

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

федеральное государственное автономное образовательное  
учреждение  
высшего образования

**«Национальный исследовательский ядерный университет  
«МИФИ»**



**Факультет «КиБ»**

**Кафедра № 12 «Компьютерные системы и технологии»**

**ОТЧЕТ**

по учебной (научно-исследовательской) практике на тему: «Создание  
простейшего интернет-магазина на основе веб-технологий среды Ruby on  
Rails»

Группа K04-361

Студент \_\_\_\_\_ Пономарёв Евгений Александрович

Руководитель \_\_\_\_\_ Роганов Евгений Александрович

Оценка \_\_\_\_\_

**Москва 2016**

**СОДЕРЖАНИЕ**

<b>ВВЕДЕНИЕ.....</b>	<b>3</b>
<b>СОЗДАНИЕ ПРИЛОЖЕНИЯ.....</b>	<b>5</b>
<b>УСТАНОВКА НЕОБХОДИМЫХ КОМПОНЕНТОВ, НАСТРОЙКА ГЕМОВ, СОЗДАНИЕ БАЗЫ ДАННЫХ...</b>	<b>5</b>
<b>ЗАПУСК АВТОНОМНОГО СЕРВЕРА.....</b>	<b>6</b>
<b>СОЗДАНИЕ ТАБЛИЦЫ ТОВАРОВ, И ИСПОЛЬЗОВАНИЕ ГЕНЕРАТОРА     ВРЕМЕННОЙ ПЛАТФОРМЫ (SCAFFOLD) ДЛЯ УЧЕТА ТОВАРОВ В ЭТОЙ     ТАБЛИЦЕ.....</b>	<b>7</b>
<b>ПРОВЕРКА ПРИЕМЛЕМОСТИ ДАННЫХ. БЛОЧНОЕ ТЕСТИРОВАНИЕ МОДЕЛЕЙ.....</b>	<b>10</b>
<b>ДОБАВЛЕНИЕ ВОЗМОЖНОСТИ АВТОРИЗАЦИИ ПОЛЬЗОВАТЕЛЯ.....</b>	<b>14</b>
<b>СОЗДАНИЕ КОРЗИНЫ ПОКУПАТЕЛЯ.....</b>	<b>15</b>
<b>ЗАКЛЮЧЕНИЕ.....</b>	<b>22</b>
<b>СПИСОК ИСТОЧНИКОВ.....</b>	<b>23</b>

## **Введение**

Ruby on Rails является средой, облегчающей разработку, развертывание и обслуживание веб-приложений. За относительно небольшое время малоизвестная технология Rails выросла до феномена мирового масштаба. Даже несмотря на то, что в настоящее время существует множество фреймворков и платформ для создания готовых веб-приложений, множество профессиональных веб-разработчиков отдают предпочтение Ruby on Rails при создании по-настоящему востребованных во всем мире веб-сайтов.

Все Rails-приложения выполняются с использованием архитектуры Модель-Представление-Контроллер (Model-View-Controller MVC). Привычная Java-разработчикам среда выполнения, к примеру Tapestry или Struts, тоже основана на MVC. Но Rails идет в использовании MVC еще дальше: при ведении разработки в Rails вы начинаете уже с работающего приложения, в котором есть место для каждой части кода, и все части вашего приложения стандартным образом взаимодействуют друг с другом.

Профессиональные программисты пишут тесты для своих работ. И здесь как никто удобна Rails, ведь все Rails-приложения имеют встроенное тестирование. По мере добавления к программному коду какой-либо функциональной возможности Rails автоматически создает программные заглушки тестов, предназначенные для её тестирования. Эта среда облегчает тестирование своих приложений, упрощая жизнь разработчикам.

Не мало важно и то, что все Rails-приложения пишутся на Ruby - современном объектно-ориентированном языке программирования. Несмотря на простоту и лаконичность языка, он позволяет выражать свои идеи четко и естественно. В результате, программы легко пишутся, становятся короче и легко читаются по прошествии значительного времени.

