDay = "Wednesday"

  when "4"

    nameOfDay = "Thursday"

  when "5"

    nameOfDay = "Friday"

  when "6", "7"

    nameOfDay = "weekend"

  else

    nameOfDay = "Wrong"

  end

  return nameOfDay

end

puts getDayWeek("56")

# #8 - Цикл While и цикл For

В языке **Ruby**, как и в большинстве других языков, существует 3 вида циклов. Каждый из них выполняет одну и ту же роль, но записывается по-разному.

## Цикл For

В цикле for все условия записываются в одном месте, что очень удобно во многих случаях. Удобно для работы с массивами. Стандартная запись такого цикла:

for i in 0..10

puts i

end

В объявлении цикла записывается следующее: переменная цикла, её начальное значение и диапазон. В примере выше будут выведены числа от 0 до 10.

names = ["Bob", "Helena", "Alex", "Rina"]

for name in names

//записываем в переменную name значения массива names и выводим его

  name += "!"

  puts name

end

puts names //массив без восклицательных знаков

names = ["Bob", "Helena", "Alex", "Rina"]

for el in 0..names.length() - 1

  names[el] += "!"

end

puts names

## Цикл While

Суть цикла while не особо отличается от цикла for. Единственное отличие заключается в способе записи цикла. В while необходимо прописать лишь условие, а все остальные параметры записываются вне цикла:

i = 1 # Создание переменной

while i <= 10 # Здесь только условие

puts i

i += 1 # Увеличение переменной

end

i = 0

while i <= 5

  puts i

  i += 1

end

secret = "Blue"

guess = ""

while guess != secret

  puts("Input your word: ")

  guess = gets.chomp() //chomp - delete <enter>

end

puts "You win!"

## Цикл times

Цикл times схож на цикл for и в плане работы ничем не отличается. Цикл times нельзя найти в других языках программирования, он присутствует только в языке Ruby.

10.times do |el|

puts el

end

Такой цикл выведет значение от 0 и до 10 включительно.

6.times do |index| //сделай 6 раз, index - переменная, с которой работаем

  puts index //выводим значение index

end

names = ["Bob", "Helena", "Alex", "Rina"]

names.each do |nm|

//каждый элемент массива мы перебираем и записываем в переменную nm

  puts nm += "!"

//добавляем ! и выводим на экран

end

# #9 - Работа с файлами (чтение и запись)

При помощи Ruby можно работать с файлами на компьютере или же на сервере.

Работа с файлами - распространенная вещь в программировании. Нам постоянно что-то надо сохранять или же брать информацию из сохраненных документов.

Перед началом работы с любым текстовым документом важно изначально открыть документ. При этом, если его не будет, то будет вызвана ошибка. Чтобы программа не ломалась необходимо всегда проверять наличие существования файла. Это можно сделать при помощи условных конструкций или при помощи специальных форматов для открытия файлов.

Для открытия файла существует функция open, которая открывает файл разными способами. Ниже представлены все возможные способы/режимы открытия файла:

**r** - открытие файла лишь для чтения. Курсор помещается в начало файла перед первым символом.

**r+** - открытие файла для чтения и записи. Курсор помещается в начало файла перед первым символом.

**w** - открытие файла лишь для записи. Файл будет полностью перезаписан новой информацией. Если файла нет, то он будет создан.

**w+** - открытие файла для записи и чтения. Файл будет полностью перезаписан новой информацией. Если файла нет, то он будет создан.

**a** - только запись в файл. Курсор будет поставлен в конце файла, если файл существует, в противном случае создается новый файл для записи.

**a+** - запись и чтение из файла. Курсор будет поставлен в конце файла, если файл существует, в противном случае создается новый файл для записи.

**b** - режим двоичного файла (может использоваться с любой из перечисленных команд).

**t** - режим текстового файла (может использоваться с любой из перечисленных команд, кроме «b»).

Для работы с файлом нам необходимо использовать класс File, который отвечает за работу с файлами и далее метод open - позволяет открыть файл для чтения, записи и тп - (1 параметр - файл, кот-й нужно открыть, далее - режим, в кот-м открываем данный файл):

File.open("text/sample.txt", "r") do |file|

# в переменную file ruby закинет данный файл, далее с файлом мы будем работать через переменную file

  puts file # вывел какие-то метаданные

  puts file.read()

# прочитали файл, записав в 1 строку, т.е. с ней можно проводить все строковые операции

  puts file.read().include? "b"

  puts file.readline()

  puts file.readline()

# читаем построчно, прочитав 1 строку, дальше чтения идет со второй строки и т.д.

  puts file.readchar()

# посимвольно

 end # закрыли файл

File.open("text/sample.txt", "r") do |file|

  for line in file.readlines()

# массив из всех строк, построчно прочитали весь файл

    puts line

  end

end # закрыли файл

file = File.open("text/sample.txt", "r")

puts file.read()

file.close() # файл необходимо закрыть

File.open("text/sample.txt", "a") do |file|

# режим записи, записываем данные в конец файла - append

  file.write("\nRuby on Rails")

end

File.open("text/index.html", "w") do |file|

# write - полностью перезаписывает файл, также может полностью создать новый файл

  file.write("<h4>Hello, world!<h4>")

end

# #10 - Отслеживание ошибок

Отслеживание ошибок помогает отлавливать проблемные участки в коде и не останавливать приложения даже в случае фатальной ошибки. Такие исключения служат для обработки всех возможных сценариев программы.

Исключения могут непредвиденно возникнуть в ходе работы программы. В случае, когда такое случается программа перестанет работать или же вовсе выключится. Чтобы такого не допустить вам всегда стоит добавлять исключения в программу.

Предположим, что в программе есть возможность деления двух чисел. Соответственно, вам стоит сразу же добавить исключение, которое будет проверять не является ли делитель числом 0.

Представим другую ситуацию. Вы хотите что-то вписать файл, но для начала работы с ним - его всегда необходимо открывать. Пропишите исключение, которое будет срабатывать если файл не найден, чтобы программа не вылетала с ошибкой при не обнаружении файла.

Конструкция самих исключений очень проста. В ней изначально прописывается проверка чего-либо, а далее отслеживание ошибки. В случае нахождения ошибки зачастую выдается какое-либо сообщение для пользователя.

Пример исключения:

begin # Пробуем что-то сделать

k = 1 / 0 # Деление на ноль

rescue ZeroDivisionError # Отслеживаем на ошибку

puts "Деление на ноль" # Мы нашли ошибку

end

list = [10, 2, 5, 8, 12]

# list["dog"] # type error

begin

  num = 10 / 0

rescue

  puts "Zero division"

end

puts "Hello"

list = [10, 2, 5, 8, 12]

begin

  list["dog"]

  num = 10 / 0

rescue TypeError

  puts "TypeError"

rescue ZeroDivisionError

  puts "Zero division"

end

puts "Hello"

list = [10, 2, 5, 8, 12]

begin

  list["dog"]

  num = 10 / 0

rescue TypeError => e # в переменную записано название ошибки и потом вывели ее

  puts e

rescue ZeroDivisionError

  puts "Zero division"

end

puts "Hello"

# #11 - Объекты и классы (ООП)

ООП складывается из объектов и классов.

Например, создаем класс Машина, в нем прописываем все характеристики машины. И дальше, создаем объекты БМВ, Шевроле и тп, и все объекты уже имеют все характеристики Машины (цвет, скорость, возможность ехать - метод, который отвечает за то, чтобы машина ехала). Будто собственный тип данных, как int num = 0.

Представим, что нам необходимо описать поведение роботов при помощи кода. Для этого мы делаем следующее:

* Создаем класс, где класс что-то в роде чертежа робота;
* Создаем объект класса, который будет представлять настоящего робота, изготовленного по чертежу (классу).

Каждый класс может иметь свои методы (функции), свойства (переменные), а также конструкторы. На основе любого класса можно создавать неограниченное количество объектов, каждый из которых будет иметь свои характеристики.

**Важный момент:** методы и функции, а также поля и переменные это одно и тоже. В классах функции называются методами, а переменные называются полями, но работают они одинаково.

**Создание класса**

Чтобы создать класс необходимо использовать ключевое слово class. Классы называются именами существительными в единственном числе, каждое слово с большой буквы без разделения пробелами. Далее внутри него можете прописать любые методы и поля, которые только захотите.

После этого на основе класса можно создавать объекты. Для этого создайте переменную, пропишите название класса и выделите память под объект:

class Some

# Тело класса

end

obj\_new = Some.new() # Создание объекта

obj\_second = Some.new() # Создание 2 объекта

class Car

  # 1 буква в верхнем регистре, остальные в нижнем

  attr\_accessor :speed, :model, :color

  # конструктор - выполняются всякий раз, когда мы создаем новый объект

  # выведет Hello дважды, т.к. при создании новых объектов Car.new()

  def initialize()

    puts "Hello"

  end

end

bmw = Car.new()

bmw.speed = 230

bmw.model = "BMW"

bmw.color = "Black"

audi = Car.new()

audi.speed = 250

audi.model = "audi"

audi.color = "white"

puts bmw.speed

puts audi.speed

class Car

  attr\_accessor :speed, :model, :color, :wheels

  # можем не только устанавливать, но и получать данные/переменные

  def initialize(speed)

    puts ("Hello " + speed.to\_s)

  end

end

bmw = Car.new(230)

bmw.speed = 230

bmw.model = "BMW"

bmw.color = "Black"

audi = Car.new(250)

audi.speed = 250

audi.model = "audi"

audi.color = "white"

puts bmw.speed

puts audi.speed

ВЫВОД:

Hello 230

Hello 250

230

250

class Car

  attr\_accessor :speed, :model, :color, :wheels

  # можем не только устанавливать, но и получать данные/переменные

  def initialize(speed, model, color)

# @var — переменная объекта.

    @speed = speed

    @model = model

    @color = color

  end

end

bmw = Car.new(230, "BMW", "Black")

audi = Car.new(250, "audi", "White")

puts bmw.speed

puts audi.speed

class Car

  attr\_accessor :speed, :model, :color, :wheels

  def initialize(speed, model, color)

    @speed = speed

    @model = model

    @color = color

  end

  def is\_speed\_car?

    if @speed > 200

      return true

    end

    return false

  end

end

bmw = Car.new(199, "BMW", "Black")

audi = Car.new(250, "audi", "White")

puts bmw.is\_speed\_car?()

puts audi.is\_speed\_car?()

1.1) имя метода пишется без больших букв, разделяя слова нижним подчеркиванием

1.2) имя метода, которое возвращает булевое значение пишется с вопросительным знаком

2) при выполнении одной операции в блоке return не пишется, методы true и false возвращают соответствующие результаты

3) блочная структура условия должна быть завершенная, чтобы не возникало путаницы, если метод будет дописан

### инкапсуляция

# #12 - Наследование, модули и завершение курса

Наследование позволяет унаследовать все от родительского класса. Наследование позволяет создать класс, что будет наследовать все методы, поля и конструкторы главного класса. Такой подход очень удобен, ведь можно использовать все необходимые основные методы, но при этом расширить функционал для каждого класса-наследника. Например, класс Транспорт и класс-наследник Мотоцикл. В Ruby отсутствует множественное наследование.

