Методические указания к выполнению лабораторной работы 5. Введение в Ruby on Rails.

Самарев Роман Станиславович
канд. техн. наук, доцент
samarev@acm.org
каф. ИУ-6 «Компьютерные системы и сети»
МГТУ им. Н.Э. Баумана

Москва 2013

Оглавление

Цель работы

Целью работы является получение теоретических сведений о принципах проектирования Model-View-Controller и получении практических навыков создания веб-приложения с использованием средств Ruby on Rails, создания простейших форм и выполнения вычислений со стороны серверной части приложения.

Принципы построения Ruby on Rails приложения

Идеология Ruby on Rails реализует концепцию Model-View-Controller (MVC) – модель-представление-контроллер. Принято считать, что MVC была описана в 1979 году Трюгве Реенскаугом (англ. Trygve Reenskaug), работавшим тогда над языком программирования Smalltalk в Xerox PARC.

В данной концепции модель ответственна за сохранение состояния приложения. В некоторых случаях состояния являются переходными, единственное назначение которых – связать взаимодействия с пользователем. В других случаях состояния постоянные и сохраняются вне приложения, например в СУБД.

Модель (Model) является больше чем набором данных. Она включает правила, которые налагаются на данные. Например скидка не может быть предоставлена на заказы дешевле 1000 руб, тогда модель должна содержать это ограничение. Таким образом, задачей модели является обеспечить целостность данных посредством применения определенных правил.

Представления (View) ответственны за формирование интерфейса пользователя, основанного на данных из модели. Например сайт магазина имеет список продуктов, отображаемых на экране. Этот список будет получен через модель, однако именно представление получит список у модели и предоставит в соответствующем формате отображения конечному пользователю. Также представление может предоставлять пользователю различные способы ввода данных, но никогда не занимается их обработкой. Представления завершает работу как только данные отображены пользователю Возможна ситуация, когда несколько представлений предоставляют доступ к одним и тем же данным, но с разными целями. Например в интернет-магазине товары должны отображаться по-разному в зависимости от того, зашел ли пользователь для оформления заказа или администратор для редактирования товаров.

Контроллеры (Controller) связывают отдельные элементы приложения. Контроллеры получают события из внешнего мира (например команды пользователя), взаимодействуют с моделями и активируют соответствующее представление для пользователя.
На рисунке 1 представлена описанная схема взаимодействия.

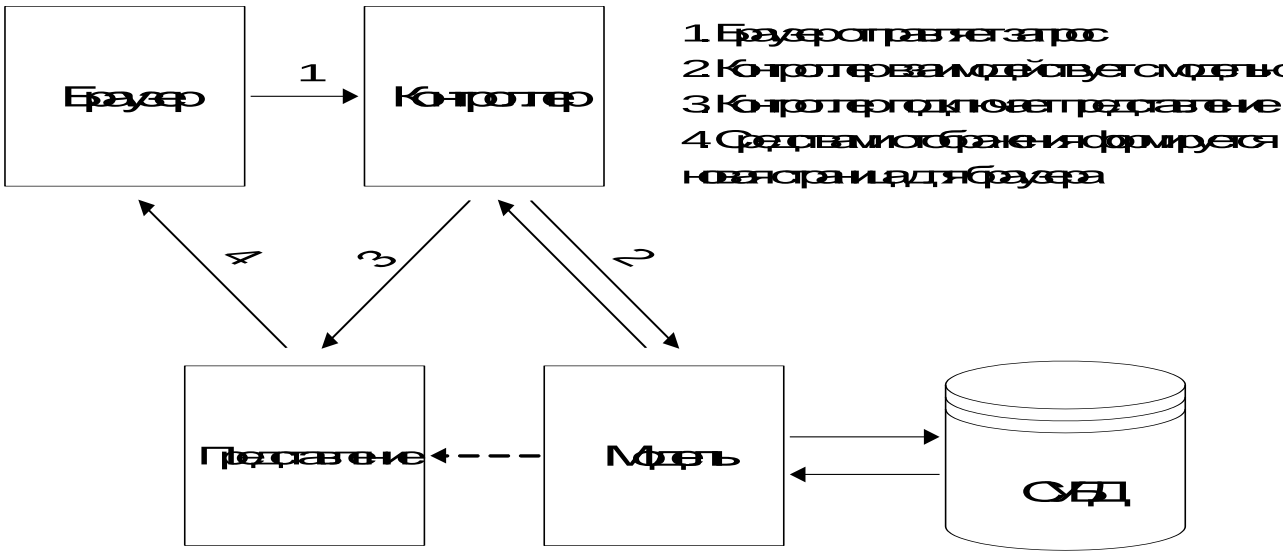

Рисунок 1 Концепция MVC

Ruby on Rails также представляет собой MVC-фреймворк. Rails задаёт структуру будущего приложения, а именно модели, представления и контроллеры как кубики функциональности, которые будут связаны вместе в момент запуска приложения. Отметим, что на данный момент используется уже 4-е поколение Rails. Примеры, рассматриваемые в данном пособии, разработаны с использованием Rails 4.0.

Для работы Rails-приложения необходимо иметь ruby, а также ряд необходимых модулей, которые могут быть установлены командой **gem install rails.**

Размещение файлов

Рассмотрим создание простейшего приложения. Rail имеет средства для автоматической генерации базовой структуры приложения. Для того, чтобы создать каркас приложения необходимо в консоли перейти в директорию, в которой должно быть размещено приложение и выполнить команду **rails new *test_app***

Будет создана директория с именем test_app, в которой и будет создана структура приложения. Во время создания приложения в консоль будут выданы примерно ниже приводимые сообщения. По ходу вывода будем иллюстрировать что это означает

Создание приложения:

```
create
```

Создание шаблона краткой информации о приложении:

```
create  README.rdoc
```

Создание файла для системы сборки Rake (аналог make для Ruby):

```
create  Rakefile
```

Вспомогательные файлы:

```
create  config.ru
create  .gitignore
```

Файл, который содержит описание необходимых приложению gem-пакетов.

```
create  Gemfile
```

Создание основного каркаса – контроллеры, представления, модели, хелперы, шаблоны типового отображения.

```
create  app
create  app/assets/images/rails.png
create  app/assets/javascripts/application.js
create  app/assets/stylesheets/application.css
create  app/controllers/application_controller.rb
create  app/helpers/application_helper.rb
create  app/mailers
create  app/models
create  app/views/layouts/application.html.erb
create  app/mailers/.gitkeep
create  app/models/.gitkeep
```

Создание конфигурационной части приложения, которая включает маршрутизацию, настройки запуска, локали.

```
create  config
create  config/routes.rb
create  config/application.rb
create  config/environment.rb
create  config/environments
create  config/environments/development.rb
create  config/environments/production.rb
create  config/environments/test.rb
create  config/initializers
create  config/initializers/backtrace_silencers.rb
create  config/initializers/inflections.rb
create  config/initializers/mime_types.rb
create  config/initializers/secret_token.rb
create  config/initializers/session_store.rb
create  config/initializers/wrap_parameters.rb
create  config/locales
create  config/locales/en.yml
create  config/boot.rb
create  config/database.yml
```

