

Р. С. Самарев

Создание простейших веб-приложений с помощью Ruby on Rails и AJAX

*Методические указания к выполнению
практикума № 5 и лабораторной работы № 5
по дисциплинам «Языки интернет-программирования»
и «Практикум по интернет-программированию»*



Москва

ИЗДАТЕЛЬСТВО
МГТУ им. Н. Э. Баумана

2 0 1 5

УДК 681.3.06
ББК 22.18
С17

Издание доступно в электронном виде на портале *ebooks.bmstu.ru*
по адресу: <http://ebooks.bmstu.ru/catalog/255/book1270.html>

Факультет «Информатика и системы управления»
Кафедра «Компьютерные системы и сети»

*Рекомендовано Редакционно-издательским советом
МГТУ им. Н.Э. Баумана в качестве методических указаний*

Рецензент *Т.Н. Захарова*

- Самарев, Р. С.**
С17 Создание простейших веб-приложений с помощью Ruby on Rails и AJAX : методические указания к выполнению практикума № 5 и лабораторной работы № 5 по дисциплинам «Языки интернет-программирования» и «Практикум по интернет-программированию» / Р. С. Самарев. — Москва : Издательство МГТУ им. Н. Э. Баумана, 2015. — 50, [6] с. : ил.

ISBN 978-5-7038-4218-8

Представлены практикум и лабораторная работа по созданию простейших веб-приложений с помощью Ruby on Rails и AJAX.

Для студентов МГТУ им. Н.Э. Баумана, изучающих дисциплины «Практикум по интернет-программированию» и «Языки интернет-программирования».

УДК 681.3.06
ББК 22.18

ISBN 978-5-7038-4218-8

© МГТУ им. Н. Э. Баумана, 2015
© Оформление. Издательство
МГТУ им. Н. Э. Баумана, 2015

Предисловие

Методические указания содержат сведения о принципах проектирования Model-View-Controller, которые используются в библиотеке Ruby on Rails.

Рассматриваются примеры построения простого Ruby on Rails веб-приложения, а также веб-приложения, сформированного с помощью специализированного генератора ресурсов.

Отдельный раздел посвящен созданию веб-приложений с асинхронным интерфейсом (AJAX) традиционными для языка Javascript средствами, а также с использованием встроенных возможностей Ruby on Rails.

Предлагаемые для выполнения практические задания закрепляют полученные учащимися теоретические знания.

Методические указания предназначены для студентов вузов, имеющих базовые знания о языках Ruby, Javascript, HTML.

Вопросы и замечания по данной работе просьба присылать автору на адрес: samarev@acm.org.

Практикум № 5

СОЗДАНИЕ ПРОСТЕЙШИХ ВЕБ-ПРИЛОЖЕНИЙ

RUBY ON RAILS

Цель работы — углубление теоретических сведений о принципах проектирования Model-View-Controller и получение практических навыков создания веб-приложения с использованием средств Ruby on Rails, построения простейших форм и выполнения вычислений на стороне серверной части приложения.

Объем работы — 4 часа.

ПРИНЦИПЫ ПОСТРОЕНИЯ ПРИЛОЖЕНИЯ

С ИСПОЛЬЗОВАНИЕМ RUBY ON RAILS

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

В данной концепции модель ответственна за сохранение состояния приложения. В одних случаях состояния являются переходными, единственное назначение которых — обеспечить взаимодействия с пользователем, в других случаях состояния постоянные и сохраняются вне приложения, например в системе управления базами данных (СУБД).

Модель является больше чем набором данных. Она включает в себя ограничения, которые налагаются на данные. Например, не может быть предоставлена скидка на заказы менее чем 1000 руб., и модель должна содержать это ограничение. Таким образом, задача модели — обеспечить целостность данных посредством применения определенных правил (ограничений).

Представления ответственны за формирование интерфейса пользователя, основанного на данных, получаемых из модели.

Например, сайт магазина имеет список продуктов, отображаемый на экране монитора. Этот список будет получен через модель именно представлением, которое передаст его в соответствующем формате конечному пользователю. Также представление может предлагать пользователю различные способы ввода данных, но оно никогда не занимается их обработкой. Представление завершает работу, как только данные переданы пользователю.

Возможна ситуация, когда несколько представлений позволяют получить доступ к одним и тем же данным, но с разными целями. Например, на сайте интернет-магазина товары должны отображаться по-разному, в зависимости от того, кто зашел на сайт: пользователь (покупатель) для оформления заказа или администратор для редактирования товаров.

Контроллеры связывают отдельные элементы приложения. Они получают события из внешнего мира (например, команды пользователя), взаимодействуют с моделями и активируют соответствующее представление для пользователя. На рис. 1 представлена описанная схема взаимодействия, т. е. концепция MVC.

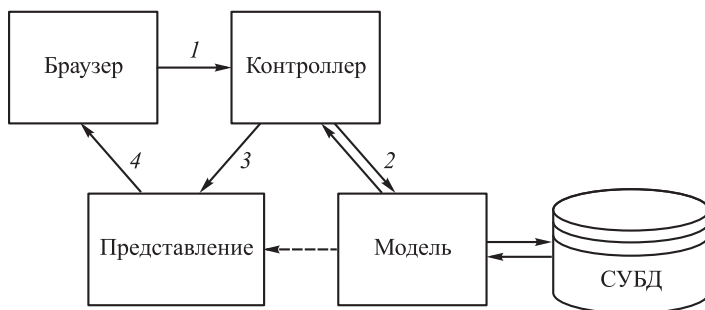


Рис. 1. Концепция MVC:

1 — браузер отправляет запрос; 2 — контроллер взаимодействует с моделью; 3 — контроллер подключает представление; 4 — средствами отображения формируется новая страница для браузера

Ruby on Rails (далее Rails) также реализует концепцию MVC. Rails задает структуру будущего приложения, а именно модели, представления и контроллеры как кубики функциональности, которые будут связаны вместе в момент запуска приложения. Отметим, что в настоящее время используется уже 4-е поколение Rails.

Примеры, приведенные в методических указаниях, 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

В результате будет создана директория с именем test_app, в которой формируется структура приложения. При этом в консоль выдаются сообщения. Далее приведены их примеры.

