

[КАК СТАТЬ АВТОРОМ](#)[Финальный питч-дек Битвы](#)[Офер за неделю для бэкендеров](#)

0

Рейтинг

Evrone[Подписаться](#)**Evrone**

28 окт 2022 в 19:05

Курс по Ruby+Rails. Часть 5. Паттерн MVC

🕒 14 мин 👁 2.9K

Блог компании Evrone, Ruby*, Ruby on Rails*

[Тutorial](#)

Паттерн MVC

evrone
→ ruby course

MVC – это главный архитектурный принцип, вокруг которого строится не только Ruby on Rails, но и любой другой фреймворк, работающий со сложными структурами данных и их отображением. Этот архитектурный паттерн появился довольно давно, на заре объектно-ориентированного программирования, но он не сразу был принят веб-программистами.

Довольно долго они работали с данными и веб-страницами в общей куче. Каждая веб-страница содержала код множества операций, которые рисовали нужный пользовательский

интерфейс. Разделение обязанностей между элементами приложения отдавалась на усмотрение программиста без строгой фиксации каких-либо паттернов или стандартов. Однако с течением времени разработчики перешли к более удобной работе – с моделями и с паттерном MVC как стандартом де-факто.

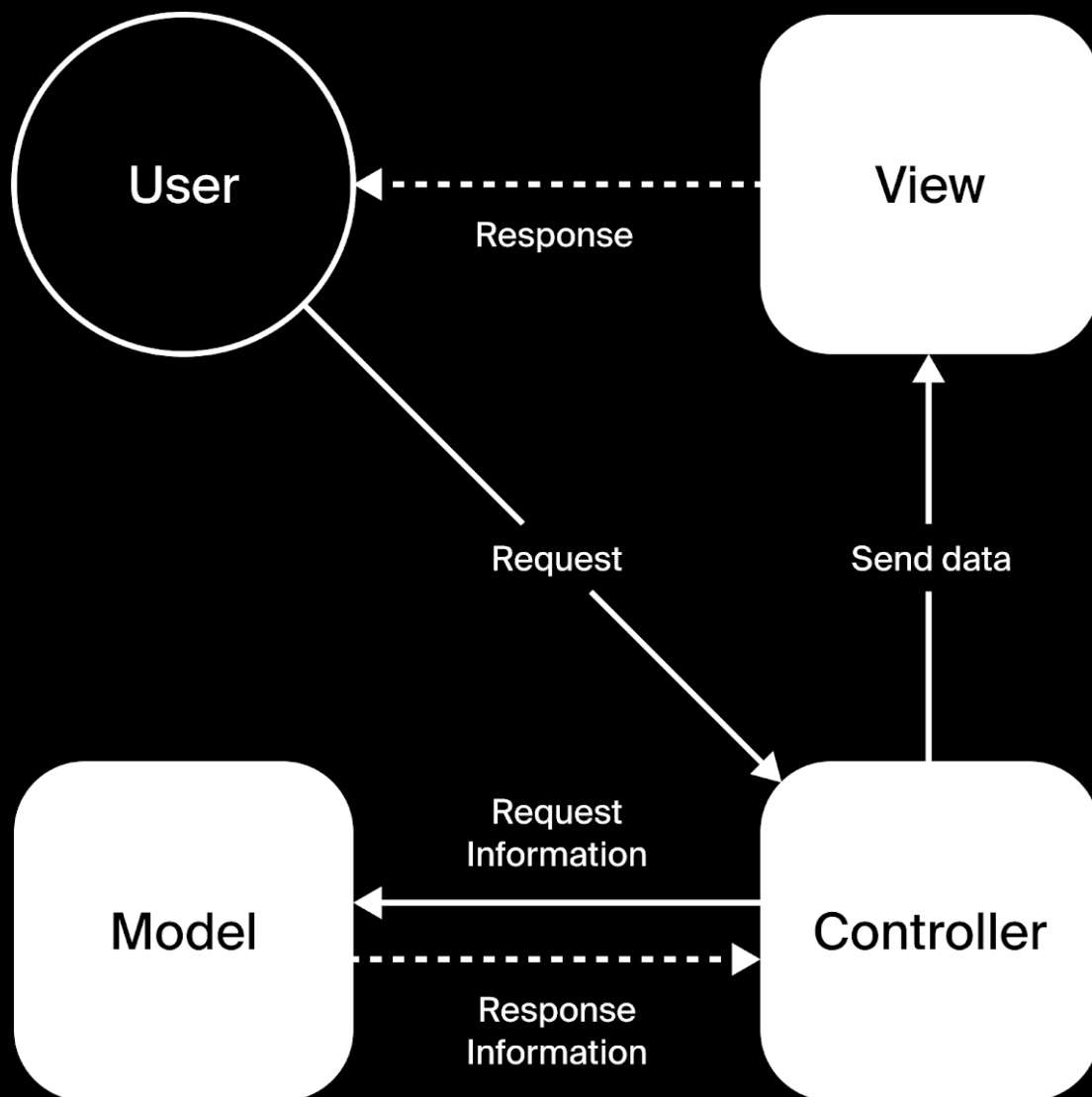


Аббревиатура MVC (или `model-view controller`) отражает систему, состоящую из модели, отображения (`view`) и контроллера. Разбирая MVC-приложение, мы столкнёмся с множеством файлов, каждый из которых определяет либо `model`, либо `view`, либо `controller`. Время от времени будут встречаться промежуточные сущности, примеры которых мы увидим чуть позже.

Процесс работы сложного веб-приложения разделен на работу с данными, представленными в виде моделей, с отображениями, с которыми взаимодействует конечный пользователь, и контроллерами – специальными механизмами, связывающими модель и отображение.

На диаграмме видно, как взаимодействуют эти сущности. Модель хранит данные, контроллер управляет вводом-выводом, а `view` – отображает. Обязательный элемент тут – пользователь, с которого начинается и которым заканчивается сложная цепочка взаимодействий. Давайте подробнее разберёмся во взаимодействиях и рассмотрим каждый элемент в отдельности:

Model-View-Controller



М — model

Модель — это сущность для работы с данными. Она хранит данные, контролирует их целостность и консистентность (то есть соответствие между разными моделями и статусу, который имеет приложение). Ещё модель производит и обрабатывает запросы на чтение или изменение данных к лежащему ниже механизму базы данных. Например, нельзя сохранить сущность пользователя, если для него не существует записи e-mail — это проверка на целостность.