В среде Ruby on Rails разработчики работают в основном с динамическим содержимым. В Rails есть множество способов создания динамических шаблонов. Наиболее распространенный из них, которым мы здесь воспользуемся, заключается во вставке кода Ruby непосредственно в шаблон. Именно поэтому наш файл шаблона называется `hello.html.erb` — суффикс `.html.erb` предписывает Rails расширить содержимое файла с помощью системы, которая называется ERB. ERB — это фильтр, устанавливаемый как часть Rails, который берет файл `.erb` и выдает преобразованную версию. Выходной файл в Rails чаще всего имеет формат HTML, но вообще-то может иметь какой угодно формат. Обычно содержимое пропускается через фильтр без изменений. А содержимое, находящееся между группами символов `<%=` и `%>`, интерпретируется как Ruby-код, который выполняется. В результате этого выполнения содержимое превращается в строку, значение которой подставляется в файл вместо последовательности `<%=...%>`.

Помимо того, что Ruby on Rails позволяет быстрее и с меньшей долей формализма удовлетворять требованиям заказчиков, данная среда позволяет значительно экономить время, выполняя простые задачи всего парой слов в строчке кода и предоставляет работоспособное программное обеспечение уже на ранней стадии цикла разработки. В целом, работа с Rails представляет собой оперирование уже готовым функционалом различных библиотек — RubyGems — для быстрого построения основ реализуемого приложения.

## **СОЗДАНИЕ ПРИЛОЖЕНИЯ**

В данной научно-исследовательской работе рассматривался функционал Ruby on Rails на примере создания простого веб-приложения типа «интернет-магазин». То есть приложение должно работать на локальном сервере и иметь минимальный необходимый функционал сайтов данного типа.

### **УСТАНОВКА НЕОБХОДИМЫХ КОМПОНЕНТОВ, НАСТРОЙКА ГЕМОВ. СОЗДАНИЕ БАЗЫ ДАННЫХ**

Начнем с того, что работа со средой Rails подразумевает наличие следующих необходимых программных средств:

- интерпретатором Ruby (т.к. система Rails написана на Ruby, и во всех приложениях используется Ruby-код). В данном случае была выбрана версия Ruby 2.2. Здесь также следует отметить, что на версии Ruby ниже 2.0.0 эта среда работать не будет;
- системой Ruby on Rails. В данном случае была выбрана версия Rails 4.2;
- интерпретатором JavaScript. Встроенные интерпретаторы этого языка имеются как в Microsoft Windows, так и в Mac OS X, и Rails будет использовать ту версию, которая уже установлена на вашей системе. В других операционных системах может понадобится отдельная установка интерпретатора JavaScript;
- базой данных (одним из первых пунктов создания и разработки веб-приложения является выбор подходящей базы данных). В данном примере используется SQLite 3. Для всех баз данных, кроме SQLite 3, потребуется установить драйвер базы данных, библиотеку, которую Rails сможет использовать для подключения и использования вашего процессора базы данных.

Хочется отметить, что при работе с Ruby on Rails, весьма уместно будет заранее ознакомиться с дополнительными модулями (гемами), так как некоторые из них фактически реализуют существенную долю общей логики приложения и даже пользовательский интерфейс. А для того, чтобы правильно использовать тот или иной гем в своём приложении, необходимо уделить всего несколько минут соответствующей документации. Таким образом, можно воспользоваться тем, что кто-то уже написал дополнительный модуль для

функции, которую нужно реализовать, а сэкономленное время потратить на собственную разработку.

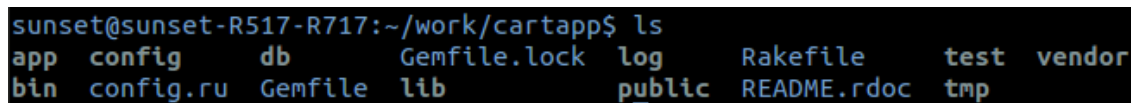
Поэтому перед тем, как начать разработку, следует уделить внимание выбору и настройке подходящих гемов, так как установка каждого из них может потребовать выполнения определенных операций, таких как внесение изменений в код отдельных файлов приложения, и требует совместимости версий с другими гемами.