Создание приложения:

```
create
```

Создание шаблона краткой информации о приложении:

```
create  README.rdoc
```

Создание файла для системы сборки Rake (аналог команды **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, помощники (helper), 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
```

Создание конфигурационной части приложения, которая включает маршрутизацию, настройки запуска, локали, параметры подключения к базам данных:

```
create config/routes.rb
create config/application.rb
create config/environment.rb
create config/environments/development.rb
create config/environments/production.rb
create config/environments/test.rb
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/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/assets
```

Директория для журналов выполнения приложения:

```
create log
```

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/functional
create test/integration
create test/unit
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/stylesheets
create vendor/plugins
```

Далее по команде **run bundle install** запускаем специальный менеджер пакетов **bundle**, который проверяет состав имеющихся gem-пакетов и устанавливает непосредственно с указанного в файле **Gemfile** сайта все необходимые пакеты. Приведем 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.
```

Таким образом, после выполнения команд **rails new testapp** и **run bundle install** в минимальном виде каркас приложения создан,

а необходимые пакеты установлены. Перейдем в директорию приложения и запустим его 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
```

где **WEBrick** — отладочный веб-сервер, который встроен в Rails и предназначен только для однопользовательской работы в целях отладки.

В отличие от веб-серверов, которые служат для эксплуатации приложений, этот веб-сервер позволяет вносить изменения в код приложения и видеть изменения без его перезапуска. В приведенном сообщении также указаны версии Rails, Ruby и строка `http://0.0.0.0:3000`. Эта строка означает, что веб-сервер присоединился на фиктивный интерфейс 0.0.0.0, который готов принимать запросы, поступающие со всех имеющихся сетевых адаптеров, а значение `:3000` — ip-порт.

Откроем браузер, введем адрес `http://localhost:3000` и получим страницу, приведенную на рис. 2. Она является страницей по умолчанию `public/index.html` (для версии Rails 3.2 и более ранних версий), которая сгенерирована в момент создания приложения. В дальнейшем эта страница должна быть удалена или заменена. В версии Rails 4 она входит в состав внешних gem-модулей, поэтому единственный вариант ее замены — изменение корневого маршрута с помощью метода `root` в файле `config/routes.rb`.

После того как браузер отобразит страницу 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 осуществлено подключение к базе данных, указанной в database.yml, а изображение успешно возвращено клиенту (код 200 OK).

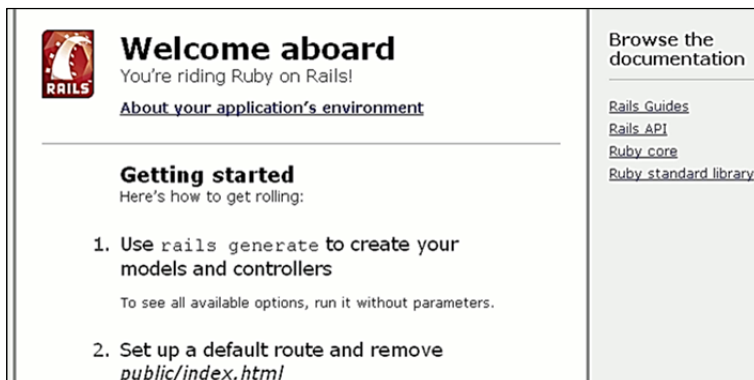


Рис. 2. Страница по умолчанию <http://localhost:3000/>

Веб-сервер остановим нажатием комбинации Ctrl-C.

Исследование Rails-приложения

В рамках знакомства с Rails используем генератор типовых контроллеров, так называемый Rails Scaffold (строительные леса).

Выполним команду для запуска Scaffold-генератора:

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

В итоге получим набор сообщений. Далее приведены их примеры.

Создание спецификации базы данных:

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

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
```

Добавление 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 контроллера:

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

Создание помощника (helper):

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

Создание теста для помощника:

```
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 запуска реального создания базы данных выполним команду

```
rake db:migrate
```

В результате будет получена база данных /db/development. sqlite3 (задана в конфигурации) по файлу миграции db/migrate/20130820160323_create_users.rb. 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
```

Здесь 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 `users`. Различие в написании этих имен заключается в принятом соглашении об именах. В момент запуска Scaffold-генератора указывается имя сущности в единственном числе, а имя контроллера и имя таблицы — во множественном числе.

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



Рис. 3. Страница <http://localhost:3000/users/>

Введем в браузер адрес <http://localhost:3000/users/>, где `users` — имя созданного контроллера, и получим страницу, приведенную на рис. 3.

Нажмем `New User`, и на экране будет отображена форма добавления нового пользователя с полями `name`, `email`, заданными при запуске Scaffold-генератора. Добавим поочередно нескольких пользователей (рис. 4 и 5).

Повторно введем в браузер адрес <http://localhost:3000/users/> и получим страницу, представленную на рис. 6.

Таким образом, после введения команды Scaffold-генератора можем убедиться, что сгенерированное приложение позволяет выполнять функции просмотра, добавления, редактирования и удаления пользователей. При этом следует обратить внимание на то, что весь необходимый код был сформирован автоматически.

Далее рассмотрим другие генераторы и на примере сгенерированного приложения разберем назначение компонентов Rails.

New user

Name

Email

[Back](#)

Рис. 4. Страница
<http://localhost:3000/users/new>

User was successfully created.

Name: Иванов А.А.

Email: ivanov@smb.ru

[Edit](#) | [Back](#)

Рис. 5. Страница
<http://localhost:3000/users/1>

Listing users

Name	Email	
Иванов А.А.	ivanov@smb.ru	Show Edit Destroy
Петров П.П.	petrov@smb.ru	Show Edit Destroy

[New User](#)

Рис. 6. Страница <http://localhost:3000/users/>

Внесем дополнения в схему взаимодействия компонентов по концепции MVC с учетом реального сгенерированного приложения Rails (рис. 7).

Принципиальных отличий на этой схеме от общей концепции MVC немного. Rails-приложение контролирует компонент с названием Rack, который является посредником в передаче запросов пользователей от веб-сервера к веб-приложению. В документации¹ приводятся следующие веб-серверы, с которыми Rack может работать:

- Mongrel;
- EventedMongrel;
- SwiftppliedMongrel;
- WEBrick;
- FCGI;
- CGI;
- SCGI;
- LiteSpeed;
- Thin.

С использованием компонента Rack построено несколько библиотек для создания веб-приложений:

¹ <http://rack.rubyforge.org/doc/>

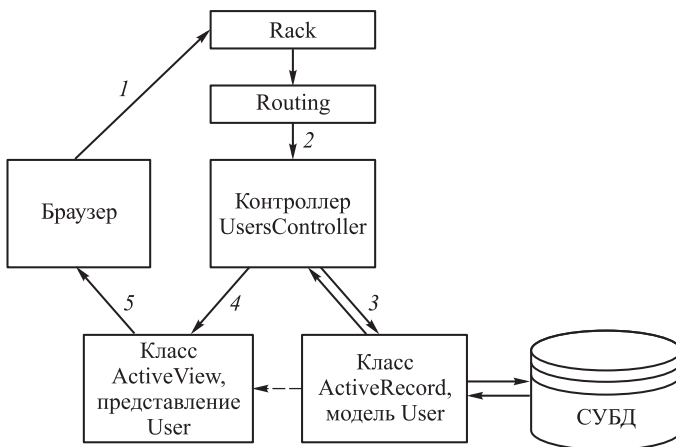


Рис. 7. Схема взаимодействия компонентов MVC на примере Rails-приложения:

1 — браузер отправляет запрос `http://localhost:3000/users/1`; 2 — маршрутизатор находит контроллер `users`; 3 — контроллер `UsersController` взаимодействует с моделью `User`; 4 — контроллер подключает представление `User`; 5 — средствами отображения формируется новая страница для браузера. Штриховой стрелкой обозначена логическая передача данных. Взаимодействие с СУБД является опциональным

- Camping;
- Coset;
- Halcyon;
- Mack;
- Maveric;
- Merb;
- Racktools::SimpleApplication;
- Ramaze;
- Ruby on Rails;
- Rum;
- Sinatra;
- Sin;
- Vintage;
- Waves;
- Wee;
- ...

Большая их часть имеет достаточно ограниченное применение. В контексте Rails-приложения необходимо знать, что запрос от Rack передается 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 завершения работы Scaffold-генератора в него была добавлена строка:

```
resources :users
```

где `resources` — создание стандартного набора REST-маршрутов, который создаст 4 именованных маршрута (в концепции CRUD — Create Read Update Destroy) и 7 действий: `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 адрес `http://localhost:3000/users/1` — неявно на маршрут `UserController#show`, но с подразумеваемым параметром `:id`.

Маршруты указывают на конкретные контроллеры, которые должны обрабатывать запросы. В данном 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 `UserController` является потомком Rails-класса `ApplicationController`.

Обратите внимание на то, что в комментариях перед методами сгенерированного класса `UserController` указаны маршруты, по которым предполагается вызов этих методов. Рассмотрим 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

```

Здесь первая строка — маршрут получения списка пользователей через класс `User`; этот класс является моделью пользователей; по имени вызываемого метода `.all` можно предположить, что будет получен список всех пользователей. Для дальнейшего использования этого списка введена переменная уровня экземпляра с именем `@users`.

Файл модели `app/models/user.rb` содержит следующий код:

```

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

```

Из этого кода следует, 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.

Возвратимся к контроллеру и методу `index`, где метод `respond_to` предназначен для предоставления возможности получения ответа от

веб-сервера в различных форматах. Однако явно метод может и не использоваться, если формат получения единственный. Здесь же предусмотрено два формата: `html` и `json`. Для проверки этого введем в браузере адрес: `http://localhost:3000/users.json`. В результате 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\u0418\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\u0410.", "updated_at":"2012-08-17T17:36:57Z"}]
```

В случае `format.html` предполагается использование представления `index.html.erb`. Обратите внимание на то, что в приведенном ранее коде метода `index` запись `"# index.html.erb"` — всего лишь комментарий, поясняющий, откуда будет использовано представление.

В процессе выполнения Scaffold-генератора представления были сгенерированы для каждого маршрута, однако сейчас необходим лишь файл `app/views/users/index.html.erb`. Этот файл содержит следующий 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 присутствуют HTML-разметка, специальная разметка в форме `<% %>` и разметка `<%= ... %>`, которая содержит вызовы Ruby-кода и специальных методов типа `link_to`. Обратите внимание на то, что для вывода списка пользователей формируется HTML-таблица. Однако число пользователей не известно, поэтому в коде использован цикл генерации строк:

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

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

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

Следует также иметь в виду, что этот шаблон не содержит всей необходимой HTML-разметки. Шаблон отображения страниц приложения был сгенерирован на самом первом этапе генерации Rails-приложения при выполнении Scaffold-генератора 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>
```

Здесь содержится внешнее обрамление, а вставка конкретных шаблонов представлений выполняется вместо строки `<%= yield %>`.

Завершим исследование Rails-приложения анализом сгенерированных тестов. Отметим, что по умолчанию в Rails генерируются

тесты уже устаревшей системы тестирования Unit. Для использования системы тестирования rspec необходимо осуществить дополнительные действия. В данных указаниях рассмотрим Unit-тесты.

Для запуска тестов достаточно выполнить в директории Rails-приложения команду **rake**. Файл Rakefile содержит цель test в качестве цели по умолчанию. Дополнительно укажем трассировку вызовов и запустим rake – trace. Приведем пример полученного сообщения:

```
** 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

```

Из этого сообщения следует, что было запущено 7 тестов, принято 10 утверждений, ошибок² не обнаружено.

Большая часть тестов ничего не содержит. Однако имеется сгенерированный функциональный тест в файле `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
end

```

² Если при прохождении теста обнаружена ошибка доступа к базе данных, проверьте значение переменной окружения `RAILS_ENV` (значение которой должно быть `development` или быть пусто) или сбросьте ее 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 следует, что предполагается эмуляция GET-, POST-, PUT-, DELETE-запросов протокола HTTP посредством вызова соответствующих методов, которым передаются URL, что позволяет проверить поведение всех маршрутов и методов сгенерированного контроллера.

Запуск только функциональных тестов осуществляется командой

```
rake test:functionals
```

На этом в исследовании сгенерированного приложения остановимся и перейдем к примеру создания собственного приложения. Более подробно ознакомиться с настройкой веб-приложения можно в приложении 1.

ПРИМЕР ВЕБ-ПРИЛОЖЕНИЯ С ФОРМОЙ ДЛЯ ВВОДА ДАННЫХ

Создание приложения

Разработаем приложение-калькулятор, задача которого — принять введенные значения и выдать результат. Сначала откроем консоль и выполним ряд следующих действий:

- 1) запустим команду **rails new calc** и войдем в созданную директорию calc;
- 2) запустим команду **rails generate controller Calc input view**;
- 3) откроем файл `app/views/input.html.erb` в текстовом редакторе и добавим следующий код:

```
<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) откроем файл `app/views/view.html.erb` в текстовом редакторе и добавим код

```
<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) откроем файл `app/controllers/calc_controller.rb` в текстовом редакторе и добавим код

```

class CalcController < ApplicationController
  def input
    end

  def view
    v1, v2 = 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
end

```

6) запустим веб-приложение командой

rails server -e development

или используем более короткий ее вариант:

rails s

На рис. 8 представлена страница с формой для ввода значений, которая должна стать доступной браузеру по адресу <http://localhost:3000/calc/input>, а на рис. 9 — страница с результатом вычисления.

Рис. 8. Страница <http://localhost:3000/calc/input>

Рис. 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. Этот код выполняется в процессе подстановки шаблона.

Шаблон представляет собой текст, имеющий вставки специального формата для кода и выражений.

Вставки для кода:

```
<% ruby code %>
```

Вставки для выражений:

```
<%= ruby expression %>
```

Код, расположенный внутри вставки, выполняется в процессе обработки шаблона. Причем операции консольного вывода могут привести 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 `<% print "hello" %>` 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 (см. фрагмент примера в п. 3 создания приложения):

```
<%= 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, включающими в себя Ruby-код, может содержаться любой текст, который будет вставлен в итоговый текст. Однако текст, формируемый методами `form_tag`, `label_tag`, `text_field_tag`,

submit_tag, будет размещен точно в позиции включающих их вставки. Отметим, что эти методы реализует Ruby on Rails (более подробно об этом см. в приложении 2).

Функциональные тесты контроллеров

Функциональные тесты предназначены для контроля функционирования конкретных действий контроллера. Они позволяют проверить:

- является ли успешным или неуспешным 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 следующие результаты:

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

Файл для функционального теста формируется автоматически при генерации контроллера. Необходимо только наполнить его соответствующими действиями. Файл находится в директории test/controllers и называется calc_controller. После автоматической генерации он выглядит таким 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 содержит проверку факта корректного выполнения действий контроллера указанных при его генерации, по запросу типа `get` (в данном случае HTTP-get-запрос лишь эмулируется). Для действия `:input` проверку результата `:success` можно считать корректной, поскольку задачей проверки состава созданной формы данный тест не осуществляет. Тестирование действия `:view` в данном коде неполноценно, поскольку ни входные данные, ни наличие соответствующих сформированных объектов не проверяются. Добавим два дополнительных теста, проверяющих конкретные результаты выполнения действий:

- тест со значениями параметров `v1=1, v2=10, op= '+'`, проверяющий, что операция сложения `1+10` возвращает результат `11`;
- тест на получение результата `'Unknown!'`, если входные данные не корректны.

Имена тестов задаются в виде строки, причем эта строка должна отражать реально выполняемые действия. Приведем 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 incorrect params" do
    get :view
    assert_equal assigns[:result], 'Unknown!'
  end
end
```

В строке `get :view, {v1: 1, v2: 10, op: '+'}` выполняется вызов метода `get`, которому передается имя метода (действия в терминологии Rails) контроллера `view`, причем в форме хэша указаны значения параметров. Результат проверяется в утверждении

равенства `assert equal`, при этом доступ к созданным внутри действия переменным уровня экземпляра осуществляется с помощью хэша `assigns`. В случае `:view` с дополнительными параметрами переменная `@result` (в коде теста обращение выполняется по имени через хэш `assigns`) имеет значение 11, а без дополнительных параметров она должна содержать строки 'Unknown!'.

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

ПОРЯДОК ВЫПОЛНЕНИЯ ЗАДАНИЯ

1. Сгенерируйте каркас Rails-приложения в директории, полный путь к которой содержит только символы кодировки ASCII-7bit.

2. С помощью команды **rails generate controller** сформируйте контроллер для реализации логики приложения и двух действий: ввод данных, просмотр результата.

3. Допишите код сформированного контроллера для расчета функции, заданной индивидуально. Предварительно разработайте и отладьте программу вычисления функции вне Rails-приложения и разместите в контроллере уже отлаженный код.

4. Напишите в файле представления (`.erb`) код для генерации формы ввода данных, необходимых при расчете, а также код для форматирования результатов расчета в виде таблицы с использованием соответствующих элементов разметки.

5. Отладьте и проверьте работу приложения.

6. Замените обращение по корневому адресу на обращение к действиям созданного контроллера.

7. Реализуйте функциональный тест разработанного контроллера приложения на базе каркаса, сформированного при его создании. Проверьте выполнение теста.

СОДЕРЖАНИЕ ОТЧЕТА

Отчет должен содержать:

1) исходные коды контроллера, представлений и функционального теста с указанием имени файла;

- 2) изображения страниц с формой ввода значений и вывода результатов вычислений;
- 3) результат выполнения функционального теста.

КОНТРОЛЬНЫЕ ВОПРОСЫ

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

НАСТРОЙКА RAILS-ПРИЛОЖЕНИЯ

Bundler. Bundler (узел, связка, пачка) не является компонентом Rails, однако рекомендован для использования. 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 выполнены разработка и тестирование приложения. Именно решением этих задач и занимается средство под названием Bundler, запуск которого осуществляется командой `bundle`.

Конфигурацию пакетов задает файл `Gemfile`. Этот файл генерируется автоматически при создании базовой структуры Rails-приложения и может иметь 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
```

Отметим, что файл использует синтаксис языка Ruby. Однако для простоты назовем методы типа `source`, `gem`, `group` параметрами, поскольку так они именуются в конфигурационных файлах.

Параметр `source` указывает на адрес репозитория пакетов обычно `'http://rubygems.org'`, в некоторых случаях — на адрес

'https://rubygems.org'. Считается, что использование протокола https может повысить уровень достоверности скачиваемых пакетов, поскольку сертификаты шифрования, используемые в этом протоколе, должны быть официально подписаны, однако для операционной системы MS Windows это может создать проблему отсутствия сертификатов и невозможность подключения вообще. В операционных системах семейства Linux полноценно поддерживаются оба протокола http, https.

Параметр `gem` указывает на имя `gem`-пакета, который необходим приложению. Примеры подключения `gem`-пакетов с некоторыми опциями приведены ниже.