Перед вами типичная модель:

```
# /app/models/payments.rb

# == Schema Information
#
# Table name: payments
#
#  id          :integer          not null, primary key
#  user_id     :integer          indexed
#  sum         :decimal(10, 2)
#  paid_at     :datetime
#  creator_id  :integer          indexed
#  description :string
#  created_at  :datetime
#  updated_at  :datetime
#  notify      :boolean          default(TRUE)
#

class Payment < ApplicationRecord
  include ArelHelpers::ArelTable
  include UserFinance

  belongs_to :user

  validates :sum, :paid_at, presence: true
  validates :sum, numericality: { greater_than_or_equal_to: 1 }

  after_create :notify_user, if: :notify

  has_paper_trail

  private

  def notify_user
    text = %(
      Hi, #{user.profile.name}! :slightly_smiling_face:
      :money_with_wings: Money was transferred from your balance.
      Details here: #{RoutesUrlHelper.new.work_url}
    )
    NotifierService.send_slack_message user, text
  end
end
```

Этот код взят из живого проекта Evrone. Здесь много непонятных пока для вас слов, но мы выделим несколько основных элементов. Мы видим здесь описание связей между этой

моделью и другой моделью `user`. Видим проверку консистентности и целостности данных – объявление `validates`. Также мы видим бизнес-логику: что делает эта модель после сохранения данных, а также обращения к некоторым библиотечным функциям. Ещё тут можно увидеть служебный метод для нотификации пользователя.

В Ruby on Rails для создания моделей мы чаще всего пользуемся библиотекой `ActiveRecord`. Подробно мы разберём её в одной из будущих лекций, а пока нам будет достаточно знать, что она реализует шаблон проектирования «object-relation mapper». Это значит, что модель формирует единую сущность из объектно-ориентированного кода на Ruby и таблицы в базе данных.

В примере вы видите сущность, которая реализует модель `payment` – на нижнем уровне это таблица с некоторыми необходимыми полями. В ходе взаимодействия базы данных и кода этого ruby-объекта и создаётся модель.

V — view

Перейдём к `view` или «отображениям». Их задача – взаимодействовать с пользователем. Пользователем может быть человек, который видит наше приложение через браузер или мобильное устройство.