Для создания наследования используется следующая структура:

class Car

attr\_accessor :wheels

# Создаем общее поле,

# так как у всех машин есть колеса

end

class BMW < Car # Класс наследник

# Поле для установки значения про "М" серию

attr\_accessor :is\_m\_serias

# Поле выше будет видно только в классе BMW

end

class Transport

  attr\_accessor :color, :wheels

  def sayBipBip

    puts "Bip Bip"

  end

end

class Car < Transport

  attr\_accessor :is\_mechanic

  def saySome

    puts @color

  end

end

class Moto < Transport

  def sayBipBip

    puts "Bip Bip 2"

  end

end

bmw = Car.new()

audi = Car.new()

yamaha = Moto.new()

bmw.color = "Black"

audi.color = "Red"

puts bmw.color

puts audi.saySome

puts audi.sayBipBip

puts yamaha.sayBipBip

Модули - есть гигантское приложение с кучей методов, было бы очень здорово вынести все эти методы в отдельный файл и в принципе потом просто ссылаться на этот файл (или модулю) - и просто брать оттуда необходимые нам методы.

Т.о. упростим код основного файла - не будет кучи различных методов - методы, которые принадлежат к автомобилям вынести в один модуль, к людям - в другой модуль. Соответственно просто по модулю обращаться к нужному нам методу и, при этом, эти методы никак не будут пересекаться между собой - т.е. в одном и в другом модуле можно иметь одинаковые методы с одинаковыми именами, и они не будут мешать друг другу.

Создаем новый файл tools.rb:

module Tools

  def sayHello(name)

    puts "Hi, #{name}"

  end

  def sayBye(name)

    puts "Bye, #{name}"

  end

end

Подключаем нужный файл, а потом нужный модуль:

require\_relative "tools.rb"

include Tools

Tools.sayHello("Fara")

Tools.sayBye("Ana")

# Изучение Ruby On Rails / #1 - Создание веб сайта на Ruby

Фреймворк **Ruby On Rails** появился еще в 2005 году. С тех пор он постоянно улучшался и к сегодняшнему дню на его основе было создано множество веб сайтов и веб приложений.

Мы подобрали несколько самых популярных сайтов созданных на **Ruby**:

1. [HULU](https://www.hulu.com/welcome?) - стриминговое ТВ и видео;
2. [GitHub](https://github.com/) - платформа для программистов;
3. [Yellow Pages](https://www.yellowpages.com/) - онлайн газета «горячих» новостей;
4. [AirBnb](http://www.airbnb.com/) - крупнейший сайт для поиска и аренды квартир по всему миру;
5. [Ask.fm](https://ask.fm/) - популярный сервис в формате вопрос-ответ;
6. [Indiegogo](http://www.indiegogo.com/) - краудфандинговая площадка по типу KickStarter.

Эти и многие другие сайты были созданы при помощи Ruby On Rails. На сегодняшний день фреймворк является одним из наиболее популярных инструментов для создания сайтов. Фреймворк позволяет сделать сайт быстро, сайты имеют четкий и понятный код, а также могут легко быть переделаны под новые нужды.

Установка Ruby\_on\_rails на windows:

* <http://railsinstaller.org/en> - скачиваю Ruby\_on\_rails - Packages included are [**Ruby**](http://www.ruby-lang.org/) 2.3.3, [**Rails**](http://rubyonrails.org/) 5.1, [**Bundler**](http://gembundler.com/), [**Git**](http://git-scm.com/), [**Sqlite**](http://sqlite.org/), [**TinyTDS**](https://github.com/rails-sqlserver/tiny_tds), [**SQL Server Support**](https://github.com/rails-sqlserver/activerecord-sqlserver-adapter), [**DevKit**](https://github.com/oneclick/rubyinstaller/wiki/Development-Kit)
* Once the file has fully downloaded, double-click it to open the installer wizard.
* Open up the Git Bash application from your RailsInstaller folder. Having trouble finding it? Look under the "Start" menu."
* In your Git Bash prompt, type the following command and hit ENTER.

curl http://installrails.com/update\_rubygems.rb | ruby

gem install rails --no-ri --no-rdoc

This command should update several gems for you and may take a few minutes

* Next, check to see if you have the right version of Rails installed by running:

$ rails --version

* Run the cd (change directory) command in Git Bash to change directory into your folder. Move into your new application folder:

$ cd ~/d/School21/Docker/Ruby и Ruby\_on\_rails/

$ rails new app

…

To see why this extension failed to compile, please check the mkmf.log which can

be found here:

C:/RailsInstaller/Ruby2.3.3/lib/ruby/gems/2.3.0/extensions/x86-mingw32/2.3.0/sql ite3-1.4.2/mkmf.log

extconf failed, exit code 1

Gem files will remain installed in

C:/RailsInstaller/Ruby2.3.3/lib/ruby/gems/2.3.0/gems/sqlite3-1.4.2 for

inspection.

Results logged to

C:/RailsInstaller/Ruby2.3.3/lib/ruby/gems/2.3.0/extensions/x86-mingw32/2.3.0/sql ite3-1.4.2/gem\_make.out

An error occurred while installing sqlite3 (1.4.2), and Bundler cannot continue.

Make sure that `gem install sqlite3 -v '1.4.2' --source 'https://rubygems.org/'`

succeeds before bundling.

In Gemfile:

sqlite3

* Переходим в папку app и меняем в Gemfile:

I had the same issue.  
I solved this problem by changing sqlite 3 in gemfile:

FROM: gem 'sqlite3'  
TO : gem 'sqlite3', '~> 1.3.6'

then run bundle and my problem is solved.

* Командная строка - в папке app - пишем bundle и все собирается:

Bundle complete! 13 Gemfile dependencies, 66 gems now installed.

Use `bundle info [gemname]` to see where a bundled gem is installed.

Фактически у вас уже есть функциональное приложение на Rails. Чтобы убедиться, нужно запустить веб-сервер на вашей машине. Это можно осуществить, запустив следующую команду из директории и переходим на localhost:3000:

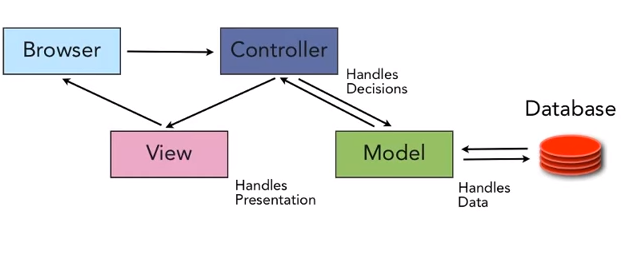
$ rails server

СЕРВЕР ЗАПУЩЕН!!! --- Ctrl^C = чтобы выключить

# #2 - MVC модель и создание страничек

В **Ruby on Rails** используется классический шаблон проектирования - MVC. Эта модель разделяет все файлы в проекте на три категории: Модель (Model), Вид (View) и Контроллер (Controller). Каждая часть отвечает за определенные действия. Рассмотрим каждую по-отдельности:

1. **Контроллер** - выполняет обработку URL запросов (переход пользователем на главную страницу, на страницу «/article» или на любую другую страницу). Контроллер понимает на какую страницу сейчас перешли и в зависимости от неё открывает определенный HTML шаблон. Перед открытием шаблона контроллер может связаться с моделью и получить значения из БД, что будут переданы позже в шаблон;
2. **Вид** - выполняет роль обычного HTML шаблона, который будет показан пользователю в качестве страницы веб сайта. Эти шаблоны вызываются при помощи контроллеров в зависимости на каком URL адресе находиться пользователь;
3. **Модель** - отвечает за функциональную часть в приложении. В моделях происходит связь с базой данных, работа с API и тому прочие моменты. Получив какие-либо значения из базы данных их можно передать обратно в контроллер и далее, они будут переданы в **Вид** (*View*).



Контроллер - отвечает за то ***КУДА*** пользователь перешел. Модель - отвечает за ***ФУНКЦИОНАЛ*** (связывается с базой данных, если это необходимо) и отдает его обратно в контроллер. Вид - HTML-шаблон с данными полученными от модели. А потом HTML-страница обратно возвращается к браузеру.

Для создания новой страницы необходимо создать контроллер, а также прописать обработку всех **URL** адресов. Каждый метод в контроллере будет вызывать определенный HTML шаблон с таким же названием файла, что и название метода. К примеру, метод «how\_are\_you» будет вызывать файл с названием «how\_are\_you.html.erb».