Создание директории для базы данных:

```
create  db
create  db/seeds.rb
```

Директория для будущей документации:

```
create  doc
create  doc/README_FOR_APP
```

Создание директорий для библиотек функций:

```
create  lib
create  lib/tasks
create  lib/tasks/.gitkeep
create  lib/assets
create  lib/assets/.gitkeep
```

Директория для журналов выполнения приложения:

```
      create  log
      create  log/.gitkeep
```

Директория для общедоступных статических файлов:

```
      create  public
      create  public/404.html
      create  public/422.html
      create  public/500.html
      create  public/favicon.ico
      create  public/index.html
      create  public/robots.txt
```

Директория служебных скриптов Rails:

```
      create  script
      create  script/rails
```

Директория тестов приложения:

```
      create  test/fixtures
      create  test/fixtures/.gitkeep
      create  test/functional
      create  test/functional/.gitkeep
      create  test/integration
      create  test/integration/.gitkeep
      create  test/unit
      create  test/unit/.gitkeep
      create  test/performance/browsing_test.rb
      create  test/test_helper.rb
```

Временная директория для служебных целей. Она же для хранения кэша.

```
      create  tmp/cache
      create  tmp/cache/assets
```

Директории для размещения дополнительных модулей:

```
      create  vendor/assets/javascripts
      create  vendor/assets/javascripts/.gitkeep
      create  vendor/assets/stylesheets
      create  vendor/assets/stylesheets/.gitkeep
      create  vendor/plugins
      create  vendor/plugins/.gitkeep
```

Далее run bundle install запускает специальный менеджер пакетов bundle, который проверяет состав имеющихся gem-пакетов и устанавливает непосредственно с указанного в файле Gemfile сайта все необходимые пакеты. Приведём лишь небольшой фрагмент этого вывода.

```
…
Using sprockets (2.10.0)
Using sprockets-rails (2.0.0)
Using rails (4.0.0)
Using rdoc (3.12.2)
Installing sass (3.2.10)
Installing sass-rails (4.0.0)
Installing sdoc (0.3.20)
…
Your bundle is complete! Use `bundle show [gemname]` to see where a bundled gem
is installed.
```

В минимальном виде каркас приложения создан. Теперь перейдём в директорию приложения и запустим приложение командой rails server (или rails s).

Получим примерно следующее сообщение:

```
=> Booting WEBrick
=> Rails 4.0.0 application starting in development on http://0.0.0.0:3000
=> Run `rails server -h` for more startup options
=> Ctrl-C to shutdown server
[2013-08-20 11:54:14] INFO  WEBrick 1.3.1
[2013-08-20 11:54:14] INFO  ruby 2.0.0 (2013-06-27) [i386-mingw32]
[2013-08-20 11:54:14] INFO  WEBrick::HTTPServer#start: pid=2980 port=3000
```

WEBrick – отладочный веб-сервер, который встроен в Rails и предназначен только для однопользовательской работы с целью отладки! В отличии от веб-серверов, предназначенных для эксплуатации приложений, этот веб-сервер позволяет вносить изменения в код приложения и видеть изменения без его перезапуска.

В сообщении, выданном на экран также указаны версии Rails, Ruby, а также строка http://0.0.0.0:3000. Эта строка означает, что веб-сервер присоединился на фиктивный интерфейс 0.0.0.0, означающий готовность принимать запросы, поступающие со всех имеющихся сетевых адаптеров, а значение :3000 – ip-порт. Откроем браузер и введём адрес http://localhost:3000. Получим сообщение следующего вида (рисунок 2).

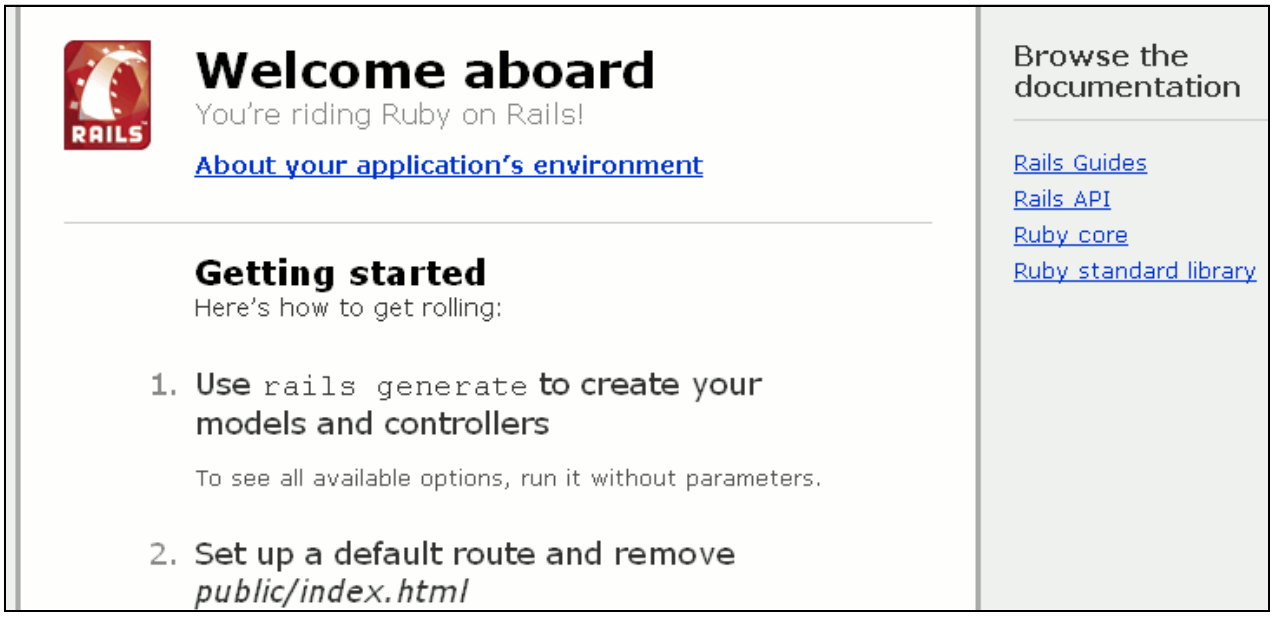

Рисунок 2 Страница по умолчанию http://localhost:3000/

Данная страница является страницей по умолчанию public/index.html (для Rails 3.2 и ранее), сгенерирована в момент создания приложения. В дальнейшем она должна быть удалена или заменена. В Rails 4 эта страница входит в состав внешних gem-модулей, поэтому единственный вариант замены — изменение корневого маршрута с помощью метода root в файле config/routes.rb.

В то же время в консоли, в которой запущен WEBrick увидим примерно следующий текст:

```
Started GET "/assets/rails.png" for 127.0.0.1 at 2013-08-20 19:48:39 +0400
Connecting to database specified by database.yml
Served asset /rails.png - 200 OK (15ms)
```

Из этого текста следует, что к серверу обратились с локального адреса 127.0.0.1 и запросили методом GET изображение "/assets/rails.png". В результате выполнения запроса было выполнено подключение к базе данных, указанной в database.yml, а изображение успешно возвращено клиенту (код 200 OK). Теперь остановим веб-сервер нажатием комбинации Ctrl-C.