Пользователем может быть и не человек, а сервис или микросервис, который обращается к нашему приложению по JSON API и запрашивает или отправляет нам какие-то данные.

Важно, что этот абстрактный пользователь работает с нашим приложением через «view» или «отображения». Он не лезет напрямую в базу данных, не работает напрямую с моделями. Он пользуется теми отображениями, которые наше приложение ему предоставляет.

На иллюстрации – типичное отображение:

```
-# /app/views/payments/_form.html.haml
- inline ||= false
= bootstrap_form_for [:admin, payment.user, payment], remote: true,
  data: { type: 'json',
    prompt: ({ message: 'Enter the reason of payment:',
      default: '',
      param: 'payment[description]' } if inline) }
  html: ({ class: 'd-flex' } if inline) do |f|
= f.number_field :sum, { label: 'Sum for payment',
  step: '0.01' }.merge(inline ? { label_class: 'erp-label-sm'
    class: 'form-control-sm',
    wrapper: { class: 'mb-0' }
```

```

                                style: 'width: 130px' } :
= f.check_box :notify, { label: 'Notify employee' } unless inline
= f.text_area :description unless inline
= instance_form_submit f, caption: 'Pay', class: ('btn-sm ml-2' if inline), disabled:

```

Оно представляет собой форму, которая вносит изменения в модель, которую мы рассматривали до этого – модель `payment`. Отображения могут быть статическими, пассивными – они просто показывают состояние объекта, нашу модель. А могут быть интерактивными и указывать на способы взаимодействия с моделью.

C — controller

Теперь настало время заняться контроллерами. Дело в том, что модель не образует сама своё отображение. Она не общается с пользователем напрямую. Должен быть инструмент, который поможет им взаимодействовать. Например, создаст эту модель из данных, хранящихся в СУБД, сформирует отображение, отправит его пользователю. Или такой сценарий: пользователь работает с формой в отображении, нажимает «enter» или «ok», и отправляет в наше приложение какой-то HTTP-запрос. Посредником между пользователем, моделью и отображением является контроллер. Контроллер можно представить себе как умный маршрутизатор, который управляет взаимодействиями между моделью и пользователем или отображением.

Кратко по контроллеру:

1. Он принимает, интерпретирует и валидирует всё, что вводит пользователь.
2. Он создаёт на основе имеющихся моделей отображения и отправляет их пользователям.
3. Он принимает от пользователей и отправляет моделям сообщения, вызывая те или иные методы в моделях или иную бизнес-логику.

На иллюстрации вы видите код одного из контроллеров:

```

# /app/controllers/payments_controller.rb
class PaymentsController < BaseController
  respond_to :html, :json
  before_action :load_user
  before_action :load_and_authorize_payment, except: %i[new create]

  def new
    @payment = @user.payments.build(params[:payment] ? payment_params : {})
    authorize! :create, @payment
  end
end

```

```
    render layout: false
  end

  def create
    @payment = @user.payments.build(
      { creator_id: current_user.id, paid_at: DateTime.current }.merge(payment_params)
    )
    authorize! :create, @payment
    @payment.save
    respond_with :admin, @payment, location: (request.referer unless referer_params[
  end

  # ...

  private

  # ...
end
```

В нём содержится несколько выражений, которые программируют поведение приложения при получении какого-то http-запроса или при формировании http-ответа.

Давайте повторим полученные сведения: MVC – это базовый архитектурный паттерн, на котором построен Ruby on Rails. Он определяет компонентную структуру приложения: оно состоит из моделей, отображений и контроллеров. Каждый из этих компонентов играет свою роль в стеке приложения. Модель отвечает за данные, контроллер маршрутизирует сообщения между моделью и отображением, отображение показывает состояние модели пользователю и позволяет с ней взаимодействовать.

MVC образует замкнутый поток управления внутри приложения. Он позволяет эффективно описывать компоненты, эффективно компоновать код и бизнес-логику. При этом приложение не является произвольным набором кода, в котором сложно понять, что с чем взаимодействует.

Теперь давайте представим себе маршрут, по которому движутся данные в MVC-приложении:

1. Процесс начинается с того, что пользователь видит на веб-странице какое-то состояние приложения – это работает отображение. Какой-то экран или форму, с которыми можно что-то сделать. Пользуясь доступными в отображении элементами управления, пользователь отправляет приложению HTTP-запрос.