<http://rusrails.ru/getting-started-with-rails#dopuscheniya-v-etom-rukovodstve> - руководство по Ruby on Rails

В директории имеется несколько автоматически сгенерированных файлов и папок, задающих структуру приложения на Rails. Большая часть работы в этом самоучителе будет происходить в папке app, но сейчас пробежимся по функциям каждой папки, которые создает Rails в новом приложении по умолчанию:

|  |  |
| --- | --- |
| **Файл/Папка** | **Назначение** |
| app/ | Содержит контроллеры, модели, вьюхи, хелперы, рассыльщики, каналы, задания и ассеты вашего приложения. Мы рассмотрим эту папку подробнее далее. |
| bin/ | Содержит Rails скрипты которые стартуют ваше приложение, также директория может содержать другие скрипты которые вы используете для настройки, обновления, деплоя или запуска. |
| config/ | Конфигурации маршрутов, базы данных вашего приложения, и т.д. Более подробно это раскрыто в [Конфигурирование приложений на Rails](http://rusrails.ru/configuring-rails-applications) |
| config.ru | Конфигурация Rack для серверов, основанных на Rack, используемых для запуска приложения. Подробнее о Rack смотрите на [сайте Rack](https://rack.github.io/). |
| db/ | Содержит текущую схему вашей базы данных, а также миграции базы данных. |
| Gemfile Gemfile.lock | Эти файлы позволяют указать, какие зависимости от гемов нужны для вашего приложения на Rails. Эти файлы используются гемом Bundler. Подробнее о Bundler смотрите на [сайте Bundler](https://bundler.io/). |
| lib/ | Внешние модули для вашего приложения. |
| log/ | Файлы логов приложения. |
| package.json | Этот файл позволяет указать, какие зависимости npm необходимы для приложения Rails. Этот файл используется Yarn. Подробнее о Yarn смотрите на [сайте Yarn](https://yarnpkg.com/lang/en/). |
| public/ | Единственная папка, которая доступна извне как есть. Содержит статичные файлы и скомпилированные ассеты. |
| Rakefile | Этот файл находит и загружает задачи, которые могут быть запущены в командной строке. Определенная задача доступна во всех компонентах Rails. Вместо изменения Rakefile, можно добавить свои собственные задачи, добавив файлы в директорию lib/tasks приложения. |
| README.md | Это вводный мануал для вашего приложения. Его следует отредактировать, чтобы рассказать остальным, что ваше приложение делает, как его настроить, и т.п. |
| storage/ | Файлы Active Storage для сервиса Disk. Это раскрывается в руководстве [Обзор Active Storage](http://rusrails.ru/active_storage_overview). |
| test/ | Юнит-тесты, фикстуры и прочий аппарат тестирования. Это раскрывается в руководстве [Тестирование приложений на Rails](http://rusrails.ru/a-guide-to-testing-rails-applications) |
| tmp/ | Временные файлы (такие как файлы кэша и pid) |
| vendor/ | Место для кода сторонних разработчиков. В типичном приложении на Rails включает внешние гемы. |
| .gitignore | Этот файл сообщает git, какие файлы (явно или по шаблону) ему следует игнорировать. Подробнее об игнорировании файлов смотрите [GitHub - Ignoring files](https://help.github.com/articles/ignoring-files). |
| .ruby-version | Этот файл содержит дефолтную версию Ruby. |

Назначением контроллера является получение определенных запросов к приложению. ***Роутинг*** решает, какой контроллер получит какие запросы. Часто имеется более одного маршрута к каждому контроллеру, и различные маршруты могут быть обработаны различными экшнами. Назначением каждого экшна является сбор информации для предоставления ее во вьюху.

Назначением вьюхи является отображение этой информации в удобочитаемом формате. Необходимо отметить важное различие, что местом, в котором собирается информация, является контроллер, а не вьюха. Вьюха должна только лишь отображать эту информацию. По умолчанию шаблоны вьюх пишутся на языке, названном eRuby (Embedded Ruby), который конвертируется циклом запросов в Rails до отправки пользователю.

Нужно создать, как минимум, контроллер и вьюху (*представление*). Для создания нового контроллера, нужно запустить генератор "controller" и сказать ему, что вы хотите контроллер с именем "Welcome" с экшном по имени "index", вот так:

$ rails generate controller Welcome index

Или просто создать контроллер:

$ rails g controller Posts

* Rails создаст несколько файлов и маршрут:

create app/controllers/posts\_controller.rb

invoke erb

create app/views/posts

invoke test\_unit

create test/controllers/posts\_controller\_test.rb

invoke helper

create app/helpers/posts\_helper.rb

invoke test\_unit

invoke assets

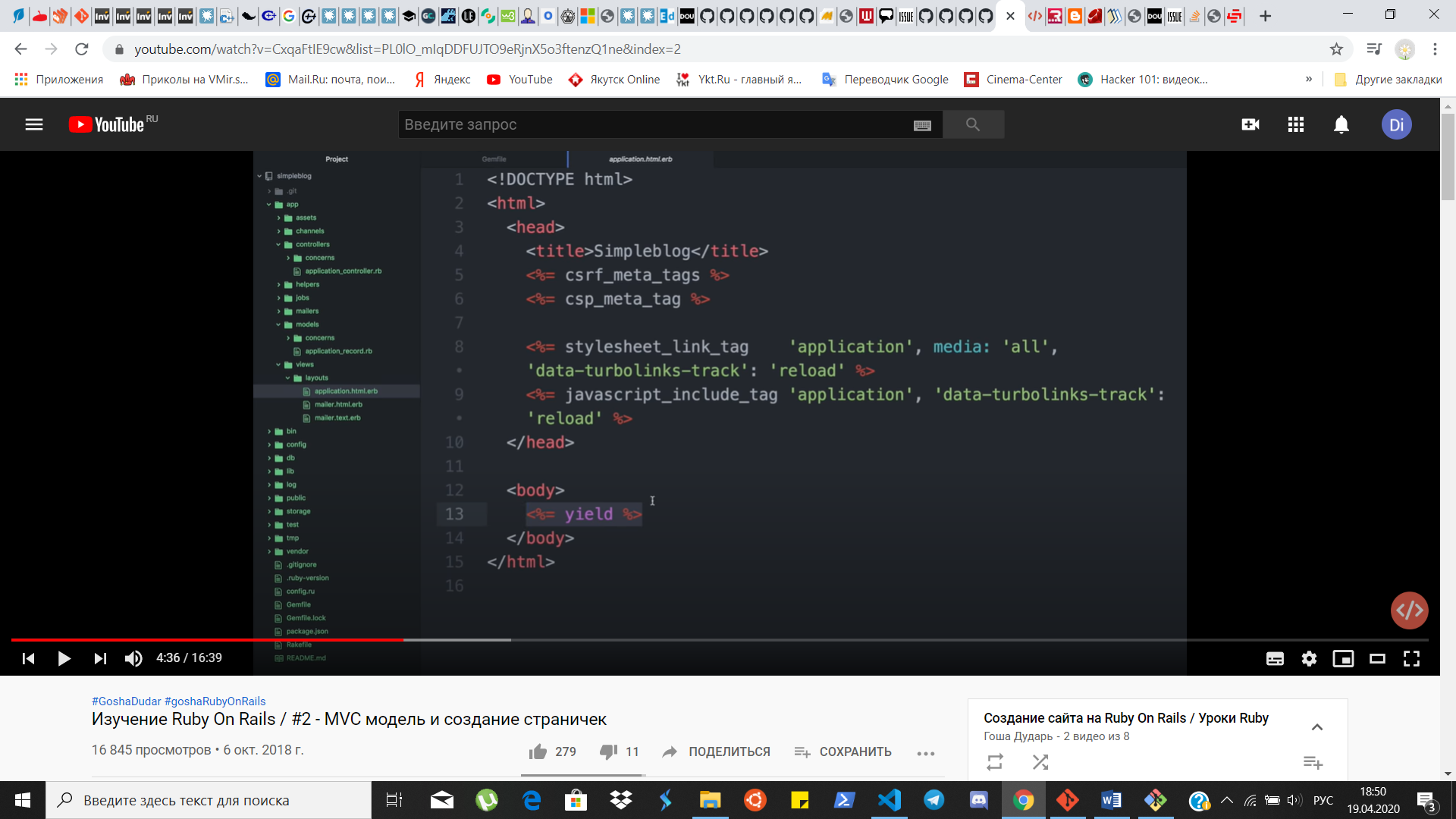
invoke coffee

create app/assets/javascripts/posts.coffee

invoke scss

create app/assets/stylesheets/posts.scss

* app/controllers - контроллеры - application\_controller.rb - самый главный котроллер от которого будем создавать другие контроллеры.
* app/models - модели - различная логика.
* app/views - вид.
* В app/views - у нас прописаны различные HTML-шаблоны (application.html.erb - динамически подключает стили и сценарии, вещи, которые могут отличаться от странички к страничке):



* Благодаря конструкции yield будет подключаться вся html-страничка.
* config/routes.rb - в этом файле записываются отслеживание различных url адресов. Это маршрутный файл вашего приложения, который содержит варианты входа на сайт на специальном языке DSL (domain-specific language, предметно-ориентированный язык программирования), который говорит Rails, как соединять входящие запросы с контроллерами и экшнами.

Контроллер - это просто класс, унаследованный от ApplicationController. В этом классе вы должны определить методы, которые станут экшнами для этого контроллера. Эти экшны будут выполнять операции CRUD со статьями в вашей системе.

В Ruby методы бывают public, private и protected, но экшнами контроллеров могут быть только public методы. Подробнее смотрите в [Programming Ruby](http://www.ruby-doc.org/docs/ProgrammingRuby/).

class PostsController < ApplicationController

  def index

  end

end

* controller/posts\_controller.rb - новый контроллер, который наследует от главного контроллера.
* метод index - отвечает за обработку главной страницы.

Обычные экшны, такие как этот (метод index), имеют связанные с ними вьюхи для отображения их информации.

Простейшим шаблоном, работающим в данном случае, будет расположенный в app/views/posts/index.html.erb. Расширение этого файла важно: первое расширение это формат шаблона, а второе расширение это обработчик, который будет использован для рендеринга шаблона. Rails пытается найти шаблон с именем posts/index в app/views приложения. Форматом для этого шаблона может быть только **html**, и обработчик по умолчанию это **erb**. Rails использует другие обработчики для других форматов. Обработчик **builder** используется для создания шаблонов **XML**, а обработчик **coffee** использует CoffeeScript для создания шаблонов **JavaScript**. Поскольку мы хотим создать новую форму HTML, будем использовать язык ERB, разработанный, чтобы внедрять Ruby в HTML.

Следовательно, файл должен называться posts/index.html.erb и должен быть расположен в директории app/views приложения.

* app/views/posts/ - создаем новый файл index.html.erb - прописываем html код:

<h2>Главная страница</h2>

* Переходим в файл routes.rb и пропишем обработку url-адреса:

Rails.application.routes.draw do

  root 'posts#index'

end

* root - отвечает ИСКЛЮЧИТЕЛЬНО за главную страницу - корневая директория, потом указываем какой контроллер отвечает за ее отображения и какой метод отвечает за html-шаблон - отображения главной страницы.

!!!!!! Запустила сервер - $ rails s

* Вышла ошибка:

ActionView::Template::Error (TypeError: Объект не поддерживает это свойство или метод):

4: <title>App</title>

5: <%= csrf\_meta\_tags %>

6:

7: <%= stylesheet\_link\_tag 'application', media: 'all', 'data-turbolinks-track': 'reload' %>

8: <%= javascript\_include\_tag 'application', 'data-turbolinks-track': 'reload' %>

9: </head>

10:

Ошибка возникает из-за проблем с поиском правильных ресурсов, которые находятся в приложении/активах. Чтобы решить эту проблему, вы можете выполнить следующие шаги. If you are running in WIndows, the coffee-script-source 1.9.0 does not work on windows.