Исследование Rails-приложения

В рамках знакомства с Rails воспользуемся генератором типовых контроллеров, так называемым Rails Scaffold (строительные леса). Внимание, пример приводится только для иллюстрации и в лабораторной непосредственно в работе использоваться не будет!

Для этого выполним команду:

```
rails generate scaffold User name:string email:string
```

В итоге получим примерно следующий набор сообщений. Проиллюстрируем его по ходу.

Создание спецификации базы данных:

```
      invoke  active_record
      create    db/migrate/20120817160323_create_users.rb
```

И создание эквивалентной модели:

```
      create    app/models/user.rb
```

Создание Unit-тестов приложения:

```
      invoke    test_unit
      create      test/unit/user_test.rb
      create      test/fixtures/users.yml
```

Прописывание нового маршрута:

```
      invoke  resource_route
       route    resources :users
```

Создание нового контроллера:

```
      invoke  scaffold_controller
      create    app/controllers/users_controller.rb
```

Генерация представлений:

```
      invoke    erb
      create      app/views/users
      create      app/views/users/index.html.erb
      create      app/views/users/edit.html.erb
      create      app/views/users/show.html.erb
      create      app/views/users/new.html.erb
      create      app/views/users/_form.html.erb
```

Создание функционального теста:

```
      invoke    test_unit
      create      test/functional/users_controller_test.rb
```

Создание помощника:

```
      invoke    helper
      create      app/helpers/users_helper.rb
```

Создание теста для него:

```
      invoke      test_unit
      create        test/unit/helpers/users_helper_test.rb
```

Создание «полезностей» (Assets):

```
      invoke  assets
      invoke    coffee
      create      app/assets/javascripts/users.js.coffee
      invoke    scss
      create      app/assets/stylesheets/users.css.scss
      invoke  scss
      create    app/assets/stylesheets/scaffolds.css.scss
```

Для реального создания базы данных запустим:

```
rake db:migrate
```

Произойдет создание базы данных /db/development.sqlite3 (задан в конфигурации) по файлу миграции db/migrate/20130820160323_create_users.rb. Посмотрим содержимое этого файла:

```
class CreateUsers < ActiveRecord::Migration
  def change
    create_table :users do |t|
      t.string :name
      t.string :email

      t.timestamps
    end
  end
end
```

Выделенные строки соответствуют полям, которые были указаны при запуске Scaffold-генератора. При этом будет создана таблица users, а каждая запись будет содержать name, email и timestamp. Обратите внимание на то, что в команде запуска Scaffold-генератора было использовано имя User, а таблица называется user**s**! Причина этого заключается в принятом соглашении об именах. В момент запуска генератора указывается называние сущности в единственном числе, а имя контроллера, таблица будет образовано множественным числом.

После создания базы данных можно опять запустить веб-сервер командой rails server.

Обратимся по адресу http://localhost:3000/users/. В данном случае users – имя созданного контроллера. Получим следующую страницу:

Рисунок 3 Страница http://localhost:3000/users/

Нажмем «New User» и получим форму добавления нового пользователя с полями name, email, заданными при запуске Scaffold-генератора. Добавим поочерёдно нескольких пользователей.

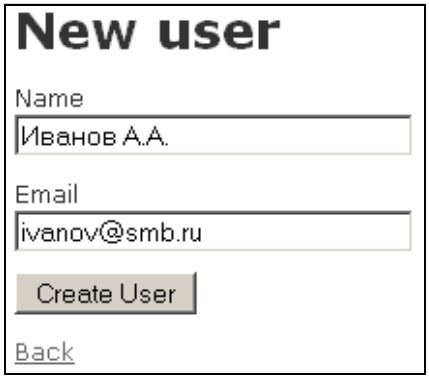

Рисунок 4 Страница http://localhost:3000/users/new

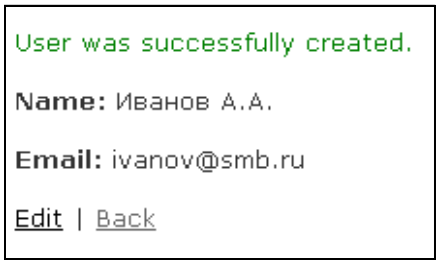

Рисунок 5 Страница http://localhost:3000/users/1

В итоге возвращаемся на страницу http://localhost:3000/users/ и видим страницу, представленную на рисунке 6.

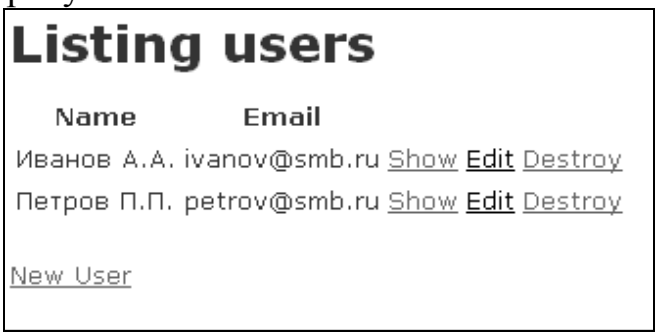

Рисунок 6 Страница http://localhost:3000/users/

В итоге можем убедиться, что сгенерированное приложение позволяет выполнять просмотр, добавление, редактирование и удаление пользователей. И это при том, что пока еще руками написаны лишь команды создания rails-приложения и запуск scaffold-генератора.

В дальнейшем будут показаны и некоторые другие генераторы, однако на примере сгенерированного приложения разберём назначение компонентов Rails.

Внесём дополнения в диаграмму взаимодействия компонентов по концепции MVC с учётом реального сгенерированного приложения Rails.