2. Контроллер получает этот запрос и интерпретирует. Данные, которые ввёл пользователь, проходят через защитные конструкции – валидаторы. Запрос превращается в вызов методов в логике приложения.
3. Контроллер вызывает исполнение бизнес-логики. Бизнес-логика – это сложное понятие. Для простоты мы будем представлять её себе как обобщённое функциональное содержимое приложения.
4. Бизнес-логика внутри нашего приложения выполняет нужную трансформацию состояния моделей. Производимые действия на этом уровне разнообразны, но в итоге всё приводит к изменению состояния модели и сохранению этого состояния.

Настало время упомянуть важную особенность веб-технологии в целом. HTTP-протокол, в котором мы работаем, – это stateless-протокол. То есть программная логика нашего приложения живёт в пределах обработки одного HTTP-запроса: от получения запроса до отправки ответа. Важно, чтобы информация, связывающая разные запросы, некое общее состояние системы сохранялось. Бизнес-логика как раз формирует и сохраняет это состояние.

После того, как новый статус приложения сохранён в базе данных, формируется нужный ответ на запрос. Бизнес-логика возвращает данные ответа в контроллер. Контроллер на основании этих данных формирует и отправляет пользователю отображение модели, которую мы изменили. Отображение показывает пользователю новое состояние модели. Круг замыкается.

Структура бизнес-логики и паттерны проектирования

Давайте подробнее остановимся на структуре бизнес-логики. Мы уже знаем, что приложения сложны, в них много страниц, форм, кнопок. Даже если перед нами только одна кнопка, например, «вызвать такси» – под капотом, как мы уже знаем, движется сложный поток управления и трансформации.

Бизнес-логика – не какой-то отдельный элемент. Это совокупность элементов, которые выполняют трансформацию данных в соответствии с «заказом» пользователя. Она должна эффективно работать в коде приложения, поэтому важно её правильно скомпоновать и разместить – приложение должно работать быстро и корректно. И вот тут может возникнуть ряд проблем, которые могут сделать код и структуру приложения избыточно сложными.

Разработчики давно спорят, где размещать бизнес-логику. Что предлагает нам Ruby on Rails?

«По учебнику» бизнес-логика размещается в моделях. Там можно и нужно размещать код, который следит за корректностью и консистентностью данных, за тем, как формируются запросы к этим данным. Но если мы разместим всё возможное взаимодействие с данными в

моделях, то объём кода в них вырастет. Классы станут огромными, кодом будет трудно управлять. Учтём, что над программой работает часто не один человек, а целая команда. Программисты сменяют друг друга на одних и тех же участках кода. Код должен быть понятен и читаем.

При утяжелении кода модели возникает проблема привнесённой сложности – сложности не задачи, которую решает бизнес-логика, а сложность именно кода, его тяжеловесность. Работа с таким кодом требует лишнего времени. Но логику ведь надо куда-то деть? Часть её можно поместить в контроллеры. Но тут нас поджидают другие грабли – избыточно сложные контроллеры – тоже плохо. Ведь контроллер управляет запросами и сообщениями, он должен оставаться компактным маршрутизатором.

Можно некоторые элементы бизнес-логики помещать и во view. Например, какую-то сложную логику формы для взаимодействия с пользователем. Но если мы например, засунем туда формирование запроса к данным через обращения к моделям, то перегрузим view. Физически язык это позволяет, но так теряется контроль за кодом, взаимосвязанные части логики неструктурированно расползаются по приложению.

Чтобы упорядочить распределение логики между моделями, отображениями и контроллерами, есть ряд приёмов, которые называются «паттерны проектирования». Эти инструменты – следствие компромиссов. Мы с одной стороны размещаем логику наиболее правильно, а с другой – приложение работает достаточно быстро. Вот эти инструменты.

Service-object

Первый паттерн проектирования, с которым мы познакомимся – это service object или service. Он ещё может называться interactor, transaction, operation или saga.

Пример такого сервиса:

```
# Service
class CsvGenerator
  attr_reader :filename

  def call(input_data)
    headers, *body = input_data

    CSV.generate do |csv|
      csv << headers
      body.each { |row| csv << row }
    end
  end
end
```

```
end  
end
```

Это инструмент для группировки бизнес-логики вокруг одной, возможно сложной, многосоставной или продолжительной по времени операции.