Добавьте **gem 'coffee-script-source', '1.8.0'** в Gemfile и запустите **bundle install,** затем запустите **bundle update coffee-script-source**. Перезапустите сервер rails. Или существует грязный обходной путь, который заключается в изменении кода из:

<%= stylesheet\_link\_tag "application", media: "all", "data-turbolinks-track" => true %>

<%= javascript\_include\_tag "application", "data-turbolinks-track" => true %>

<%= csrf\_meta\_tags %>

в:

<%= stylesheet\_link\_tag "default", media: "all", "data-turbolinks-track" => true %>

<%= javascript\_include\_tag "default", "data-turbolinks-track" => true %>

<%= csrf\_meta\_tags %>

Но обратите внимание, что этот обходной путь на самом деле не решает проблему приложения, он просто заставляет ваше приложение работать без ожидаемых таблиц стилей и JavaScript.

!!!!!

* Через routes.rb - пошла обработка главной страницы, понял, что мы на нее зашли - контроллер posts с методом index - метод index сам по себе вызывает шаблон index.html.erb в папке views/posts.
* Теперь маршрут, контроллер, экшн и вьюха гармонично работают.
* Создаем новый контроллер:

$ rails g controller Pages - будут те страницы, которые не обладают дополнительной динамической информацией. Они всегда отображаются одинаково.

Он отвечает не только за одну страничку about, а, например, за другие статические страницы, которые не изменяются динамически. Например, наша страничка со статьями - она может изменяться динамически, потому что там post/1 - одна статья, post/2 - вторая статья и т.д.

controller/pages\_controller.rb:

class PagesController < ApplicationController

  def about

    @heading = "Страничка про нас!"

    @text = 'Some text...'

  end

end

@heading, @text - переменные объекта.

В папке создаем новый файл views/pages/about.html.erb:

<h3><%= @heading %></h3>

<p><%= @text %></p>

Переходим и прописываем верное отслеживание нужного нам адреса (localhost:3000/about) и вызывать нужный нам метод - routes.rb:

Rails.application.routes.draw do

  root 'posts#index'

  get 'about' => 'pages#about'

end

* 'about' - url-адрес, который мы отслеживаем.
* 'pages#about' - какой контроллер и какой метод у нас отвечает за отображение страницы, данного url-адреса.
* get - для обработки других страниц, root - исключительно за главную страницу.

#3 - Создание и сохранение постов в БД

**Ruby on Rails** имеет мощный и очень легкий в плане использования функционал для создания и сохранения какой-либо информации в базу данных.

Чтобы получать данные от пользователя вам необходимо создать форму, которая будет выполнена при помощи «form\_for» (с Rails 5.1 используется form\_with, form\_for - устарел). Такая форма может иметь множество полей и значений, которые в дальнейшем будут передаваться в базу данных в определённую таблицу.

<https://stackoverflow.com/questions/43868976/rails-5-form-for-vs-form-with>

<http://rusrails.ru/rails-form-helpers>

<%= form\_for :some, url: main\_path do |f| %>

<%= f.text\_area(:message, {:class => 'form-control'}) %>

<%= f.submit("Готово", {:class => 'btn btn-warning'}) %>

<% end %>

За передачу данных в MVC отвечает Модель. При помощи модели создается новая таблица в базе данных и при сохранении нового объекта на основе модели создается новая запись в соответствующей таблице.

Сама передача данных выполняется автоматически, единственное что необходимо выполнить, так это создать специальные методы обработки данных в Контроллере.

<http://rusrails.ru/getting-started-with-rails#dopuscheniya-v-etom-rukovodstve>

После того, как вы увидели, как создать контроллер, экшн и вьюху, давайте создадим что-то более вещественное.

Теперь в приложении app, мы создадим новый ресурс. ***Ресурс*** - это термин, обозначающий коллекцию схожих объектов, таких как статьи, люди или животные. Можно создавать, читать, обновлять и уничтожать элементы для ресурса, и эти операции называются операциями ***CRUD (create, read, update, destroy)***.

<https://docs.microsoft.com/ru-ru/azure/architecture/best-practices/api-design>

<https://www.yiiframework.com/doc/guide/2.0/ru/rest-resources>

<https://www.youtube.com/watch?v=7qYhLDpLV6E&list=PL0lO_mIqDDFUJTO9eRjnX5o3ftenzQ1ne&index=3>

Rails представляет метод **resources**, который используется для объявления стандартного ресурса **REST**. Вам необходимо добавить ресурс posts в config/routes.rb, этот файл будет выглядеть так:

Rails.application.routes.draw do

  root 'posts#index'

  get 'about' => 'pages#about'

# в дальнейшем нам потребует отслеживать много url-адресов. Н-р, для создания постов у нас есть url-адрес и url-метод, кот-й наз-ся new, update, destroy и т.д.-много адресов для их удаления, обновления, изменения - соотв-но мы можем прописать их всех вручную: get 'posts/:id/new' => 'posts#new' - и переходя по ссылке мы бы создавали пост - однако это все долго и сложно, потому есть более простой метод, кот-й позволит прописать отслеживания сразу всех нужным нам url-адресов:

  resources :posts #проиписываем с каким контроллером мы работаем

end

$ rails routes

Prefix Verb URI Pattern Controller#Action

root GET / posts#index

about GET /about(.:format) pages#about

posts GET /posts(.:format) posts#index

POST /posts(.:format) posts#create

new\_post GET /posts/new(.:format) posts#new

edit\_post GET /posts/:id/edit(.:format) posts#edit

post GET /posts/:id(.:format) posts#show

PATCH /posts/:id(.:format) posts#update

PUT /posts/:id(.:format) posts#update

DELETE /posts/:id(.:format) posts#destroy

Сперва нам необходимо где-то создать новую статью. Наиболее подходящим местом будет /posts/new. С уже определенным маршрутом уже можно осуществлять запросы к /posts/new приложения.

* Заходим в app/controller/posts\_controller.rb и прописываем метод new:

class PostsController < ApplicationController

  def index

  end

  def new

  end

end

* Заходим в app/views/posts/ и создаем файл new.html.erb:

<h3>Новый пост</h3>

* Создаем первую форму.

<http://rusrails.ru/rails-form-helpers>

Формы в веб-приложениях - это основной интерфейс для пользовательского ввода. Однако, разметка форм может быстро стать нудной в написании и поддержке из-за необходимости обрабатывать имена элементов управления формы и их бесчисленные атрибуты. Rails устраняет эту сложность, предоставляя хелперы вьюх для генерации разметки форм. Однако, поскольку эти хелперы имеют разные принципы использования, разработчикам нужно знать различия между похожими методами хелперов, прежде чем начать их использовать.

ПРИМЕЧАНИЕ: This is really in preparation for rails 5.1 where only form\_with should be used. It's meant to serve as a replacement for the two methods form\_for and form\_tag.

**form\_for** and **form\_tag** in Rails were very similar, both allowed you to create a form tag but the first one uses model’s attributes to build create or update form, while the second one simply creates an HTML form tag with the passed URL as action.

Для создания формы в этом шаблоне, мы будем использовать ***form builder***. Основной ***form builder*** для Rails представлен методом хелпера по имени **form\_with**. Для использования этого метода добавьте код в app/views/posts/new.html.erb:

<h3>Новый пост</h3>

<%= form\_with scope: :post, url: posts\_path, local: true do |form| %>

<p>

  <%= form.label :title %><br>

  <%= form.text\_field :title %>

</p>

<p>

  <%= form.label :text %><br>

  <%= form.text\_area :text %>

</p>

<p>

  <%= form.submit %>

</p>

<% end %>

При вызове **form\_with** - передали в него определяющий объект для этой формы. В нашем случае это символ :post. Это сообщает хелперу **form\_with**, для чего эта форма. Внутри блока для этого метода, объект **FormBuilder** - представленный как **form** - используется для создания двух меток и двух текстовых полей, по одному для заголовка и текста статьи. Наконец, вызов **submit** на объекте **form** создаст кнопку отправки формы.

Хотя, у этой формы есть одна проблема. Если посмотрите на сгенерированный HTML, просмотрев исходник страницы, то увидите у формы атрибут action, указывающий на **/posts/new**. Это проблема, поскольку этот маршрут ведет на ту же самую страницу, и этот маршрут должен использоваться только для отображения формы для новой статьи.

Форме нужно использовать иной URL, чтобы вести куда-то еще. Это можно быстро сделать с помощью опции **:url** для **form\_with**. Обычно в Rails, экшн, используемый для подтверждения формы **new**, такой как эта, называется "**create**", поэтому форма должна указывать на этот экшн.

Отредактируйте строчку **form\_with** в **app/views/posts/new.html.erb** следующим образом:

<%= form\_with scope: :post, url: posts\_path, local: true do |form| %>

В этом примере в опцию **:url** передан хелпер **posts\_path**.

Хелпер **posts\_path** сообщает Rails указать форме на URI Pattern, связанный с префиксом **posts**; и форма пошлет (по умолчанию) запрос POST на этот маршрут. Он связан с экшном **create** текущего контроллера, PostsController.

ПРИМЕЧАНИЕ: По умолчанию **form\_with** отправляет формы с использованием Ajax, тем самым не осуществляя редирект всей страницы. Чтобы облегчить чтение данного руководства, мы отключили это с помощью **local: true**.

* Теперь нужно определить экшн **create** в классе **PostsController** в **app/controllers/posts\_controller.rb**, ниже экшна **new**, как показано:

  def new

  end

  def create

    @post = Post.new(post\_params)

    @post.save

    redirect\_to @post

    # render plain: params[:post].inspect

  end

  private def post\_params

      params.require(:post).permit(:title, :text)

    end

end

* Указываем, какой должен быть отклик. Мы просто добавили экшн **create**, но не указали ничего о том, каким должен быть отклик. В нашем случае, экшн **create** должен сохранить нашу новую статью в базу данных.

При отправке формы, ее поля будут посланы в Rails как параметры. К этим параметрам можно обратиться из экшнов контроллера, как правило, для выполнения определенных задач. Чтобы увидеть, на что похожи эти параметры, измените экшн **create** так:

 # render plain: params[:post].inspect

Тут метод **render** (функция отладчик, в будущем нигде не используется) принимает очень простой хэш с ключом **:plain** и значением **params[:post].inspect**. Метод **params** это объект, представляющий параметры (или поля), приходящие от формы. Метод **params** возвращает объект **ActionController::Parameters**, позволяющий получать доступ к ключам хэша с использованием или строк, или символов. В этой ситуации имеют значение только параметры, пришедшие от формы.