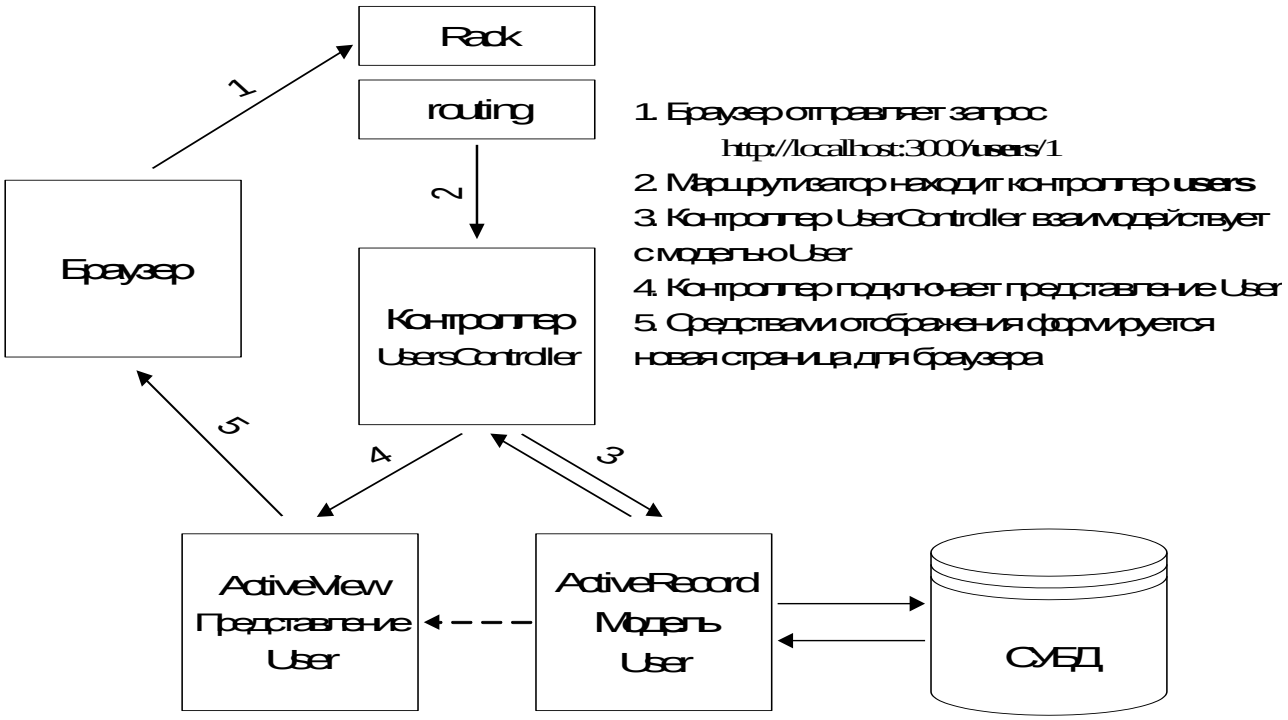

Рисунок 7 MVC на примере Rails-приложения.

Принципиальных отличий на данной диаграмме от общей концепции MVC не много. Rails приложение контролирует компонент с названием Rack, который является посредником в передаче запросов пользователей от веб-сервера к веб-приложению. В документации[1] приводятся следующие веб-серверы, с которыми rack может работать:

- Mongrel
- EventedMongrel
- SwiftipliedMongrel
- WEBrick
- FCGI
- CGI
- SCGI
- LiteSpeed
- Thin

С использованием этого компонента построено несколько фреймворков веб-приложений:

- Camping
- Coset
- Halcyon
- Mack
- Maveric
- Merb
- Racktools::SimpleApplication

[1] http://rack.rubyforge.org/doc/

- Ramaze
- Ruby on Rails
- Rum
- Sinatra
- Sin
- Vintage
- Waves
- Wee
- …

Большая их часть имеет достаточно ограниченное применение. В контексте Rails необходимо знать, что запрос от Rack поступает компоненту маршрутизации. Исходный код, который контролирует маршрутизацию, расположен в файле config/routes.rb.

Изначально этот файл содержит лишь каркас для добавления маршрутов

```
TestApp::Application.routes.draw do
…
end
```

Однако после работы Scaffold-генератора в него была добавлена строка:

```
  resources :users
```

Запись resources означает создание стандартного набора REST маршрутов, который создаст 4 именованных маршрута (в концепции CRUD - Create Read Update Destroy) и 7 действий всего: index, show, new, create, edit, update, destroy.

Маршруты выделяются из URL. Например, адреса http://localhost:3000/users/ и http://localhost:3000/users/new явно ведут на маршруты UserController#index, UserController#new, а http://localhost:3000/users/1 - неявно на UserController#show, но с подразумеваемым параметром :id.

Маршруты указывают конкретные контроллеры, которые должны обрабатывать запросы. В нашем случае существует единственный контроллер app/controllers/application_controller.rb. После автоматической генерации он содержит следующий код:

```
class UsersController < ApplicationController
  # GET /users
  # GET /users.json
  def index
    @users = User.all

    respond_to do |format|
      format.html # index.html.erb
      format.json { render json: @users }
    end
  end
```

```
  # GET /users/1
  # GET /users/1.json
  def show
    @user = User.find(params[:id])

    respond_to do |format|
      format.html # show.html.erb
      format.json { render json: @user }
    end
  end

  # GET /users/new
  # GET /users/new.json
  def new
    @user = User.new

    respond_to do |format|
      format.html # new.html.erb
      format.json { render json: @user }
    end
  end

  # GET /users/1/edit
  def edit
    @user = User.find(params[:id])
  end

  # POST /users
  # POST /users.json
  def create
    @user = User.new(params[:user])

    respond_to do |format|
      if @user.save
        format.html { redirect_to @user, notice: 'User was successfully
created.' }
        format.json { render json: @user, status: :created, location: @user }
      else
        format.html { render action: "new" }
        format.json { render json: @user.errors, status: :unprocessable_entity }
      end
    end
  end

  # PUT /users/1
  # PUT /users/1.json
  def update
    @user = User.find(params[:id])

    respond_to do |format|
      if @user.update_attributes(params[:user])
        format.html { redirect_to @user, notice: 'User was successfully
updated.' }
        format.json { head :no_content }
      else
        format.html { render action: "edit" }
        format.json { render json: @user.errors, status: :unprocessable_entity }
      end
    end
  end

  # DELETE /users/1
  # DELETE /users/1.json
  def destroy
```

```
    @user = User.find(params[:id])
    @user.destroy

    respond_to do |format|
      format.html { redirect_to users_url }
      format.json { head :no_content }
    end
  end
end
```

Класс UsersController является потомком Rails-класса ApplicationController. Обратите внимание на то, что в комментариях перед методами сгенерированного класса UsersController указаны маршруты, по которым предполагается вызов этих методов. Рассмотрим внимательно один из маршрутов и, соответственно один метод контроллера. Например index:

```
  def index
    @users = User.all

    respond_to do |format|
      format.html # index.html.erb
      format.json { render json: @users }
    end
  end
```

Первая строка метода содержит получение списка пользователей через класс User. Этот класс является моделью пользователей. Из вызова метода .all можем предположить получение списка всех пользователей. Для дальнейшего использования этого списка заведена переменная уровня экземпляра с именем @users.

Модель app/models/user.rb содержит код:

```
class User < ActiveRecord::Base
  attr_accessible :email, :name
end
```

Из этого кода можем заключить, что класс User является потомком Rails-класса ActiveRecord::Base. Более того, в классе объявлено два атрибута- :email и :name, поэтому в экземплярах этого класса к ним можно непосредственно обратиться.

Возвращаемся к контроллеру. Метод respond_to предназначен для предоставления возможности получения ответа от веб-сервера в различных форматах. Однако явно он может и не использоваться, если формат получения единственный. Здесь же предусмотрено два формата. Попробуем ввести в браузере адрес: http://localhost:3000/users**.json**. В итоге получим ответ в формате json типа:

```
[{"created_at":"2013-08-20T17:33:40Z","email":"ivanov@smb.ru","id":1,"name":
"\u0418\u0432\u0430\u043d\u043e\u0432 \u0410.\u0410.","updated_at":"2013-08-
20T17:33:40Z"},
```

```
{"created_at":"2013-08-20T17:36:57Z","email":"petrov@smb.ru","id":2,"name":
"\u041f\u0435\u0442\u0440\u043e\u0432 \u041f.\u041f.","updated_at":"2012-08-
17T17:36:57Z"}]
```

В случае format.html указан предполагается использование представления index.html.erb. Обратите внимание на то, что здесь "**# index.html.erb**" – всего лишь комментарий поясняющий, откуда будет использовано представление.