До появления этого инструмента многие «организующие» операции помещали в контроллеры. Сейчас сложное взаимодействие стали выносить в специальные классы. Классы эти, как правило, имеют интерфейс, состоящий из одного публичного метода `call`. Если вспомните лекцию по функциональному программированию, то узнаете в сервисах паттерн «функциональный объект». Это значит, что объект, порождаемый этим классом, служит функцией, которая получает аргументы для своих параметров, объявленных в методе `call`, и производит с ними нужные действия.

На примере мы видим генерацию CSV в ответ на введенные данные. Этот код выделен в отдельный класс, лежит в своём файле, который правильно поименован. У других программистов, которые поддерживают этот проект, не возникает проблем с тем, чтобы найти этот код и изменить его, если это необходимо.

Получается, что мы избавили модель, view и контроллер от лишнего процедурного кода и не свели с ума других программистов, которые должны знать, где лежит часть бизнес-логики. Проблема решена!

Вызываются сервис-объекты контроллерами изнутри action'ов контроллера и возвращают порождённые данные в action контроллера для отображения пользователю.

Посмотрите на более сложный пример интерактора:

```
# Interactor  
class ShowDaysOff < BaseInteractor  
  Dry::Validation.load_extensions(:monads)  
  
  param :filter_params, proc(&:to_h)  
  
  FilterSchema = Dry::Validation.Params do  
    required(:user_id).filled(:int?)  
    required(:since).filled(:date?)  
    required(:through).filled(:date?)  
  end  
  
  def call  
    filters = yield accept_filter_params
```

```
    slots = yield apply_filters(initial_scope, filters)
    grouped_slots = slots.group_by { |slot| Project::VACATION_TYPES_BY_PROJECT_ID
    build_days_off(grouped_slots, filters)
end

private

# ...

end
```

Предыдущий сервис до этого состоял из пары операций, а в интеракторе возможно построение длинных цепочек операций, где каждая следующая операция зависит от предыдущей. Вместе с тем, по сути и то, и другое – сервис-объекты.

Value object

Следующий паттерн проектирования – это value object или «объект для значений». В таком объекте можно группировать сложные данные в виде компактной структуры. Value object'ы переносят между частями бизнес-логики пакеты связанных между собой данных. Данные в Value object'ах можно передавать из контроллера в модель, между моделями, между моделями и view и так далее.

Вот пример value object:

```
class DetailedPayment < Dry::Struct
  attribute :payment
  attribute :payment_salaries
  attribute :unrecognized_salaries
  attribute :payment_expenses
  attribute :unrecognized_expenses
  attribute? :unknown_sum

  def details
    # ... constructing details
    details
  end

  delegate :paid_at, to: :payment

  private

  def salary_details(salary)
```

```
# ...  
end  
  
def expense_details(expense)  
  # ...  
end  
end
```

Их особенность в том, что они содержат методы, которые должны возвращать данные, характерные для наших взаимодействий.

Данные они не преобразуют, являясь, по сути, просто контейнерами. Они, конечно, могут порождать какие-то данные на основе данных, например, модели или какой-то формы. Но это должны быть несложные операции, чаще всего – конвертирование типов или структур. Сложную логику сюда помещать не надо.

Presenter

Ещё один интересный паттерн – presenter. Его можно назвать близким родственником view. Он берёт какой-то объект, например, модель и «представляет» её для взаимодействия с отображением.

Посмотрите на пример такого объекта:

```
class EventPlanPresenter  
  attr_reader :object  
  
  def initialize(object)  
    @object = object  
  end  
  
  def time_until_end_of_day  
    time_diff = Time.current.end_of_day - Time.current  
    Time.at(time_diff.to_i.abs).utc.strftime '%H hours %M minutes'  
  end  
end
```

Presenter добавляет к модели некоторые методы, которые управляют её презентационной логикой (отсюда и название паттерна). Он группирует элементы для более удобной передачи данных во view. Получается, что эта логика вынесена из view, за счёт чего

отображения получаются легче. Одновременно, модель не перегружена чисто презентационным кодом.

Decorator

У presenter есть брат – паттерн decorator. Иногда их даже путают. Decorator «украшает» другие объекты (чаще всего модели) тем, что добавляет к ним какую-то функциональность. Но отличие от presenter всё таки есть: decorator содержит элементы логики преобразования данных, а не чисто презентационные.