Убедитесь, что у вас есть твердое понимание метода params, так как вы будете его использовать очень часто. Давайте рассмотрим в качестве примера **URL: http://www.example.com/?username=dhh&email=dhh@email.com**. В этом URL, params[:username] будет равен "dhh" и params[:email] будет равен "dhh@email.com".

Если еще раз отправить форму, вы увидите что-то вроде следующего:

<ActionController::Parameters {"title"=>"First Article!", "text"=>"This is my first article."} permitted: false>

Теперь этот экшн отображает параметры для статьи, пришедшие из формы. Однако, это все еще бесполезно. Да, вы видите параметры, но по сути ничего не делаете с ними.

* **Создание модели Post**

Контроллер отвечает за отслеживание url-страниц, модели отвечают за функционал. Н-р, сохранение данных в базу данных в этом примере.

**Модели в Rails** используют имя в единственном числе, а их соответствующая таблица в базе данных - имя во множественном числе. Rails предоставляет генератор для создания моделей, которым пользуются большинство разработчиков на Rails для создания новых моделей. Для создания новой модели, запустите эту команду в своем терминале:

$ rails g model Post title:string text:text

invoke active\_record

create db/migrate/20200420125524\_create\_posts.rb

create app/models/post.rb

invoke test\_unit

create test/models/post\_test.rb

create test/fixtures/posts.yml

С помощью этой команды мы сообщаем Rails, что хотим модель **Post** с атрибутом **title** строкового типа и атрибутом **text** текстового типа. Эти атрибуты автоматически добавятся в таблицу **posts** и привяжутся к модели **Post**.

Rails в ответ создаст ряд файлов. Сейчас нам интересны только app/models/post.rb и db/migrate/20140120191729\_create\_posts.rb (у вас имя может немного отличаться). Последний ответственен за создание структуры базы данных, поэтому мы и рассмотрим его далее.

**Active Record** достаточно сообразителен, чтобы автоматически связать имена столбцов с атрибутами модели, что означает, что внутри моделей Rails не нужно объявлять атрибуты, Active Record сделает это автоматически.

* **Запуск миграции**

Как вы уже видели, **rails generate model** создал файл миграции базы данных в директории **db/migrate**.

**Миграции** - это класс Ruby, разработанный для того, чтобы было просто создавать и модифицировать таблицы базы данных. Rails использует команды **rake** для запуска миграций, и возможна отмена миграции после того, как она была применена к вашей базе данных. Имя файла миграции включает временную метку, чтобы быть уверенным, что они выполняются в той последовательности, в которой они создавались.

Если Вы заглянете в файл db/migrate/YYYYMMDDHHMMSS\_create\_posts.rb (помните, у вас файл имеет немного другое имя), вот что там обнаружите:

class CreatePosts < ActiveRecord::Migration[5.1]

  def change

    create\_table :posts do |t|

      t.string :title

      t.text :text

      t.timestamps

    end

  end

end

Эта миграция создает метод **change**, вызываемый при запуске этой миграции. Экшны, определенное в этой миграции, также является обратимым, что означает, что Rails знает, как отменить изменения, сделанные этой миграцией, в случае, если вы решите их отменить позже. Когда вы запустите эту миграцию, она создаст таблицу **posts** со строковым столбцом и текстовым столбцом. Она также создаст два поля временных меток для отслеживания времени создания и обновления статьи.

Сейчас нам нужно использовать команду rails, чтобы запустить миграцию (чтобы мы смогли работать с таблицей):

$ **rails db:migrate**

Rails выполнит эту команду миграции и сообщит, что он создал таблицу **Posts** с двумя полями - title и text.

== 20200420125524 CreatePosts: migrating ========================

-- create\_table(:posts)

-> 0.0016s

== 20200420125524 CreatePosts: migrated (0.0026s) =================

Так как вы работаете по умолчанию в среде development, эта команда будет применена к базе данных, определенной в разделе development вашего файла config/database.yml. Если хотите выполнить миграции в другой среде, например в production, следует явно передать ее при вызове команды: rails db:migrate RAILS\_ENV=production.

* **Сохранение данных в контроллере**

Возвратимся к PostsController, нам нужно изменить экшн create, чтобы использовать новую модель Post для сохранения данных в базе данных. Откройте app/controllers/posts\_controller.rb и измените экшн create следующим образом:

  def create

    @post = Post.new(params[:post])

    @post.save

    redirect\_to @post

  end

Вот что тут происходит: каждая модель Rails может быть инициализирована с помощью соответствующих атрибутов, которые будут автоматически привязаны к соответствующим столбцам базы данных. В первой строчке мы как раз это и делаем (помните, что **params[:post]** содержит интересующие нас атрибуты). Затем **@post.save** ответственен за сохранение модели в базу данных. Наконец, мы перенаправляем пользователя на экшн **show**, который мы определим позже.

**P** в **Post.new** заглавная, хотя все остальные ссылки на статьи используют строчное написание. В этом контексте мы ссылаемся на класс по имени **Post**, который определен **в app/models/post.rb**. Имена классов в Ruby должны начинаться с заглавной буквы.

Как мы увидим далее, **@article.save** возвращает булевое значение, показывающее, была ли сохранена модель, или нет.

Если теперь перейти на http://localhost:3000/posts/new, вы почти сможете создать новую статью. Попробуйте! Вы должны получить ошибку, которая выглядит так:

Forbidden attributes for new article

Rails имеет ряд защитных мер, помогающих писать безопасные приложения, и вы только что столкнулись с одной из них. Она называется strong parameters и требует, чтобы мы указали Rails, какие именно параметры разрешено передавать в экшны нашего контроллера.

Почему об этом нужно беспокоиться? Возможность взять и автоматически назначить все параметры контроллера в вашу модель облегчит работу программиста, но это удобство также позволяет использование в злонамеренных целях. Что, если запрос к серверу был специально создан как отправка формы новой статьи, но также включает дополнительные поля, которые нарушают целостность вашего приложения? Они будут 'массово назначены' в вашу модель и затем в базу данных среди нормальных данных — потенциально ломая ваше приложение, или еще хуже.

Нам нужно определить разрешенные параметры нашего контроллера, чтобы предотвратить ошибочное массовое назначение. В этом случае, мы хотим и разрешить, и затребовать параметры **title** и **text** для правильного использования в **create**. Синтаксис для этого представлен методами **require** и **permit**. Изменение затрагивает одну строчку в экшне **create**:

  params.require(:post).permit(:title, :text)

# require берет из формочки :posts, далее с permit указываем разрешенные поля - title и text

Это часто выносится в отдельный метод, чтобы его можно было повторно использовать в нескольких экшнах одного контроллера, например в **create** и **update**. Метод часто делают **private** (это уже не связано с массовым назначением), чтобы убедиться, что он не может быть вызван вне предназначенного контекста. Вот результат:

  def create

    @post = Post.new(post\_params) # получаем title и text

    @post.save #сохраняем в базе данных

    redirect\_to @post # перенаправляем на новую страничку - вызывает метод show

    # render plain: params[:post].inspect

  end

  private def post\_params

      params.require(:post).permit(:title, :text)

    end

end

* **Отображение статей**

Если снова отправить форму, Rails сообщит о ненайденном экшне **show**. Нам это уже не нужно, поэтому давайте добавим экшн **show** до того.

Как вы уже видели в результатах выполнения rails routes, маршрут для экшна **show** следующий:

article GET /articles/:id(.:format) articles#show

Специальный синтаксис **:id** сообщит rails, что этот маршрут ожидает параметр **:id**, который в нашем случае будет идентификатором статьи.

Как и раньше, нам необходим экшн show в app/controllers/posts\_controller.rb и его соответствующая вьюха.

* Частой практикой является помещать стандартные экшны **CRUD** в каждом контроллере в следующем порядке: **index, show, new, edit, create, update и destroy**. Можно использовать любой порядок, какой хотите, но помните, что это публичные методы; как уже упоминалось ранее, они должны быть помещены перед определением видимости **private** в контроллере.

  def show

    @post = Post.find(params[:id])

# находит нужную нам статью и может ее отобразить

  end

Нужно отметить несколько вещей. Мы использовали **Post.find** для поиска статьи, в которой мы заинтересованы, передав **params[:id]**, чтобы получить параметр **:id** из запроса. Также мы использовали переменную экземпляра (с префиксом @) для хранения ссылки на объект статьи. Мы сделали так, потому что Rails передаст все переменные экземпляра во вьюху.

Теперь создайте новый файл **app/views/posts/show.html.erb** со следующим содержимым:

<p>

  <strong>Title:</strong>

  <%= @post.title %>

</p>

<p>

  <strong>Text:</strong>

  <%= @post.text %>

</p>

С таким изменением, вы, наконец, сможете создать новую статью. Посетите http://localhost:3000/articles/new

ОШИБКА

..$ rails s

=> Booting WEBrick

=> Rails 4.0.4 application starting in development on http://0.0.0.0:3000

=> Run `rails server -h` for more startup options

=> Ctrl-C to shutdown server

A server is already running. Check /home/..name/rprojects/railsapp/tmp/pids/server.pid.

Exiting

* rm tmp/pids/server.pid
* Все запустилось.

# #4 - Добавление Bootstrap 4 стилей к сайту

Нам все еще нужен способ для отображения списка всех наших статей, давайте сделаем его.

Добавьте в **index** внутри **PostsController** в файле **app/controllers/posts\_controller.rb**. Когда мы пишем экшн **index**, обычной практикой является разместить его как первый метод в контроллере:

class PostsController < ApplicationController

  def index

    @post = Post.all

# метод all, кот-й вытащит нам абсолютно все статьи из нашей модели

  end

И, наконец, вьюха для этого экшна, расположенная в app/views/posts/index.html.erb:

<h1>Список постов</h1>

<% @post.each do |post| %>

# each - перебираем все статьи, указываем переменную ч/з кот-ю будем перебирать

<h3><%= post.title %></h3> # выводим ч/з переменную пост наш title

<p><%= post.text %></p> # выводим ч/з переменную пост наш текст статьи

<% end %> # закрыли цикл

<%%> - указали без “=” - т.к. «=» указывается тогда, когда хотим вывести какую-то информацию, а сейчас мы просто создаем цикл.

Переменные в этих двух **index** - должны соответствовать (**post**) - потому что передаем **@post** (которая содержит все статьи - Post.all), и ее перебираем в цикле.

Теперь, если перейти в **http://localhost:3000/posts**, можно увидеть список всех статей, которые вы уже создали.

* Добавляем стили от Bootstrap

Полезные ссылки:

* [Bootstrap CDN](https://www.bootstrapcdn.com/);
* Официальный [сайт Bootstrap](https://getbootstrap.com/);
* [Видеокурс по Bootstrap](https://itproger.com/course/bootstrap).