В момент генерации представления были сгенерированы для каждого маршрута, однако сейчас нас интересует лишь app/views/users/index.html.erb. Этот файл содержит код:

```
<h1>Listing users</h1>

<table>
  <tr>
    <th>Name</th>
    <th>Email</th>
    <th></th>
    <th></th>
    <th></th>
  </tr>

<% @users.each do |user| %>
  <tr>
    <td><%= user.name %></td>
    <td><%= user.email %></td>
    <td><%= link_to 'Show', user %></td>
    <td><%= link_to 'Edit', edit_user_path(user) %></td>
    <td><%= link_to 'Destroy', user, method: :delete, data: { confirm: 'Are you
sure?' } %></td>
  </tr>
<% end %>
</table>

<br />

<%= link_to 'New User', new_user_path %>
```

В коде присутствует HTML-разметка и специальная разметка <% %>, а также <%= … %>, которая содержит вызовы Ruby-кода и специальных методов типа link_to. Обратите внимание на то, что для вывода списка пользователей формируется таблица. Однако число пользователей неизвестно, поэтому здесь использован цикл генерации строк

```
<% @users.each do |user| %>
  <tr>
...
  </tr>
<% end %>
```

Подстановка имени пользователя и его электронного адреса производится в соответствии с кодом цикла @users.each do |user| строках

```
    <td><%= user.name %></td>
    <td><%= user.email %></td>
```

Обратите внимание также на то, что этот шаблон не содержит всей необходимой HTML-разметки. Шаблон отображения страниц приложения был сгенерирован на самом первом этапе генерации rails-приложения и хранится в файле app/views/layouts/application.html.erb. Его содержимое представлено ниже:

```
<!DOCTYPE html>
<html>
<head>
  <title>TestApp</title>
  <%= stylesheet_link_tag    "application", :media => "all" %>
  <%= javascript_include_tag "application" %>
  <%= csrf_meta_tags %>
</head>
<body>

<%= yield %>

</body>
</html>
```

Как видим, здесь содержится внешнее обрамление, а вставка конкретных шаблонов представлений производится вместо строки <%= yield %>.

Последнее, что следует посмотреть – сгенерированные тесты. Отметим, что по-умолчанию в Rails генерируются тесты уже устаревшей системы тестирования Unit. Для использования rspec необходимо выполнить дополнительные действия. Тем не менее, рассмотрим то, что уже предоставлено.

Для запуска тестов достаточно запустить в директории rails-приложения команду rake. Файл Rakefile содержит цель test в качестве цели по-умолчанию. Дополнительно укажем трассировку вызовов и запустим rake --trace.
Получим что-то типа:

```
** Invoke default (first_time)
** Invoke test (first_time)
** Execute test
** Invoke test:run (first_time)
** Execute test:run
** Invoke test:units (first_time)
** Invoke test:prepare (first_time)
** Invoke db:test:prepare (first_time)
** Invoke db:abort_if_pending_migrations (first_time)
** Invoke environment (first_time)
** Execute environment
** Invoke db:load_config (first_time)
** Execute db:load_config
** Execute db:abort_if_pending_migrations
** Execute db:test:prepare
** Invoke db:test:load (first_time)
** Invoke db:test:purge (first_time)
** Invoke environment
** Invoke db:load_config
** Execute db:test:purge
** Execute db:test:load
** Invoke db:test:load_schema (first_time)
** Invoke db:test:purge
```

```
** Execute db:test:load_schema
** Invoke db:schema:load (first_time)
** Invoke environment
** Invoke db:load_config
** Execute db:schema:load
** Execute test:prepare
** Execute test:units
Run options:

# Running tests:

Finished tests in 0.000000s, NaN tests/s, NaN assertions/s.

0 tests, 0 assertions, 0 failures, 0 errors, 0 skips
** Invoke test:functionals (first_time)
** Invoke test:prepare
** Execute test:functionals
Run options:

# Running tests:

.......

Finished tests in 1.843750s, 3.7966 tests/s, 5.4237 assertions/s.

7 tests, 10 assertions, 0 failures, 0 errors, 0 skips
** Invoke test:integration (first_time)
** Invoke test:prepare
** Execute test:integration
** Execute default
```

Как видим, было запущено 7 тестов. Принято 10 утверждений. Ошибок [2]не обнаружено.

Большая часть тестов ничего не содержит. Однако имеется сгенерированный функциональный test\functional\users_controller_test.rb, который содержит следующий текст:

```
require 'test_helper'

class UsersControllerTest < ActionController::TestCase
  setup do
    @user = users(:one)
  end

  test "should get index" do
    get :index
    assert_response :success
    assert_not_nil assigns(:users)
  end

  test "should get new" do
    get :new
    assert_response :success
  end
```

[2] Если при прохождении теста обнаружена ошибка доступа к БД, проверьте значение переменной окружения RAILS_ENV (должно быть development или быть пусто) или сбросьте её значение командой set RAILS_ENV=

```
  test "should create user" do
    assert_difference('User.count') do
      post :create, user: { email: @user.email, name: @user.name }
    end

    assert_redirected_to user_path(assigns(:user))
  end

  test "should show user" do
    get :show, id: @user
    assert_response :success
  end

  test "should get edit" do
    get :edit, id: @user
    assert_response :success
  end

  test "should update user" do
    put :update, id: @user, user: { email: @user.email, name: @user.name }
    assert_redirected_to user_path(assigns(:user))
  end

  test "should destroy user" do
    assert_difference('User.count', -1) do
      delete :destroy, id: @user
    end

    assert_redirected_to users_path
  end
end
```

Из приведенного текста видим, что предполагается эмуляции GET, POST, PUT, DELETE-запросов HTTP посредством вызова соответствующих методов, которым передаются URL, позволяющие проверить поведение все маршруты и методы сгенерированного контроллера.

Запуск только функциональных тестов производится командой
`rake test:functionals`

На этом в исследовании сгенерированного приложения остановимся и перейдем к справочному изложению компонентов Rails.

Настройка приложения

Bundler

Bundler не является компонентом Rails, однако рекомендован для использования. Bundle в переводе на русский означает узел, связка, пачка, а основное назначение этого средства – обеспечить необходимый приложению набор пакетов в соответствии с заданным списком. Более того, пакеты должны не только совпадать по названию, но и быть тех же версий, на которых проведена разработка и тестирования приложения.
Для решения этой задачи существует файл Gemfile. Этот файл генерируется автоматически при создании базовой структуры приложения и имеет примерно следующий вид:

```
source 'http://rubygems.org'

gem 'rails', '4.0.0'

# Bundle edge Rails instead:
# gem 'rails', :git => 'git://github.com/rails/rails.git'

gem 'sqlite3'


# Gems used only for assets and not required
# in production environments by default.
group :assets do
  gem 'sass-rails',   '~> 4.0.0'
  gem 'coffee-rails', '~> 4.0.0'

  # See https://github.com/sstephenson/execjs#readme for more supported runtimes
  # gem 'therubyracer', :platforms => :ruby

  gem 'uglifier', '>= 1.0.3'
end
```

Параметр **source** указывает на адрес репозитория пакетов. Обычно - 'http://rubygems.org'. В некоторых случаях может указывать на 'http://rubygems.org'. Считается, что использование протокола https может повысить уровень достоверности скачиваемых пакетов, поскольку сертификаты шифрования, используемые в этом протоколе, должны быть официально подписаны.