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 запуска Rails-приложения: `test`, `development`, `production`. Поскольку пользователи приложения не будут заниматься разработкой, то им не нужны пакеты, которые относятся к конфигурации `test` или `development`. Приведем пример разделения на группы:

```
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 требуется установить только в указанной конфигурации, следует применить параметр `--without`, например:

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

После выполнения команды **bundle install** в директории приложения создается файл `Gemfile.lock`, который содержит список конкретных пакетов с указаниями номеров версий и всеми зависимостями, которые имелись 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 директории `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 файлы и параметры, которые содержатся в них (см. гл. *Rails Environments and Configuration* [fernandez] [5]).

В процессе запуска приложения подключаются три файла:

1) `config/boot.rb`, устанавливающий путь к `Gemfile` и запускающий `bundle/setup`;

2) `config/application.rb`, который загружает все `gem`-пакеты rails, а также пакеты для текущей конфигурации приложения, установленной в переменной `Rail.env`;

3) `config/environment.rb`, который 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, бóльшая часть параметров удовлетворяет значениям по умолчанию. В будущем может понадобиться изменение языка по умолчанию в параметре `config.i18n.default_locale`.

КРАТКОЕ ОПИСАНИЕ МЕТОДОВ — ГЕНЕРАТОРОВ РАЗМЕТКИ ФОРМЫ HTML

Метод `form_tag(url_for options = {}, options = {}, &block)` служит для прописывания тега формы. Обратите внимание на то, что метод поддерживает блоки. Параметр `&block` является альтернативным для Ruby способом передачи блока.

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

Примеры вызова метода:

```
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 `label_tag(name = nil, content_or_options = nil, options = nil, &block)` предназначен для формирования HTML-метки и прописывания имени в качестве атрибута.

Примеры вызова метода:

```
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 `text_field_tag(name, value = nil, options = {})` служит для формирования текстового поля.

Допустимые значения options:

- `:disabled` — если оно 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, который печатается по умолчанию в поле до тех пор, пока пользователь не начнет ввод.

Примеры вызова метода:

```
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 `radio_button_tag(name, value, checked = false, options = {})` используется для создания селектора в форме круга. Чтобы предоставить возможность выбора одного значения из группы,

следует сформировать несколько селекторов с одним и тем же именем.

Допустимым значением options является :disabled, т.е. если оно установлено в true, то пользователь не сможет использовать это поле.

Примеры вызова метода:

```
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 submit_tag(value = "Save changes", options = {}) применяется для формирования кнопки.

Допустимые значения options:

- :data — используется для добавления пользовательских данных;
- :disabled — если оно 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 вызова метода:

```
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" />
```

Дополнительно рекомендуется ознакомиться со следующими источниками:

- http://guides.rubyonrails.org/form_helpers.html;
- http://guides.rubyonrails.org/layouts_and_rendering.html;
- <http://api.rubyonrails.org/classes/ActionView/Helpers/FormTagHelper.html>

Лабораторная работа № 5

СОЗДАНИЕ ПРОСТЕЙШИХ ВЕБ-ПРИЛОЖЕНИЙ

RUBY ON RAILS. AJAX

Цель работы — углубление теоретических сведений о принципах работы асинхронного веб-интерфейса и получение практических навыков создания веб-приложения с использованием средств Ruby on Rails и технологии AJAX.

Объем работы — 2 часа.

АСИНХРОННАЯ ОБРАБОТКА

AJAX (Asynchronous JavaScript + XML), или асинхронный JavaScript и XML представляет собой набор технологий, предназначенных для решения задачи фоновой доставки данных на страницу пользователя. Традиционный подход к организации взаимодействия клиента и сервера заключается в том, что при каждом сеансе взаимодействия браузера и сервера сервер возвращает новую страницу, которая полностью заменяет предыдущую загруженную браузером страницу. Это создает ряд сложностей, таких как невозможность обновления информации на странице без визуально заметной перерисовки, а также увеличивает время реакции элементов интерфейса на действия пользователя.

После появления языка программирования JavaScript, позволяющего описать логику поведения браузера и управлять DOM-элементами HTML-страницы, и механизма описания данных в формате XML стало возможным создать технологию, позволяющую в фоновом режиме обеспечить доставку данных по запросу браузера. Несмотря на то, что в аббревиатуре AJAX присутствует XML, сейчас этот язык разметки чаще всего заменяют языком JSON, который является более компактным для передачи данных по сети, сохраняя иерархическую структуру и возможность восприятия этой структуры человеком.

ДОБАВЛЕНИЕ АСИНХРОННОЙ ОБРАБОТКИ В ПРИЛОЖЕНИЕ

Пример исходного приложения

В качестве исходного используем приложение, разработанное при выполнении практикума № 5, задачей которого является вычисление результата арифметической операции над введенными значениями. Напомним процесс его создания. Необходимо открыть консоль и выполнить следующие действия:

1) запустить команду **rails new calc** и войти в созданную директорию **calc**;

2) запустить команду **rails generate controller Calc input view**;

3) открыть файл **app/views/input.html.erb** в текстовом редакторе и ввести следующий код:

```
<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** в текстовом редакторе и ввести следующий код:

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

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

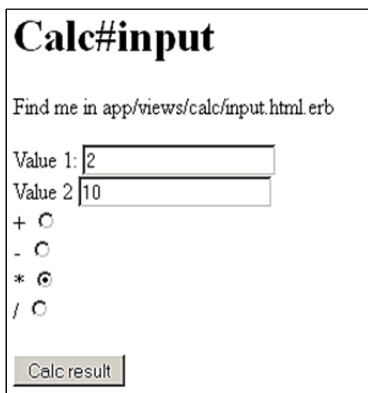
<%= link_to "Repeat calculation", :calc_input %>
```


5) открыть файл `app/controllers/calc_controller.rb` в текстовом редакторе и ввести следующий код:

```
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
```

Запускаем веб-сервер командой **rails server** и проверяем результат. На рис. 10 представлена страница, которую отобразит браузер при подключении по адресу `http://localhost:3000/calc/input`, а на рис. 11 — результат вычисления операции.



Calc#input

Find me in `app/views/calc/input.html.erb`

Value 1:

Value 2:

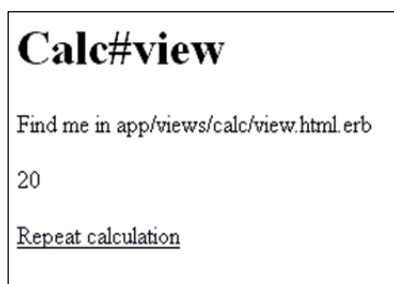
+ ☐

- ☐

* ☒

/ ☐

Рис. 10. Страница `http://localhost:3000/calc/input`



Calc#view

Find me in `app/views/calc/view.html.erb`

20

[Repeat calculation](#)

Рис. 11. Страница `http://localhost:3000/calc/view` после нажатия **Calc result**

После того как приложение стало работоспособным, изменим его так, чтобы результат выполнения отображался без перерисовки страницы в целом.

Рассмотрим два варианта преобразования приложения к асинхронному: 1) традиционными средствами JavaScript; 2) средствами, предусмотренными в Ruby on Rails 4.

Асинхронная обработка средствами JavaScript

Ruby on Rails не ограничивает возможности реализации веб-приложения, даже если способ его реализации не является рекомендуемым. Поэтому допустимой считается реализация интерфейса пользователя посредством написания кода непосредственно на языках HTML и Javascript. При этом также возможны варианты использования AJAX.

Первым шагом в реализации асинхронного приложения является доработка кода действия контроллера с тем, чтобы обеспечить выдачу результата в формате JSON. Используем метод `respond_to` и добавим поддержку этого формата. Поскольку калькулятор возвращает значение `@result`, имеющее тип число или строке, сформируем объект, для которого будет проводиться сериализация поля типа и поля значения. Для этого используем хэш: `{type: @result.class, value: @result}`. В противном случае формируемый ответ будет состоять только из числа или строки (в настройках приложения можно включить добавление корневого элемента в JSON).

Приведем доработанный код:

```
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

    respond_to do |format|
      format.html
```

```

    format.json do
      render json:
        {type: @result.class.to_s, value: @result}
    end
  end
end
end
end

```

В рассматриваемом примере методу `respond_to` предписано обрабатывать форматы `html` и `json`. Причем для формата `html` будет использовано представление с шаблоном `view.html.erb`, а для формата `json` будет явно вызван метод `render`. Обратите внимание на то, что в соответствии с синтаксисом Ruby записи `format.html` и `format.json` являются вызовом методов `#html` и `#json` объекта `format`. Если у метода отсутствует блок, значит, будет запущен механизм поиска файла представления с именем, которое соответствует имени действия, и расширением, соответствующим имени вызванного метода объекта `format`. Если же блок присутствует, будет вызван код из блока.

После добавления контроллера следует запустить приложение и проверить результат в браузере. Необходимо сформировать URL, позволяющий запустить запрос на получение результата в JSON. Для данного приложения такой URL будет иметь вид `http://localhost:3000/calc/view.json?v1=123&v2=234&op=*`

Если все выполнено правильно, то на экране получим JSON в следующем виде:

```
{ "type": "Fixnum", "value": 28782 }
```

Далее необходимо сформировать страницу, содержащую форму для ввода, и асинхронный обработчик. Форму для ввода можем легко получить, используя уже готовую форму в шаблоне `erb`. Для этого необходимо открыть в браузере страницу `http://localhost:3000/calc/input` и скопировать из кода страницы разметку, соответствующую форме. Обратите внимание на то, что статический код `html`-страниц для Rails является возможным, но не рекомендуемым, поэтому этот пример следует рассматривать лишь как учебный.

Код страницы `index.html`, которую необходимо разместить в директории `public` веб-приложения, выглядит следующим образом:

```

<!DOCTYPE html>
<html>

```

```

<head>
  <meta http-equiv="Content-Type" content="text/html;
charset=utf-8" />
  <title>AJAX calc test</title>
  <script>
    calc_url= location.protocol + "://" + location.host +
                                     "/calc/view.json"

    function send_query(){
      var frm = document.forms.calc_form;

      var param_str = "";
      var radios = frm.op;
      for (var i = 0; i < radios.length; i++) {
        if (radios[i].checked) {
          param_str = "?" +
            "v1=" + frm.v1.value +
            "&v2=" + frm.v2.value +
            "&op=" +
            encodeURIComponent(frm.op[i].value)

          break;
        }
      }
      if (param_str == "") return false;

      var my_JSON_object = {};
      var http_request = new XMLHttpRequest();
      http_request.open("GET", calc_url+param_str, true);
      http_request.onreadystatechange = function () {
        var done = 4, ok = 200;
        if (http_request.readyState == done &&
            http_request.status == ok) {
          my_JSON_object =
            JSON.parse(http_request.responseText);
          show_result(my_JSON_object);
        }
      };
      http_request.send(null);
      return false;
    }

    function show_result(data){
      var result = document.getElementById("result");
      result.innerHTML = "<hr/>Result is: " + data.value +
        "<hr/><p>" + Date() + "</p>";
    }
  </script>

```

```

</head>
<body>
  <form name="calc_form" method="get">
    <label for="Value_1:">Value 1:</label>
    <input id="v1" name="v1" type="text" /> <br/>
    <label for="Value_2">Value 2</label>
    <input id="v2" name="v2" type="text" /> <br/>
    <label for=" " ">+</label>
    <input id="op_" name="op" type="radio"
      value="+" /><br/>
    <label for=" " ">-</label>
    <input id="op_" name="op" type="radio"
      value="-" /><br/>
    <label for=" " ">*</label>
    <input id="op_" name="op" type="radio"
      value="*" /><br/>
    <label for=" " ">/</label>
    <input id="op_" name="op" type="radio"
      value="/" /><br/>
    <br/>

    <input name="commit" type="submit"
      value="Calc result" onclick="return send_query();" />
  </form>

  <div id="result"></div>
</body>
</html>

```

Необходимо обратить внимание на следующие моменты. Кнопка формы вызывает зарегистрированный обработчик

```
onclick="return send_query();"

```

То есть при нажатии этой кнопки форма не отправляет данные автоматически, а вызывает метод `send_query()`. Этот метод предназначен для формирования запроса и описан в основной части JavaScript-кода. Для работы приложения необходимо сформировать URL, содержащий параметры для вычисления. Эти параметры должны быть получены из формы. Поскольку форма имеет имя, доступ к ней возможен через одноименное свойство. Текстовые поля доступны также по имени. Кнопочный селектор необходимо считывать в цикле.

Итак, строка, содержащая URL, состоит из следующих компонентов:

```
location.protocol + "//" + location.host +
    "/calc/view.json" +
    "?" +
    "v1=" + frm.v1.value +
    "&v2=" + frm.v2.value +
    "&op=" +
    encodeURIComponent(frm.op[i].value)
```

Для отправки запроса создается объект `XMLHttpRequest`. Следует обратить внимание, что реальная отправка данных на сервер происходит лишь в момент вызова `http_request.send(null)`; Все операции до этого момента являются подготовительными. Строка, содержащая

```
http_request.onreadystatechange = function () {...}
```

регистрирует обработчик ответа, причем этот код будет вызван только после получения данных или при выявлении ошибки передачи данных.

Метод `send_query()` возвращает `false` с тем, чтобы форма не перезагружалась браузером. В противном случае браузер перезагрузит страницу полностью, а результаты выполнения асинхронного запроса не будут отображены.

Учитывая то, что данные передаются в формате JSON, необходимо после получения преобразовать их в объект JavaScript. Для этого используем метод

```
JSON.parse(http_request.responseText)
```

передавая ему текст ответа сервера. Дальнейшая работа с объектом осуществляется так же, как и с любым другим объектом JavaScript. Созданный метод `show_result(data)` получает этот объект. Поскольку переданные данные имеют формат

```
{"type": "Fixnum", "value": 28782}
```

значение будет находиться в свойстве объекта `value` и может быть использовано непосредственно для вывода в предварительно подготовленное в разметке поле `div`.

Далее следует запустить приложение и проверить его работу через разработанную страницу.

Недостаток этого способа организации асинхронного взаимодействия состоит в том, что Rails хотя и позволяет иметь страницы, но предполагает отсутствие статических страниц. Частично

недостаток можно скомпенсировать. Для этого необходимо поместить указанный html-код в шаблон `view/calc/index.html.erb`, а в коде контроллера запретить подставлять внешнее оформление следующим образом:

```
class CalcController < ApplicationController
  def input
    render layout: false
  end
  ...
end
```

Кроме того, необходимо прописать корневой маршрут `root 'calc#input'` для доступа по умолчанию в файле `config/routes.rb`

С этого момента приложение может быть запущено, а его работа не будет отличаться от первого варианта организации AJAX-взаимодействия.

Асинхронная обработка с помощью Ruby on Rails

Рассмотренные выше варианты являются достаточно простыми по реализации, но весьма громоздкими и нетехнологичными. Rails имеет средства для автоматизации формирования AJAX-запросов с помощью так называемых AJAX-helpers (AJAX-помощники) и встроенной поддержки библиотеки jQuery. Кроме того, существует возможность написания скриптов на языке Coffee-Script, код которого транслируется в JavaScript на этапе запуска приложения. Для простоты рассмотрим кодирование на языке JavaScript.

Сначала изменим контроллер `Calc` по аналогии с предыдущим вариантом. То есть добавим метод `respond_to` для того, чтобы обеспечить обработку запросов к json:

```
class CalcController < ApplicationController
  ...
  def view
    ...
    respond_to do |format|
      format.html
      format.json do
        render json:
          {type: @result.class.to_s, value: @result}
      end
    end
  end
end
```

```
end  
end
```

Далее изменим генератор формы в шаблоне `app/views/calculations.html.erb`, т. е. заменим строку

```
<%= form_tag("/calculations/view", :method => "get") do %>
```

на строку

```
<%= form_tag("/calculations/view.json", :method => "get",  
            remote: true, id: 'calc_form') do %>
```

Обратите внимание на то, что ресурс, к которому обращается форма, теперь называется `view.json`. Для того чтобы форма стала асинхронной, следует добавить свойство `:remote => true`. Необходимые изменения будут выполнены генератором HTML автоматически. Код, который формирует асинхронный запрос, также будет добавлен автоматически. Эта технология называется Unobtrusive JavaScript.

На эту же форму следует добавить поле для вывода значений

```
<div id="result"></div>
```

Последним шагом является добавление обработчика JavaScript. Файл `app/assets/javascripts/calculations.js.coffee` создается в момент генерации контроллера, но он не потребуется, поскольку будем писать на языке JavaScript. Удаляем этот файл и создаем там же файл `calculations.js`.

Этот файл будет содержать следующий код (используется библиотека jQuery, поэтому запись весьма компактна):