Чтобы подключить стили к проекту Ruby On Rails необходимо подключить нужный файл со стилями к основной странице, что находиться по адресу: **«app/views/layouts/application.html.erb».**

Подключив нужные стили вы можете использовать их где-угодно в своих HTML-шаблонах.

* Переходим по ссылке <https://www.bootstrapcdn.com/> - копируем путь к нужному нам css файлу и прописываем этот путь в **«app/views/layouts/application.html.erb».** Все будет работать, потому что в этот файл встраиваются все наши шаблоны.

<!DOCTYPE html>

<html>

  <head>

    <title>App</title>

    <%= csrf\_meta\_tags %>

    <%= stylesheet\_link\_tag    'https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/css/bootstrap.min.css'%>

    <%= stylesheet\_link\_tag    'application', media: 'all', 'data-turbolinks-track': 'reload' %>

    <%= javascript\_include\_tag 'application', 'data-turbolinks-track': 'reload' %>

  </head>

* Теперь необходим сам bootstrap - переходим на офиц сайт <https://getbootstrap.com/> - во вкладку examples - берем любой, и делаем посмотреть код страницы. Копируем лишь шапку. И вставляем в <body></body> - перед шаблоном **yield**

<body>

    <div class="d-flex flex-column flex-md-row align-items-center p-3 px-md-4 mb-3 bg-white border-bottom shadow-sm">

  <h5 class="my-0 mr-md-auto font-weight-bold">Пример</h5>

  <nav class="my-2 my-md-0 mr-md-3">

    <span class="p-2 text-dark"><%= link\_to "Главная страница", home\_path %></span>

    <span class="p-2 text-dark"><%= link\_to "Про нас", about\_path %></span>

  </nav>

  <span class="p-2 text-dark"><%= link\_to "Добавление статьи", new\_post\_path %></span>

</div>

  <div class="container">

// поместили в div сlass=container, чтобы все не было прижато к левому краю

    <%= yield %>

  </div>

  </body>

* Ссылки в Ruby\_on\_Rails добавляются по-другому, чем в html.
* Метод **link\_to** - один из встроенных хелперов Rails. Он создает гиперссылку, на основе текста для отображения и указания куда перейти - в нашем случае путь для контроллера posts.
* Далее текст, который будет отображаться **«Главная страница»**, в качестве 2-го параметра саму ссылку **home\_path** - как раз переход на главную страницу, суффикс \_path нам как раз и говорит, что это ссылка - чтобы ссылка сработала переходим в **routes.rb** и прописываем псевдоним **as**:

Rails.application.routes.draw do

  get 'posts/index'

  root 'posts#index', as: 'home'

  get 'about' => 'pages#about', as: 'about'

  resources :posts

//здесь мы не прописали as, т.к. потому что resources все уже создает за нас, поэтому сверху прописали просто new\_post\_path

end

* Добавляем теперь стили к форме. Для этого заходим в views/posts/new.html.erb:

<h3>Новый пост</h3>

<%= form\_with scope: :post, url: posts\_path, local: true do |form| %>

<p>

  Название<br>

  <%= form.text\_field(:title, {class:'form-control'}) %>

</p>

<p>

  Текст статьи<br>

  <%= form.text\_area(:text, {class:'form-control'}) %>

</p>

<p>

  <%= form.submit({class:'btn btn-success'}) %>

</p>

<% end %>

* Bootstrap - Т.к. объект, то прописывается в фигурных скобках. Textual form controls—like <input>s, <select>s, and <textarea>s—are styled with the .form-control class (для работы с текстовыми полями). Included are styles for general appearance, focus state, sizing, and more. All textual <input>, <textarea>, and <select> elements with .form-control are set to width: 100%; by default. Not just the width of the screen it’s in, but the width of the container it’s in. Aside from setting it’s width, .form-control also tweaks the <input>'s appearance (including the borders, padding, colors, etc.).

<https://www.w3schools.com/bootstrap/bootstrap_forms.asp>

* Bootstrap - The .btn classes are designed to be used with the <button> element. However, you can also use these classes on <a> or <input> elements (though some browsers may apply a slightly different rendering).

<https://getbootstrap.com/docs/4.3/components/buttons/>

 #5 - Валидация форм

Валидация форм это важный момент в любом веб сайте. Без проверки форм пользователи смогут отправлять неверные данные на сервер. В уроке мы научимся проверять данные введенные в форму и показывать пользователю различные ошибки.

Всю валидацию форм необходимо прописывать внутри Моделей, которые отвечают за определенные таблицы в базе данных. В модели вы можете прописать необходимую для вас проверку и в случае неверного заполнения формы выдавать ошибку пользователю.

* Переходим по пути **app/models/post.rb**:

class Post < ApplicationRecord

  validates :title, presence: true, length: {minimum: 5}

// указываем какие поля проверяем :title

end

* presence: true - означает, что проверка будет вызываться автоматически, если false - то надо будет вызывать вручную.
* Длина минимум 5 символов.

Заметьте, что класс **Post** наследован от **ApplicationRecord**. **ApplicationRecord** наследуется от **ActiveRecord::Base**, который обеспечивает огромную функциональность для ваших моделей Rails, включая основные операции для базы данных **CRUD** (Create, Read, Update, Destroy - создать, читать, обновить, уничтожить), валидации данных, сложную поддержку поиска и возможность устанавливать отношения между разными моделями.

Rails включает методы, помогающие проверить данные, которые вы передаете в модель. Эти изменения позволят быть уверенным, что все статьи имеют заголовок длиной как минимум пять символов.

* Далее, чтобы все сработало переходим по пути app/controllers/posts\_contriller.rb и добавляем дополнительную проверку:

def new

    @post = Post.new

  end

  def create

    @post = Post.new(post\_params)

    if @post.save

      redirect\_to @post

    else

      render 'new'

    end

  end

* Если мы можем сохранить наш пост, тогда мы перенаправляем на метод show, кот-й показывает нам страничку со статьей, кот-ю мы только что создали - это если мы можем сохранить, а мы можем если не возникло ошибок - была пройдена валидация.
* Если не смогли сохранить то **render ‘new’** - страница просто перезагрузится - не будет ни добавления статьи, ни сохранения, ни переадресация пользователя на нужную страничку со статьей. Просто обновление страницы.

Если перезагрузите **http://localhost:3000/posts/new** и попытаетесь сохранить статью без заголовка, Rails вернет вас обратно на форму.

* Нужно сказать пользователю, что что-то пошло не так. Для этого нужно модифицировать **app/views/posts/new.html.erb** для проверки на сообщения об ошибке:

<h3>Новый пост</h3>

<%= form\_with scope: :post, url: posts\_path, local: true do |form| %>

<% if @post.errors.any? %>

  <% @post.errors.full\_messages.each do |msg| %>

    <div class="alert alert-danger"><%= msg %></div>

  <% end %>

<% end %>

<p>

  Название<br>

  <%= form.text\_field(:title, {class:'form-control'}) %>

</p>

* Если **post** (объект, который мы передаем) содержит какие-то ошибки, тогда в цикле мы будем перебирать все ошибки, кото-е возникли **full\_messages** (берем полные названия ошибок, кот-е возникли) и перебираем каждую из них (**each**) - делаем - указываем переменную ч/з кот-ю будем все перебирать (**msg**).
* Bootstrap - <div class="alert alert-danger"><%= msg %></div - стилизуем нашу переменную **msg**, которую будем выводить при ошибке.
* Все ошибки можно выводить на экран при помощи специального массива errors.
* Далее - if @post.errors.any? - пока еще не работает, потому что он у нас еще неопределен в методе **new**, непонятно какое у него значение.
* В **app/controllers/posts\_contriller.rb** прописываем, объявляем переменную @post:

def new

    @post = Post.new

  end

* Проработаем еще нашу **Главную страницу -** заходим в app/views/posts/index.html.erb:

<h1>Список постов</h1>

<% @post.each do |post| %>

  <div class="card bg-light p-3">

    <div class="card-block">

      <h3 class="card-title"><%= post.title %></h3>

      <p class="card-text"><%= post.text %></p>

      <%= link\_to "Читать далее", post\_path(post), class:'card-link' %>

    </div>

  </div>

  <br>

<% end %>

* Bootstrap - переносим все в div класс - “card”. А в конце добавляем ссылку на «Читать далее», 2-й параметр - post\_path(post - в качестве id, наша переменная post - чтобы всегда переходило на нужную страницу), и в качестве кнопки - класс card-link.

<https://stackoverflow.com/questions/38620737/what-is-the-well-equivalent-class-in-bootstrap-4>

<https://getbootstrap.com/docs/4.4/components/dropdowns/>

В Ruby on Rails есть множество встроенных проверок для форм и про все из них вы можете прочитать на этом сайте.

Приведем небольшой пример. Предположим что у вас есть два поля, значения которых должны совпадать. В HTML эти поля могут быть прописаны следующим образом:

<%= text\_field :person, :email %>

<%= text\_field :person, :email\_confirmation %>

Для добавления валидации можно прописать проверку, что вернет True только в случае, если значения в полях будут совпадать:

class Person < ApplicationRecord

validates :email, confirmation: true

validates :email\_confirmation, presence: true

end

Для валидации можно больше ничего не прописывать, так как все необходимые проверки будут выполнены автоматически на стороне сервера.

# #6 - Редактирование и удаление постов

Любой блог должен иметь функционал для создания постов, а также их редактирования и удаления.

* Частой практикой является помещать стандартные экшны **CRUD** в каждом контроллере в следующем порядке: **index, show, new, edit, create, update и destroy**.
* Чтобы иметь возможность редактировать посты перейдем, сначала, по пути **app/views/posts/show.html.erb** и пропишем ссылку:
* <%= link\_to "Редактировать", edit\_post\_path(@post), class: 'btn btn-warning' %>
* edit\_post\_path(@post) - будет вести нас на ссылку редактирования, благодаря тому, что мы в **app/config/routes.rb** указали след-ее:

resources :posts //уже прописал все ссылки удаления, редактир-я и т.п.