Параметр gem указывает на имя gem-пакета, который необходимо приложению. Возможные формы:

Необходима любая версия пакета:
```
gem 'sqlite3'
```

Необходима только версия пакета '4.0.0':
```
gem 'rails', '4.0.0'
```

Необходима версия пакета не раньше '4.0.0':

```
gem 'rails', '>=4.0.0'
```

Необходима версия пакета не раньше '>=3.2.8', но менее '<3.3.0':
```
gem 'rails', '~>3.2.8'
```

Взять пакет не из source, а из указанного репозитория.
```
gem 'rails', :git => 'git://github.com/rails/rails.git'
```

Взять пакет из указанного репозитория, но с явным указанием ветви:
```
gem 'rails', :git => 'git://github.com/rails/rails.git',
:branch => '2-3-stable'
```

Группы пакетов позволяют указать состав пакетов, необходимых для определенных конфигураций: test, development, production. Поскольку пользователи приложения не будут заниматься разработкой, то им не нужны пакеты, которые относятся к конфигурации test или development.

```
group :test do
  gem "rspec"
end
group :development, :test do
  gem "ruby-debug"
end
```
По умолчанию команда bundle install установит все пакеты, которые указаны в Gemfile. Если необходимо установить только в указанной конфигурации, необходимо применить параметр –without. Например:

```
bundle install --without development test
bundle install --without test
```

После выполнения команды bundle install в директории приложения создаётся файл Gemfile.lock, который содержит список конкретных пакетов с указаниями номеров версий и всеми зависимостями, которые были на момент выполнения bundle install. При переносе приложения на другой сервер это файл гарантирует, что состав пакетов будет таким же.

Для того, чтобы собрать все пакеты вместе с приложением достаточно выполнить команду bundle package. В директорию vendor/cache/ будут помещены все пакеты в формате gem, от которых зависит приложение.

Конфигурационные параметры

Конфигурационные файлы приложения содержатся в директории config. Коротко рассмотрим основные моменты[3].

В процессе запуска приложения подключаются три файла:

- config/boot.rb – устанавливает путь к Gemfile и запускает bundle/setup

[3] См. главу Rails Environments and Configuration [fernandez]

- config/application.rb – загружает все gem-пакеты rails, а также пакеты для текущей конфигурации приложения, установленной в переменной Rail.env;
- config/environment.rb – запускает все модули инициализации и само приложение.

Файл application.rb содержит настройки, применимые к приложению, независимо от конфигурации. По-умолчанию этот файл имеет следующий вид:

```
module TestApp # это имя, указанное при создании приложения!
  class Application < Rails::Application
    # Settings in config/environments/* take precedence over those specified here.
    # Application configuration should go into files in config/initializers
    # -- all .rb files in that directory are automatically loaded.

    # Custom directories with classes and modules you want to be autoloadable.
    # config.autoload_paths += %W(#{config.root}/extras)

    # Only load the plugins named here, in the order given (default is alphabetical).
    # :all can be used as a placeholder for all plugins not explicitly named.
    # config.plugins = [ :exception_notification, :ssl_requirement, :all ]

    # Activate observers that should always be running.
    # config.active_record.observers = :cacher, :garbage_collector, :forum_observer

    # Set Time.zone default to the specified zone and make Active Record auto-convert to this zone.
    # Run "rake -D time" for a list of tasks for finding time zone names. Default is UTC.
    # config.time_zone = 'Central Time (US & Canada)'

    # The default locale is :en and all translations from config/locales/*.rb,yml are auto loaded.
    # config.i18n.load_path += Dir[Rails.root.join('my', 'locales', '*.{rb,yml}').to_s]
    # config.i18n.default_locale = :de

    # Configure the default encoding used in templates for Ruby 1.9.
    config.encoding = "utf-8"

    # Configure sensitive parameters which will be filtered from the log file.
    config.filter_parameters += [:password]

    # Enable escaping HTML in JSON.
    config.active_support.escape_html_entities_in_json = true

    # Use SQL instead of Active Record's schema dumper when creating the database.
    # This is necessary if your schema can't be completely dumped by the schema dumper,
    # like if you have constraints or database-specific column types
    # config.active_record.schema_format = :sql

    # Enforce whitelist mode for mass assignment.
    # This will create an empty whitelist of attributes available for mass-assignment for all models
    # in your app. As such, your models will need to explicitly whitelist or blacklist accessible
    # parameters by using an attr_accessible or attr_protected declaration.
    config.active_record.whitelist_attributes = true

    # Enable the asset pipeline
    config.assets.enabled = true

    # Version of your assets, change this if you want to expire all your assets
    config.assets.version = '1.0'
  end
end
```

Как видим, большая часть параметров удовлетворяет значениями по-умолчанию. В будущем может понадобиться изменение языка по-умолчанию config.i18n.default_locale.

config/environments/development.rb
config/environments/production.rb

config/environments/test.rb

config/locales
config/database.yml
config/initializers/secret_token.rb
config/initializers/session_store.rb

Формы. Передача данных.

Пример приложения с формой

Разработаем приложение-калькулятор, задачей которого будет принять введенные значения и выдать результат.

1. ввести rails new calc и войти в созданную директорию calc
2. ввести rails generate controller Calc input view
3. открыть файл app/views/input.html.erb

```
<h1>Calc#input</h1>
<p>Find me in app/views/calc/input.html.erb</p>

<%= form_tag("/calc/view", :method => "get") do %>
      <%= label_tag("Value 1:") %>
      <%= text_field_tag(:v1) %>  <br/>
      <%= label_tag("Value 2") %>
      <%= text_field_tag(:v2) %>  <br/>

      <%= label_tag("+") %>
      <%= radio_button_tag(:op, "+") %><br/>
      <%= label_tag("-") %>
      <%= radio_button_tag(:op, "-") %><br/>
      <%= label_tag("*") %>
      <%= radio_button_tag(:op, "*") %><br/>
      <%= label_tag("/") %>
      <%= radio_button_tag(:op, "/") %><br/>
      <br/>

      <%= submit_tag("Calc result") %>
<% end %>
```

4. открыть файл app/views/view.html.erb

```
<h1>Calc#view</h1>
<p>Find me in app/views/calc/view.html.erb</p>

<p id="result"><%= @result %></p>

<%= link_to "Repeat calculation", :calc_input %>
```