Посмотрим:

```
class EventPlanDecorator
  attr_reader :object

  def initialize(object)
    @object = object
  end

  def last_call_days_left
    (object.last_call - Date.current).to_i
  end

  def last_call_soon?
    last_call_days_left.in?(0..14)
  end

  def users_not_going
    EventPlanUser.where(event_plan: object).refused.count
  end
end
```

И presenter, и decorator «украшают» одну и ту же сущность – event_plan. Но в presenter была «художественная» логика, которая формировала сообщение с часами и минутами, а decorator содержит вычисления и преобразования данных.

При работе с презентерами и декораторами встречается процедурная ловушка, в которую иногда попадают программисты. Они перегружают presenter'ы и decorator'ы бизнес-логикой. Другие программисты, которые будут читать ваш код, должны понимать, что декорирующий или презентующий объект охватывает какой-то компактный, понятный и законченный кусок логики, поэтому не старайтесь впихнуть в один класс слишком много.

Form object

Для работы с бизнес-логикой форм существует паттерн Form Object. Эти объекты формируются из данных, которые приходят в запросе от пользователя. Например, пользователь заполнил какую-то форму в приложении и послал данные серверу. Контроллер принял запрос, интерпретировал его, превратил в некоторую структуру данных с элементами этого запроса: поля, параметры и так далее.

Сами по себе запросы могут быть очень сложными. Элементы в них могут взаимодействовать друг с другом, поэтому надо сложно проверять корректность сложных запросов. Эту логику можно разместить в контроллер, а можно вынести как раз в form object'ы. Вот довольно большой пример одного из них:

```
class UserVacationForm < AbstractUserVacationForm
  include ActiveModel::Validations::Callbacks

  attribute :type, Integer
  attribute :hours, Float

  # ...

  validates :type, presence: true
  validate :sanitize_hours, if: -> { hours.present? }
  validate :check_vacation
  validate :check_hours, if: -> { hours.present? && begin_date.present? && end_date.present? }

  before_validation :improve_attributes

  def improve_attributes
    self.end_date = begin_date if begin_date.present? && end_date.blank?
  end

  def min_date
    if vacation?
      Date.current + Settings.vacation.vacation_buffer_period.months
    else
      Date.current.beginning_of_month - Settings.vacation.min_begin_date_for_vacation
    end
  end

  def project
    @project ||= Project.find(type)
  end

  private
```

```
# ...  
end
```

Они похожи на модель: принимают и трансформируют данные, чтобы потом передать на хранение. С другой стороны – это всё-таки не модели. Механизма сохранения данных тут нет. Цель такого объекта – предварительно обработать данные, перед тем, как они попадут в модель, интерактор, сервис и так далее. Используя form object'ы, мы разгружаем отображения и модели, так как убрали из них препроцессинг данных.

Query object

Переходим к последнему на сегодня паттерну, query object – объект-запрос. Он формирует сложные запросы к базе данных.

Библиотека active record, с помощью которой мы создаём модели в rails-приложениях, очень умная. Она хорошо конструирует сложные sql-запросы, но иногда накопленного кода не хватает, элементы бизнес-логики требуют ещё более сложных запросов, чем доступны в active record «из коробки». Тогда мы используем query object'ы. Они получают данные HTTP-запроса и на выходе формируют сложные sql-запросы, или из сложных предварительных запросов делают простые окончательные. Данные, которые они получают, отправляются в модель, интерактор или другой компонент бизнес-логики.

И снова пример:

```
class IndexQuery < BaseQuery  
  param :filter, reader: :private do  
    option :key_accountant_id, Dry::Types['coercible.integer'], optional: true  
  end  
  
  option :initial_scope, reader: :private, default: proc { Project.active.not_vacation }  
  
  delegate :key_accountant_id, to: :filter  
  private :key_accountant_id  
  
  def where(scope)  
    scope.yield_self { |sc| simple_cond(sc, :key_accountant_id, key_accountant_id) }  
  end  
end
```

Короткое summary

Главная задача всех инструментов выше – сформировать логику изящно и понятно, чтобы код можно было легко поддерживать, чтобы соблюдались базовые принципы проектирования – своеобразный этикет разработчиков.