* edit\_post\_path(@post) - в качестве параметра мы указали нашу переменную @post - мы ее и будем редактировать.
* Чтобы все сработало переходим в контроллер posts\_controller.rb и добавляем метод **edit**:

  def edit

   @post = Post.find(params[:id])

//ищем и записываем в переменную нужную нам статью

 end

* Т.к. мы создали новый метод, нам необходимо создать новый html-шаблон в **views - edit.html.erb**:

<h3>Редактирование поста</h3>

<%= form\_with scope: :post, url: post\_path(@post), local: true, method: :patch do |form| %>

<% if @post.errors.any? %>

  <% @post.errors.full\_messages.each do |msg| %>

    <div class="alert alert-danger"><%= msg %></div>

  <% end %>

<% end %>

<p>

  Название<br>

  <%= form.text\_field(:title, {class:'form-control'}) %>

</p>

<p>

  Текст статьи<br>

  <%= form.text\_area(:text, {class:'form-control'}) %>

</p>

<p>

  <%= form.submit({class:'btn btn-success'}) %>

</p>

<% end %>

* Берем весь код из **new.html.erb** и добавляем - **url: post\_path(@post)**, - добавляем путь к той статье, кот-ю будем редактировать
* И указывам метод заполнения полей - **method: :patch** - позволит полностью заполнить все поля.
* А можно и проще:
* <%= form\_with(model: @post, local: true) do |form| %>
* Передача объекта статьи в метод **form\_with** автоматически установит **URL** для отправки формы отредактированной статьи. Эта опция сообщает Rails, что мы хотим, чтобы эта форма была отправлена с помощью **PATCH**, метода **HTTP**, от которого ожидается, что он используется для обновления ресурсов в соответствии с протоколом **REST**.
* Также, передача объекта модели в **form\_with**, наподобие **model: @article** во вьюхе редактирования выше, заставит хелпер формы заполнить поля формы соответствующими значениями объекта. Передача в пространство имен символа, такая как **scope: :article**, что было сделано во вьюхе **new**, только создаст пустые поля формы.

ПРИМЕЧАНИЕ

HTTP определяет множество **методов запроса**, которые указывают, какое желаемое действие выполнится для данного ресурса. Несмотря на то, что их названия могут быть существительными, эти методы запроса иногда называются *HTTP глаголами*. Каждый реализует свою семантику, но каждая группа команд разделяет общие свойства: так, методы могут быть [безопасными](https://developer.mozilla.org/ru/docs/%D0%A1%D0%BB%D0%BE%D0%B2%D0%B0%D1%80%D1%8C/safe), [идемпотентными](https://developer.mozilla.org/ru/docs/%D0%A1%D0%BB%D0%BE%D0%B2%D0%B0%D1%80%D1%8C/Idempotent) или [кэшируемыми](https://developer.mozilla.org/ru/docs/%D0%A1%D0%BB%D0%BE%D0%B2%D0%B0%D1%80%D1%8C/cacheable).

[**GET**](https://developer.mozilla.org/ru/docs/Web/HTTP/Methods/GET)

Метод GET запрашивает представление ресурса. Запросы с использованием этого метода могут только извлекать данные.

[**HEAD**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/HEAD)

HEAD запрашивает ресурс так же, как и метод GET, но без тела ответа.

[**POST**](https://developer.mozilla.org/ru/docs/Web/HTTP/Methods/POST)

POST используется для отправки сущностей к определённому ресурсу. Часто вызывает изменение состояния или какие-то побочные эффекты на сервере.

[**PUT**](https://developer.mozilla.org/ru/docs/Web/HTTP/Methods/PUT)

PUT заменяет все текущие представления ресурса данными запроса. HTTP-метод PUT предназначен для создания или замены ресурса по заданному URL. Возьмем, к примеру, файл. Когда вы загружаете файл в S3 по определенному URL, вы либо хотите создать его по тому адресу, либо заменить уже имеющийся там файл. Это PUT-запрос.

[**DELETE**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/DELETE)

DELETE удаляет указанный ресурс.

[**CONNECT**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/CONNECT)

CONNECT устанавливает "туннель" к серверу, определённому по ресурсу.

[**OPTIONS**](https://developer.mozilla.org/ru/docs/Web/HTTP/Methods/OPTIONS)

OPTIONS используется для описания параметров соединения с ресурсом.

[**TRACE**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/TRACE)

TRACE выполняет вызов возвращаемого тестового сообщения с ресурса.

[**PATCH**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/PATCH)

PATCH используется для частичного изменения ресурса. PATCH — это метод, который не объявляется ни безопасным, ни идемпотентным, и позволяет производить полное или частичное обновление, возможно с побочным эффектом на смежные ресурсы. На практике, как вы можете видеть, PATCH обычно в большей степени подходит для обновления ресурсов, чем PUT.

<https://developer.mozilla.org/ru/docs/Web/HTTP/Methods>

* Далее, чтобы сохранить изменения прописываем метод **update** в **posts\_controller.rb**:

def update

    @post = Post.find(params[:id])

    if @post.update(post\_params)

      redirect\_to @post

    else

      render 'edit'

    end

  end

* Если обновляется без ошибок, то передаем в метод **show**, иначе отправляем на ту же самую страничку **edit**.
* Новый метод, **update**, используется, когда хотите обновить запись, которая уже существует, и он принимает **хэш**, содержащий атрибуты, которые вы хотите обновить. Как и прежде, если будет ошибка обновления статьи, мы хотим опять показать форму пользователю.
* Мы заново использовали метод **post\_params**, который определили ранее для экшна **create**.

Примечание: Не обязательно передавать все атрибуты в **update**. К примеру, если был вызван **@post.update(title: 'A new title')**, Rails обновит только атрибут **title**, оставив все другие атрибуты нетронутыми.

**Обновление данных**

Для редактирования, а также удаления данных в Ruby on Rails есть встроенные методы, которые делают все за несколько строчек кода. Чтобы добавить функционал редактирования постов достаточно добавить метод **edit** и прописать HTML-шаблон для редактирования.

def edit # Редактирование поста (вывод HTML шаблона)

@post = Post.find(params[:id])

end

def update # Обработка формы обновление поста

@post = Post.find(params[:id])

if(@post.update(post\_params))

redirect\_to @post

else

render 'edit'

end

end

**Использование партиалов для очистки повторения во вьюхах**

Наша страница **edit** очень похожа на страницу **new**, фактически они используют один и тот же код для отображения формы. Давайте уберем это дублирование, использовав **партиал вьюхи**. По соглашению, файлы **партиалов** начинаются со знака подчеркивания.

Подробнее о партиалах можно прочитать в руководстве [Макеты и рендеринг в Rails](http://rusrails.ru/layouts-and-rendering-in-rails).

Создайте новый файл **app/views/posts/\_form.html.erb** со следующим содержимым:

<%= form\_with model: @post, local: true do |form| %>

<% if @post.errors.any? %>

  <% @post.errors.full\_messages.each do |msg| %>

    <div class="alert alert-danger"><%= msg %></div>

  <% end %>

<% end %>

<p>

  Название<br>

  <%= form.text\_field(:title, {class:'form-control'}) %>

</p>

<p>

  Текст статьи<br>

  <%= form.text\_area(:text, {class:'form-control'}) %>

</p>

<p>

  <%= form.submit({class:'btn btn-success'}) %>

</p>

<% end %>

Все, за исключением объявления **form\_with**, осталось тем же самым. Причина, по которой можно использовать это более короткое и простое объявление **form\_with** по сравнению с другим формами, в том, что **@post** — это ресурс, соответствующий полному набору ресурсных маршрутов, и Rails способен определить, какой использовать **URI** и **method**. Подробнее об этом использовании **form\_with** смотрите в [Resource-oriented style](https://api.rubyonrails.org/classes/ActionView/Helpers/FormHelper.html#method-i-form_with-label-Resource-oriented+style).

Давайте сейчас обновим вьюху **app/views/posts/new.html.erb**, чтобы использовать этот новый **партиал**, переписав ее полностью:

<h3>Новый пост</h3>

<%= render 'form' %>

<%= link\_to 'Назад', posts\_path, class:'card-link' %>

И то же самое для вьюхи **app/views/posts/edit.html.erb**:

<h3>Редактирование поста</h3>

<%= render 'form' %>

<%= link\_to 'Назад', posts\_path, class:'card-link' %>

**Удаление данных**

Для удаления поста все ещё проще, так как вам понадобиться лишь один метод **destroy**:

def destroy

# Находим нужный пост

@post = Post.find(params[:id])

# Удаляем найденный пост

@post.destroy

# Выполняем редирект

redirect\_to posts\_path

end

Метод роутинга **delete** должен быть использован для маршрутов, уничтожающих ресурсы. Если бы его оставить обычным маршрутом **get**, станет возможным создавать следующие злонамеренные **URL**:

<a href='http://example.com/articles/1/destroy'>look at this cat!</a>

Мы используем метод **delete** для уничтожения ресурсов, и этот маршрут связывается с экшном **destroy** в **app/controllers/posts\_controller.rb**, который еще не существует. Метод **destroy** обычно последний экшн **CRUD** в контроллере, и подобно остальным публичным экшнам **CRUD**, он должен быть расположен перед любыми **private** или **protected** методами.

* Затем пропишем кнопку удаления статьи. Прописываем метод destroy в posts\_controller.rb:

def destroy

    @post = Post.find(params[:id]) //получаем id уже из url-адреса

    @post.destroy

      redirect\_to posts\_path

  end

* Находим статью по id, удаляем и перенаправляем на главную страницу со всеми статьями. Можно вызывать **destroy** на объектах **Active Record**, когда вы хотите удалить их из базы данных. Отметьте, что нам не нужно добавлять вьюху для этого экшна, так как мы перенаправляем на **экшн index**.
* Далее заходим в **show.html.erb** и прописываем ссылку-кнопку:

<%= link\_to "Удалить пост", post\_path(@post), method: :delete, data: {confirm: "Хотите удалить статью?"}, class: 'btn btn-danger' %>

* post\_path(@post) - ссылка на данную конкретную статью - прописываем метод html-запрос на удаления, а также дополнительно передаем информацю data: - принимает значение объект confirm: (всплывающее окно) - и в нем прописываем вопрос - и класс кнопки.
* post\_path принимает модель и вытягивает оттуда to\_param, который по умолчанию возвращает id записи БД

Мы передаем именованный маршрут как второй аргумент, и опции как иной аргумент. Опции **method: :delete** и **data: { confirm: 'Are you sure?' }** используются как атрибуты **html5**, поэтому при нажатии ссылки, **Rails** сначала покажет пользователю диалог подтверждения, а затем отправит ссылку с помощью метода **delete**. Это выполняется с помощью файла **JavaScript rails-ujs**, который автоматически включается в макет приложения (**app/views/layouts/application.html.erb**) при генерации приложения. Без этого файла диалог подтверждения не будет показан.

Узнайте больше про ненавязчивый JavaScript в руководстве [Работа с JavaScript в Rails](http://rusrails.ru/working-with-javascript-in-rails).