5. Открыть файл app/controllers/calc_controller.rb

```
class CalcController < ApplicationController
  def input
  end

  def view
    v1 = params[:v1].to_i
    v2 = params[:v2].to_i
    @result = case params[:op]
      when "+" then v1 + v2;
      when "-" then v1 - v2;
      when "*" then v1 * v2;
      when "/" then v1 / v2;
      else "Unknown!"
    end
  end
end
```

Проверяем результат.
rails server –e development

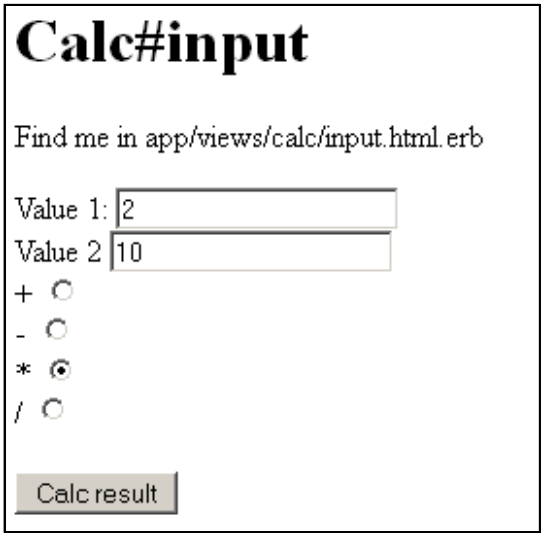

Рисунок 8 Страница http://localhost:3000/calc/input

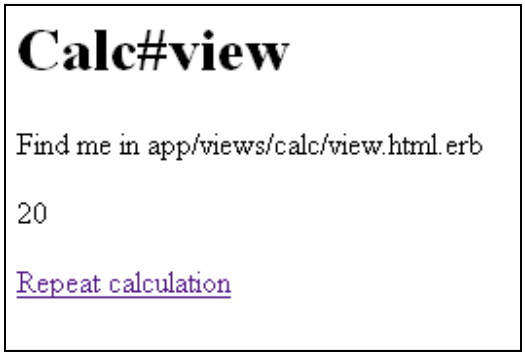

Рисунок 9 Страница http://localhost:3000/calc/view после нажатия "Calc result"

Пояснения к примеру

В приведенном примере для написания представления app/views/calc/input.html.erb использовано средство eRuby (Embedded Ruby). Это средство обеспечивает возможность написания шаблонов, в которых делаются вставки кода на языке Ruby и, который будет выполнен в процессе подстановки шаблона.

Шаблон представляет собой текст, имеющий вставки специального формата. Различают вставки для кода:
<% ruby code %>
% ruby code

И вставки для выражений.
<%= ruby expression %>

Код, расположенный внутри вставки, будет выполнен в процессе обработки шаблона. Причём любые операции консольного вывода приведут к тому, что выводимые данные будут вставлены в результирующий текст. Например вставка <% print "hello" %> будет заменена строкой hello.

Выражение необходимо в том случае, когда требуется получить результат. Например вставку выше можно заменить на вставку <%= "hello" %>. Вставка для выражений применяется для того, чтобы исключить необходимость использования методов print, puts и пр.

Ruby-код не прерывается между вставками. Это позволяет организовать его следующим образом (фрагмент примера выше):

```
<%= form_tag("/calc/view", :method => "get") do %>
      <%= label_tag("Value 1:") %>
      <%= text_field_tag(:v1) %>  <br/>

      <br/>

      <%= submit_tag("Calc result") %>
<% end %>
```

В данном случае вызывается метод form_tag, в блоке которого вызываются методы label_tag, text_field_tag, submit_tag. Между вставками, содержащими Ruby-код может содержаться любой текст, который будет вставлен в итоговый текст. Однако текст, который формируют методы form_tag, label_tag, text_field_tag, submit_tag, будет размещен точно в позиции содержащих их вставок. Отметим, что эти методы реализует Ruby on Rails. Более подробно см. http://guides.rubyonrails.org/form_helpers.html и http://guides.rubyonrails.org/layouts_and_rendering.html http://api.rubyonrails.org/classes/ActionView/Helpers/FormTagHelper.html

Ограничимся описанием использованных выше методов.

Метод для прописывания тэга формы. Обратите внимание на то, что метод поддерживает блоки. Последний параметр &block является альтернативным для Ruby способом передачи блока.
form_tag(url_for_options = {}, options = {}, &block)

Одна из возможных опций (options):
:method — устанавливает метод отправки формы "get" или "post".

Примеры:

```
form_tag('/posts')
# => <form action="/posts" method="post">

form_tag('/posts/1', method: :put)
# => <form action="/posts/1" method="post"> ... <input name="_method"
type="hidden" value="put" /> ...
```

```
form_tag('/upload', multipart: true)
# => <form action="/upload" method="post" enctype="multipart/form-data">

<%= form_tag('/posts') do -%>
  <div><%= submit_tag 'Save' %></div>
<% end -%>
# => <form action="/posts" method="post"><div><input type="submit" name="commit"
value="Save" /></div></form>
```

Метод формирования HTML-метки и прописывание имени в качестве атрибута:

```
label_tag(name = nil, content_or_options = nil, options = nil, &block)
```

Примеры:

```
label_tag 'name'
# => <label for="name">Name</label>

label_tag 'name', 'Your name'
# => <label for="name">Your name</label>

label_tag 'name', nil, class: 'small_label'
# => <label for="name" class="small_label">Name</label>
```

Метод формирования текстового поля.

```
text_field_tag(name, value = nil, options = {})
```

Допустимые значения Options:

- :disabled — Если установлено в true, то пользователь не сможет использовать это поле.
- :size — Видимый размер поля в символах.
- :maxlength — Максимальное количество символов, которое может ввести пользователь.
- :placeholder — Текст, которые печатается по умолчанию в поле до тех пор, пока пользователь не начнёт ввод.

Примеры:

```
text_field_tag 'name'
# => <input id="name" name="name" type="text" />

text_field_tag 'query', 'Enter your search query here'
# => <input id="query" name="query" type="text" value="Enter your search query
here" />

text_field_tag 'search', nil, placeholder: 'Enter search term...'
# => <input id="search" name="search" placeholder="Enter search term..."
type="text" />

text_field_tag 'request', nil, class: 'special_input'
# => <input class="special_input" id="request" name="request" type="text" />

text_field_tag 'address', '', size: 75
# => <input id="address" name="address" size="75" type="text" value="" />

text_field_tag 'zip', nil, maxlength: 5
```

```
# => <input id="zip" maxlength="5" name="zip" type="text" />

text_field_tag 'payment_amount', '$0.00', disabled: true
# => <input disabled="disabled" id="payment_amount" name="payment_amount"
type="text" value="$0.00" />

text_field_tag 'ip', '0.0.0.0', maxlength: 15, size: 20, class: "ip-input"
# => <input class="ip-input" id="ip" maxlength="15" name="ip" size="20"
type="text" value="0.0.0.0" />
```

Метод для создания селектора в форме круга. Для предоставления возможности выбора одного значения из группы следует сформировать несколько селекторов с одним и тем же именем.

```
radio_button_tag(name, value, checked = false, options = {})
```

Допустимые значения «Options»:

- :disabled — Если установлено в true, то пользователь не сможет использовать это поле.