Один из таких принципов, например, единственной ответственности. Он гласит, например, что модель должна заниматься хранением и контролем консистентности данных, но не производить бухгалтерские расчёты. Или корзина, в которую покупатель онлайн-магазина набирает товары, не должна формировать заказ или считать налоги. Эти элементы логики должны быть вынесены в соответствующие им классы.

То, что мы изучили сегодня, является архитектурным ядром Ruby on Rails. Фреймворк Ruby on Rails построен на основе архитектурного паттерна MVC. Важными архитектурными компонентами rails-приложения являются: модели, которые хранят данные и управляют их консистентностью, view или отображения, которые представляют эти данные пользователю и контроллеры, которые являются посредниками между моделями и view с одной стороны, и пользователями и моделями с другой.

Кроме этого мы разобрали, как можно управлять сложностью кода, используя некоторые паттерны проектирования. С ними код можно делать код изящным и удобным в поддержке.

Спасибо за внимание, не стесняйтесь задавать вопросы – мы обязательно ответим.

Теги: курсы программирования, ruby, ruby on rails, обучение ruby, mvc шаблон проектирования

Хабы: Блог компании Evrone, Ruby, Ruby on Rails



Evrone

Компания

Подписаться

Сайт Сайт Сайт Сайт



24

0

Карма

Рейтинг

Evrone @Evrone

Пользователь

Подписаться



 Комментарии 3

Публикации

ЛУЧШИЕ ЗА СУТКИ ПОХОЖИЕ



mr-pickles

18 часов назад

16-, 8- и 4-битные форматы чисел с плавающей запятой



Средний



15 мин



6.8K



+68



89



28



ru_vds

15 часов назад

Искусство создания понятных графиков



Средний



7 мин



3.6K

Тutorial

Перевод



+38



79



2



dlinyj

20 часов назад

Измерение скорости чтения-записи носителей с помощью утилиты dd



Средний



11 мин



4K

Кейс



+38



48



29



SLY_G

10 часов назад

Известная, но очень странная кошачья повадка: кошки, приносящие игрушку – это эволюционная загадка



5 мин



7.8K

Перевод



+30



22



40

**alizar**

19 часов назад

Автономия разработчиков. Как устроены компании нового типа

**Простой**

7 мин



4K

Мнение

**+25**

20



21

**Petr0v1**

14 часов назад

Марс всё ближе: несмотря на проблемы, запуск Starship можно считать успешным



4 мин



4K

**+16**

10



12

**ne_volkov**

19 часов назад

Как в Ozon следят за чувствительной информацией в логах и при чем тут Толкиен?

**Простой**

10 мин



3.2K

Кейс

**+15**

22



3

**Doctor_IT**

15 часов назад

Сможет ли високосная минута решить проблему синхронизации часов?



6 мин



2.8K

**+14**

15



14

**badcasedaily1**

12 часов назад

Garbage Collection и JVM

**Простой**

17 мин



2.2K

Обзор

+13

68

3



SLY_G
16 часов назад

Дайджест научпоп-новостей за неделю, о которых мы ничего не писали

8 мин

1.6K

Дайджест

+13

5

1

Показать еще

ИНФОРМАЦИЯ

Сайт	evrone.ru
Дата регистрации	2 августа 2022
Дата основания	2008
Численность	101–200 человек
Местоположение	Россия

БЛОГ НА ХАБРЕ



19 мая в 19:04
Курс по Ruby+Rails. Часть 8. Модели и первые шаги
 1.8K 0

25 апр в 14:29
Что нового в Proxmox 7.4
 6.9K 23

6 апр в 14:00
Как добавить сторонние драйверы в установочный образ VMware ESXi 8
 4.9K 18



22 мар в 19:40

Курс по Ruby+Rails. Часть 7. Модели и ActiveRecord

 2.6K  1

27 фев в 19:55

Подробный гайд по Docker на M1

 13K  6

Ваш аккаунт	Разделы	Информация	Услуги
Профиль	Статьи	Устройство сайта	Корпоративный блог
Трекер	Новости	Для авторов	Медийная реклама
Диалоги	Хабы	Для компаний	Нативные проекты
Настройки	Компании	Документы	Образовательные
ППА	Авторы	Соглашение	программы
	Песочница	Конфиденциальность	Стартапам
			Спецпроекты



Настройка языка

Техническая поддержка