Наши поздравления, теперь вы можете создавать, просматривать все и по отдельности, обновлять и уничтожать статьи.

В основном, Rails рекомендует использовать ресурсные объекты вместо объявления маршрутов вручную. Подробнее о роутинге смотрите [Роутинг в Rails](http://rusrails.ru/rails-routing)

#7 - Добавление комментариев

Любой блог должен иметь систему комментариев, а соответственно и наш тоже. Система комментариев состоит из нескольких этапов:

* Создание формы для отправки комментария пользователем;
* Получение данных из формы;
* Добавление комментария в базу данных, при этом лишь к тому посту, под которым он был оставлен;
* Отображение всех комментариев на странице поста.

Фреймворк Ruby On Rails очень хорош в этом деле, ведь он позволяет все это проделать быстро и практически в автоматическом режиме.

Для создания комментариев необходимо создать контроллер и модель. В модели должно быть обязательное поле, связывающее каждый комментарий с определенной статьей. При создании HTML формы необходимо использовать специальный синтаксис, что поставляется вместе с фреймворком.

Добавление, редактирование и даже удаление комментариев делается точно также, как это было сделано с постами.

* Чтобы создать комментарии, создадим вторую модель Comment, содержащую ссылку на статью (все комментарии - создается табличка - записываются в базу данных), а также комментарии должны быть привязаны к конкретным статьям. Запустите следующую команду в терминале:

$ rails g model Comment username:string body:text post:references

* post:references - привязка к определенному посту. references - тип данных, как и string, и text.

Эта команда генерирует четыре файла.

|  |  |
| --- | --- |
| **Файл** | **Назначение** |
| db/migrate/20140120201010\_create\_comments.rb | Миграция для создания табл. comments в вашей базе данных (ваше имя файла будет вкл-ть др. врем. метку) |
| app/models/comment.rb | Модель Comment |
| test/models/comment\_test.rb | Каркас для тестирования модели комментария |
| test/fixtures/comments.yml | Образцы комментариев для использования в тестир-ии |

* Сначала взглянем на **app/models**/**comment.rb:**

class Comment < ApplicationRecord

  belongs\_to :post

end

Это очень похоже на модель **Post**, которую мы видели ранее. Разница в строчке **belongs\_to :post**, которая устанавливает связь **Active Record**. Вы ознакомитесь со связями в следующем разделе руководства.

Ключевое слово (**:references**), использованное в команде **bash**, это специальный тип данных для моделей. Он создает новый столбец в вашей базе данных с именем представленной модели с добавленным **\_id**, который может содержать числовые значения. Чтобы лучше понять, проанализируйте файл **db/schema.rb** после выполнения миграции.

В дополнение к модели, **Rails** также сделал миграцию для создания соответствующей таблицы БД - **db/migrate/ 20200421113547\_create\_comments.rb**:

class CreateComments < ActiveRecord::Migration[5.1]

  def change

    create\_table :comments do |t|

      t.string :username

      t.text :body

      t.references :post, foreign\_key: true

      t.timestamps

    end

  end

end

* Строчка **t.references** создает числовой столбец с именем **post\_id**, индекс для него, и ограничение внешнего ключа, указывающего на столбец **id** таблицы **posts**
* Произошло создание файла migrate, чтобы создать саму табличку в базе данных прописываем - **rails db:migrate**

Rails достаточно сообразителен, чтобы выполнять только те миграции, которые еще не были запущены для текущей базы данных.

* **Связываем модели.**

Связи Active Record позволяют Вам легко объявлять отношения между двумя моделями. В случае с комментариями и статьями, вы можете описать отношения следующим образом:

* Каждый комментарий принадлежит одной статье.
* Одна статья может иметь много комментариев.

Фактически, это очень близко к синтаксису, который использует Rails для объявления этой связи. Вы уже видели строчку кода в модели Comment (**app/models/comment.rb**), которая делает каждый комментарий принадлежащим статье:

class Comment < ApplicationRecord

  belongs\_to :post

end

* Теперь нужно отредактировать **app/models/post.rb**, добавив другую сторону связи:

class Post < ApplicationRecord

  has\_many :comments, dependent: :destroy

  validates :title, presence: true, length: {minimum: 5}

end

* Каждый пост имеет много комментариев, также комментарии уничтожаются вместе с постом. Более подробно о связях Active Record смотрите руководство [Связи (ассоциации) Active Record](http://rusrails.ru/active-record-associations).

Т.о, связали обе таблицы. Комментарии привязаны к постам, а посты могут иметь много комментариев.

* **Добавляем маршрут для комментариев.**

Далее добавляем маршрут. Снова откроем файл **config/routes.rb** и отредактируем его следующим образом:

Rails.application.routes.draw do

  get 'posts/index'

  root 'posts#index', as: 'home'

  get 'about' => 'pages#about', as: 'about'

  resources :posts do

    resources :comments // отслеживание url-адресов внутри других url-адресов

  end

end

Это создаст **comments** как вложенный ресурс в **posts**. Это другая сторона захвата иерархических отношений, существующих между статьями и комментариями. Более подробно о роутинге написано в руководстве [Роутинг в Rails](http://rusrails.ru/rails-routing).

* **Генерируем контроллер.**

Имея модель, обратим свое внимание на создание соответствующего контроллера. Снова будем использовать тот же генератор, что использовали прежде:

$ rails generate controller Comments

Создадутся четыре файла и пустая директория:

Как и в любом другом блоге, наши читатели будут создавать свои комментарии сразу после прочтения статьи, и после добавления комментария они будут направляться обратно на страницу отображения статьи и видеть, что их комментарий уже отражен. В связи с этим, наш CommentsController служит в качестве средства создания комментариев и удаления спама, если такой будет.

Сначала мы расширим шаблон **Post show** (**app/views/posts/show.html.erb**), чтобы он позволял добавить новый комментарий:

* Копируем из **\_form.html.erb** и из **index.html.erb** в **show.html.erb:**

<hr>

  <h3>Все комментарии</h3>

  <% @post.comments.each do |comment| %>

    <div class="alert alert-light">

      <p><strong><%= comment.username %></strong>: <%= comment.body %></p>

    </div>

  <% end %>

<hr>

<h3>Добавление комментария</h3>

<%= form\_with(model: [ @post, @post.comments.build ], local: true) do |form| %>

// позволяет создавать комментарии

<p>

  Пользователь<br>

  <%= form.text\_field(:username, {class:'form-control'}) %>

</p>

<p>

  Текст комментария<br>

  <%= form.text\_area(:body, {class:'form-control'}) %>

</p>

<p>

  <%= form.submit("Добавить комментарий", {class:'btn btn-warning'}) %>

</p>

Это добавит форму на страницу отображения статьи, создающую новый комментарий при вызове экшна create в CommentsController. Тут вызов **form\_with** использует массив, что создаст вложенный маршрут, такой как **/posts/1/comments**.

Давайте напишем create в **app/controllers/comments\_controller.rb**:

class CommentsController < ApplicationController

  def create

    @post = Post.find(params[:post\_id])

    @comment = @post.comments.create(comment\_params)

      redirect\_to post\_path(@post)

  end

  private def comment\_params

    params.require(:comment).permit(:username, :body)

  end

end

Тут все немного сложнее, чем вы видели в контроллере для статей. Это побочный эффект вложения, которое вы настроили. Каждый запрос к комментарию отслеживает статью, к которой комментарий присоединен, таким образом сначала решаем вопрос с получением статьи, вызвав **find** на модели **Post**.

Кроме того, код пользуется преимуществом некоторых методов, доступных для связей. Мы используем метод **create** на **@post.comments**, чтобы создать и сохранить комментарий. Это автоматически связывает комментарий так, что он принадлежит к определенной статье.

Как только мы создали новый комментарий, мы возвращаем пользователя обратно на оригинальную статью, используя хелпер **post\_path(@post)**. Как мы уже видели, он вызывает экшн **show** в **PostsControlle**r, который, в свою очередь, рендерит шаблон **show.html.erb**. В этом месте мы хотим отображать комментарии, поэтому давайте добавим следующее в **app/views/posts/show.html.erb**:

<h3>Все комментарии</h3>

  <% @post.comments.each do |comment| %>

    <div class="alert alert-light">

      <p><strong><%= comment.username %></strong>: <%= comment.body %></p>

    </div>

  <% end %>

* **Рендеринг коллекций партиалов.**

# #8 - Авторизация и завершение

Ruby on Rails предоставляет максимально простой вариант системы авторизации на сайте. По сути все функции и действия уже прописаны и вам лишь необходимо их активировать.

Для добавления авторизации необходимо добавить команду **http\_basic\_authenticate\_with** в начало страницы, а также указать логин и пароль.

http\_basic\_authenticate\_with name: "admin", password: "1q2",

except: [:index, :show]

Здесь же можно указать какие страницы и действия будут доступны для авторизованных пользователей.

Существует две конструкции:

* **«except»** - указываем страницы что будут доступны для незарегистрированных пользователей;
* **«only»** - указываем страницы что будут доступны только для зарегистрированных пользователей.

Rails предоставляет базовую аутентификационную систему HTTP, которая хорошо работает в этой ситуации.

В **PostsController** нам нужен способ блокировать доступ к различным экшнам, если пользователь не аутентифицирован. Тут мы можем использовать метод Rails **http\_basic\_authenticate\_with**, разрешающий доступ к требуемым экшнам, если метод позволит это.

Чтобы использовать систему аутентификации, мы определим ее вверху нашего **ArticlesController** в **app/controllers/posts\_controller.rb**. В нашем случае, мы хотим, чтобы пользователь был аутентифицирован для каждого экшна, кроме **index** и **show**, поэтому напишем так:

class PostsController < ApplicationController

  http\_basic\_authenticate\_with name: "admin", password: "123",

    except: [:index, :show]

  def index

    @post = Post.all

  end

Мы также хотим позволить только аутентифицированным пользователям удалять комментарии, поэтому в CommentsController (app/controllers/comments\_controller.rb) мы напишем:

class CommentsController < ApplicationController

  http\_basic\_authenticate\_with name: "admin", password: "123", only: :destroy

  def create

    @post = Post.find(params[:post\_id])

    @comment = @post.comments.create(comment\_params)

      redirect\_to post\_path(@post)

  end

  private def comment\_params

    params.require(:comment).permit(:username, :body)

  end

end

<https://www.youtube.com/watch?v=CiI7rJqRwWc&list=PL0ZEIs68t9d1qh6Z92-E45R8YBPvgy4wk> - посмотреть после Гоши