Примеры:

```
radio_button_tag 'gender', 'male'
# => <input id="gender_male" name="gender" type="radio" value="male" />

radio_button_tag 'receive_updates', 'no', true
# => <input checked="checked" id="receive_updates_no" name="receive_updates"
type="radio" value="no" />

radio_button_tag 'time_slot', "3:00 p.m.", false, disabled: true
# => <input disabled="disabled" id="time_slot_300_pm" name="time_slot"
type="radio" value="3:00 p.m." />

radio_button_tag 'color', "green", true, class: "color_input"
# => <input checked="checked" class="color_input" id="color_green" name="color"
type="radio" value="green" />
```

Метод для формирования кнопки.
submit_tag(value = "Save changes", options = {})
Допустимые значения «Options»:

- :data – используется для добавления пользовательских данных.
- :disabled - Если установлено в true, то пользователь не сможет использовать это поле.
- Любые другие ключи будут интерпретироваться как стандартные HTML-опции и также будут добавлены в итоговый вывод.

Примеры:

```
submit_tag
# => <input name="commit" type="submit" value="Save changes" />

submit_tag "Edit this article"
# => <input name="commit" type="submit" value="Edit this article" />

submit_tag "Complete sale", data: { disable_with: "Please wait..." }
# => <input name="commit" data-disable-with="Please wait..." type="submit"
value="Complete sale" />
```

```
submit_tag nil, class: "form_submit"
# => <input class="form_submit" name="commit" type="submit" />

submit_tag "Edit", class: "edit_button"
# => <input class="edit_button" name="commit" type="submit" value="Edit" />
```

Функциональные тесты контроллеров

Функциональные тесты предназначены для проверки функционирования конкретных действий контроллера и позволяют проверить:

- является ли успешным или не успешным обращение к заданному действию контроллера;
  было ли выполнено перенаправление пользователя на заданную страницу в процессе выполнения действия;
- имеется ли необходимый объект для использования в представлении;
- было ли сформировано необходимое сообщение для пользователя.

Для разработанного приложения-калькулятора целесообразно проверить:

- действие Calc#input возвращает success;
- действие Calc#view при наличии всех необходимых параметров создаёт @result;
- действие Calc#view при отсутствии необходимых параметров не создаёт @result.

Файл для функционального теста уже сформирован при генерации контроллера. Поэтому необходимо наполнить его соотествующими действиями. Файл находится в директории test/controllers и называется calc_controller. После автоматической генерации он выглядит следующим образом:

```
require 'test_helper'

class CalcControllerTest < ActionController::TestCase
  test "should get input" do
    get :input
    assert_response :success
  end

  test "should get view" do
    get :view
    assert_response :success
  end

end
```

Как видим, тест уже содержит проверку факта корректного выполнения действий контроллера, которые были указаны при его генерации, по запросу типа get (в данном случае get-запрос лишь эмулируется). Для действия :input проверку результата `:success` можно считать корректной, поскольку задачей проверки состава сформированной формы данный тест не проводит. Тестирование действия `:view` в данном случае неполноценно, поскольку ни входные данных, ни наличие соответствующих сформированных объектов не проверяется. Добавим два дополнительных теста, проверяющих конкретные результаты выполнения действий:

- тест со значениями параметров v1=1, v2=10, op= '+', то есть выполнить операцию сложения 1+10 и проверить результат 11.
- тест на получение результата 'Unknown!', если входные данные не корректны.

Имена тестов задаются в виде строки, причём эта строка должна отражать реально выполняемые действия. Приведём код с дополнительными тестами.

```
 require 'test_helper'

class CalcControllerTest < ActionController::TestCase
  test "should get input" do
    get :input
    assert_response :success
  end

  test "should get view" do
    get :view
    assert_response :success
  end

  test "should get 11 for view with with 1+10" do
    get :view, {v1: 1, v2: 10, op: '+'}
    assert_equal assigns[:result], 11
  end

  test "should get Unknown! for view with incorrect params" do
    get :view
    assert_equal assigns[:result], 'Unknown!'
  end

end
```

В строке `get :view, {v1: 1, v2: 10, op: '+'}` производится вызов метода get, которому передаётся имя метода-действиия контроллера view, причём указаны в форме хэша значения параметров. Результат проверяется в утверждении на равенство `assert_equal`, причём доступ к созданным внутри действия переменным уровня экземпляра осуществляется с помощью хэша **assigns**. В первом случае переменная @result (в коде теста обращение осуществляется по имени через хэш assigns) имеет значение 11, во втором должна содержать строки 'Unknown!'.

Запуск тестов осуществляется в корневой директории приложения при помощи команды **rake test** при не запущенном веб-сервере. Если тесты выполнены корректно, будет получено сообщение о 4 выполненных тестах, 4-х утверждениях и 0 ошибок. Иначе будет получено сообщение об ошибке в конкретном тесте.

Контрольные вопросы

1. Назовите основные принципы модели Model-View-Controller.
2. Приведите основные команды rails.
3. Назовите несколько методов для формирования HTML-элементов внутри шаблонов rails.
4. Приведите примеры встроенных тестов Rails и их основное назначение.

Задание на лабораторную работу

1. Сгенерировать каркас Rails-приложения в директории, полный путь к которой содержит только символы кодировки ASCII-7bit.
2. При помощи команды “rails generate controller” сформировать контроллер для реализации логики приложения и двух действий — ввод данных, просмотр результата.
3. Дописать код сформированного контролера для расчета заданной в задании функции. Рекомендуется предварительно разработать и отладить программу вычисления функции вне Rails-приложения и размещать в контроллер уже отлаженный код.
4. Написать в файле представления (.erb) код для формирование формы ввода данных, необходимых при расчёте, а также код для форматирования результатов расчёта в виде таблицы.
5. Отладить и проверить работу приложения.
6. Заменить обращение по корневому адресу на обращение к действиям созданного контроллера.
7. Реализовать функциональный тест разработанного контроллера приложения на базе каркаса, сформированного при его создании. Проверить выполнение теста.

Отчет должен содержать:

- исходные коды контроллера, представлений и функционального теста с указанием имени файла;
- изображения страниц с формой ввода значений и вывода результатов вычислений;
- результат выполнения функционального теста.

Литература

1. Гибкая разработка веб-приложений в среде Rails. 4-е издание Сэм Руби, Дэйв Томас, Дэвид Хэнссон. Серия: Для профессионалов.- Питер: 2013.- 464 стр.
2. http://russian.railstutorial.org/chapters/beginning
3. http://russian.railstutorial.org/
4. http://v32.rusrails.ru/a-guide-to-testing-rails-applications/functional-tests-for-your-controllers
5. Sam Ruby, Dave Thomas, David Heinemeier Hansson. Agile Web Development with Rails. Third Edition. The Pragmatic Bookshelf. Raleigh, North Carolina Dallas, Texas. 2009
6. Obie Fernandez. The rails 3 way. 4-th edition. Addison-Wesley. 2010
7. http://guides.rubyonrails.org/testing.html