```
function show_result(data){  
  var result = document.getElementById("result");  
  result.innerHTML = "<hr/>Result is: " + data.value +  
  "<hr/><p>" + Date() + "</p>";  
}  
$(document).ready(function(){  
  $("#calc_form").bind("ajax:success",  
                        function(xhr, data, status) {  
    // data is already an object  
    show_result(data)  
  })  
})
```

Проанализируем этот код. Метод с именем `$()` является «фирменным» методом библиотеки jQuery. Этот метод определяет в зависимости от входных параметров те действия, которые необходимо совершать. Вызов `$(document).ready(function(){...})` регистри-

рует функцию обработки, которая вызывается в тот момент, когда документ будет полностью загружен и обработан браузером. Вызов метода `$("#calc_form")` возвращает элемент HTML-разметки, имеющий идентификатор `calc_form` (форма обращения соответствует селектору CSS). Вызов метода `bind("ajax:success")` для этого элемента связывает с ним обработчик приходящих от сервера данных. Элементом с идентификатором `calc_form` является форма. По приходу данных вызывается метод `show_result(data)`, который обеспечивает вывод результата. Обратите внимание на то, что здесь данные уже представлены в виде объекта, поэтому явное преобразование из JSON не требуется.

После выполнения всех указанных действий приложение может быть запущено и проверено.

ТЕСТИРОВАНИЕ С ПОМОЩЬЮ SELENIUM IDE

Пакет Selenium IDE представляет собой модуль для браузера Firefox, позволяющий записать последовательность действий пользователя в веб-приложении. Записанная последовательность действий может быть воспроизведена тут же в браузере Firefox или экспортирована во внешний тест на языке Ruby, использующий библиотеку `selenium-webdriver`.

Модуль для браузера Firefox можно получить на странице разработчиков Selenium <http://www.seleniumhq.org/projects/ide/>. Для запуска тестов в автономном режиме необходимо использовать модуль для Ruby, который называется `selenium-webdriver` и устанавливается командой **`gem install selenium-webdriver`**. Следует обратить внимание на то, что тест, написанный на языке Ruby, может обеспечить проверку значений, которые хранят элементы разметки HTML, позволяя не только выполнять последовательный ввод данных и нажатия кнопок, но и контролировать возвращаемые значения.

На рис. 12 приведен пример записи теста в Selenium IDE.

ПОРЯДОК ВЫПОЛНЕНИЯ ЗАДАНИЯ

1. Используйте веб-приложение, разработанное при выполнении практикума № 5 (расчетная задача, выполненная по индивидуальному заданию, которая требует ввод параметров, а результатом ввода является таблица).

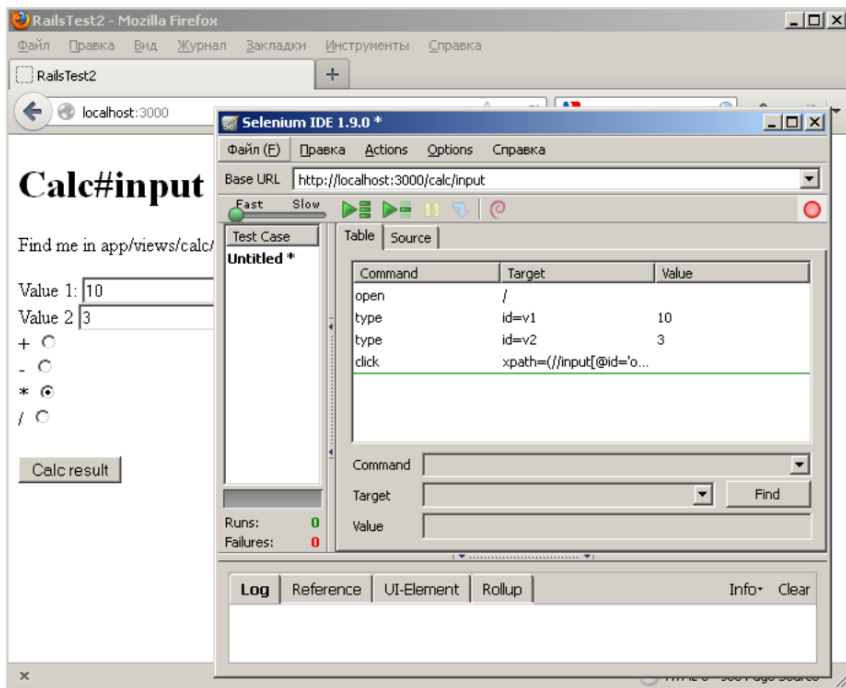


Рис. 12. Пример тестирования приложения с помощью Selenium IDE

2. Выберите способ добавления асинхронного запроса и реализуйте.
3. Отладьте веб-приложение.
4. С помощью Selenium IDE запишите тест для проверки при нескольких значениях выходных параметров.
5. Проверьте выполнение теста.

СОДЕРЖАНИЕ ОТЧЕТА

Отчет должен содержать:

- 1) исходные коды контроллера, представлений и функционального теста с указанием имени файла;
- 2) изображения страниц с формой ввода значений входных параметров и вывода результатов вычислений;
- 3) код тестов, подготовленных с помощью средств Selenium.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое AJAX?
2. Какие способы добавления асинхронного поведения предусмотрены в Ruby on Rails?
3. Как добавить обработку запросов формата json в контроллере приложения?
4. Как проконтролировать содержимое пересылаемого сервером json-пакета?
5. Что такое Selenium IDE?

Литература

Руби С., Томас Д., Хэнссон Д. Гибкая разработка веб-приложений в среде Rails. СПб.: Питер, 2013.

Hartl M. Ruby on Rails Tutorial. Learn Web Development with Rails [Электронный ресурс]. URL: <https://www.railstutorial.org/book> (дата обращения: 01.10.2014).

Функциональные тесты для ваших контроллеров [Электронный ресурс]. URL: <http://v32.rusrails.ru/a-guide-to-testing-rails-applications/functional-tests-for-your-controllers> (дата обращения: 01.10.2014).

Ruby S., Thomas D., Hansson D.H. Agile Web Development with Rails 4. The Pragmatic Bookshelf. Raleigh, North Carolina Dallas, Texas. 2013.

Fernandez O. The rails 4 way. 3-th edition. Addison-Wesley. 2014.

A Guide to Testing Rails Applications [Электронный ресурс]. URL: <http://guides.rubyonrails.org/testing.html> (дата обращения: 01.10.2014).

Working with JavaScript in Rails [Электронный ресурс]. URL: http://edgeguides.rubyonrails.org/working_with_javascript_in_rails.html (дата обращения: 01.10.2014).

Содержание

Предисловие	3
Практикум № 5. Создание простейших веб-приложений	
Ruby on Rails	4
Принципы построения приложения с использованием Ruby on Rails...	4
Пример веб-приложения с формой для ввода данных	23
Порядок выполнения задания.....	28
Содержание отчета	28
Контрольные вопросы.....	29
Приложение 1. Настройка Rails-приложения	30
Приложение 2. Краткое описание методов — генераторов разметки формы HTML	35
Лабораторная работа № 5. Создание простейших веб-приложений	
Ruby on Rails. AJAX	39
Асинхронная обработка	39
Добавление асинхронной обработки в приложение	40
Тестирование с помощью Selenium Ide	49
Порядок выполнения задания.....	49
Содержание отчета	50
Контрольные вопросы	51
Литература	52

Учебное издание

Самарев Роман Станиславович

**Создание простейших веб-приложений
с помощью Ruby on Rails и AJAX**

Редактор *О.М. Королева*

Художник *А.С. Ключева*

Корректор *Н.В. Савельева*

Компьютерная верстка *С.А. Серебряковой*

В оформлении использованы шрифты
Студии Артемия Лебедева.

Оригинал-макет подготовлен
в Издательстве МГТУ им. Н.Э. Баумана.

Подписано в печать 16.07.2015. Формат 60×90/16.
Усл. печ. л. 3,75. Тираж 50 экз. Изд. № 215-2015. Заказ .

Издательство МГТУ им. Н.Э. Баумана.
105005, Москва, 2-я Бауманская ул., д. 5, стр. 1.
press@bmstu.ru
www.baumanpress.ru

Отпечатано в типографии МГТУ им. Н.Э. Баумана.
105005, Москва, 2-я Бауманская ул., д. 5, стр. 1.
baumanprint@gmail.com

ДЛЯ ЗАМЕТОК

ДЛЯ ЗАМЕТОК