Каждый гем отвечает за определенный функционал будущего приложения и имеет свою специфику использования, что еще раз говорит о важности предыдущего замечания.

### ЗАПУСК АВТОНОМНОГО СЕРВЕРА

При установке среды Rails мы получаем новый инструмент командной строки — **rails** — который используется для конструирования каждого нового Rails-приложения. Данная команда создаёт готовую структуру каталогов и заполняет её стандартным rails-кодом.

Для начала надо дать название нашему приложению — пусть оно будет называться cartapp(сокращение от Shopping Cart Application). С помощью команды **rails new cartapp** было создано новое веб-приложение, которое в дальнейшем и будет программироваться, как интернет-магазин. Если вывести содержимое только что созданного каталога cartapp, можем увидеть следующую картину:



```
sunset@sunset-R517-R717:~/work/cartapp$ ls
app  config  db      Gemfile.lock  log      Rakefile  test  vendor
bin  config.ru  Gemfile  lib           public   README.rdoc  tmp
```

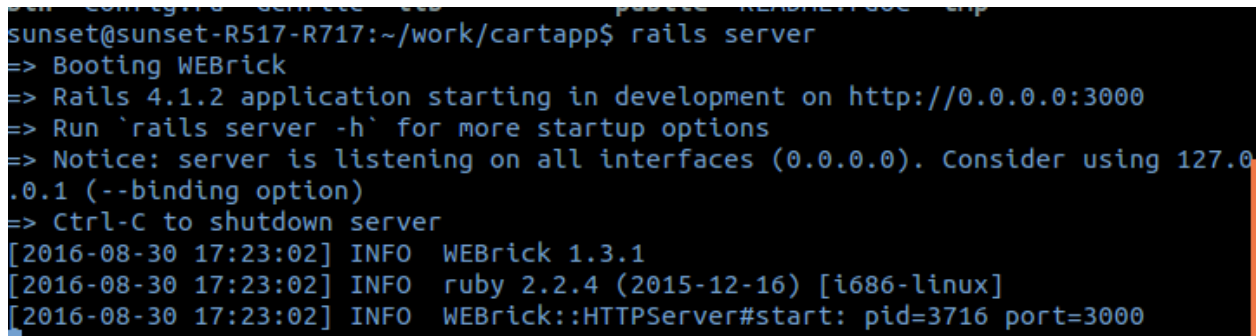
Рис.1

Представленные каталоги и их содержимое представляют собой всё, что нужно для запуска автономного(локального) веб-сервера, способного выполнять наше приложение под названием cartapp. Для запуска работы сервера в корневом каталоге приложения необходимо выполнить команду cartapp> **rails server** (или же просто **rails s**).

Какой веб-сервер запущен, зависит от того, какой сервер установлен. WEBrick — веб-сервер, специально предназначенный для Ruby, который распространяется вместе с Ruby и поэтому является гарантированно

доступным. Но если на вашей системе установлен другой веб-сервер (и Rails может его найти), команда rails server может предпочесть его серверу WEBrick. Rails можно заставить воспользоваться WEBrick, предоставив команде rails соответствующий ключ:

cartapp> **rails server webrick**



```
sunset@sunset-R517-R717:~/work/cartapp$ rails server
=> Booting WEBrick
=> Rails 4.1.2 application starting in development on http://0.0.0.0:3000
=> Run `rails server -h` for more startup options
=> Notice: server is listening on all interfaces (0.0.0.0). Consider using 127.0.0.1 (--binding option)
=> Ctrl-C to shutdown server
[2016-08-30 17:23:02] INFO WEBrick 1.3.1
[2016-08-30 17:23:02] INFO ruby 2.2.4 (2015-12-16) [i686-linux]
[2016-08-30 17:23:02] INFO WEBrick::HTTPServer#start: pid=3716 port=3000
```

Рис.2

Результатом выполнения команды должна быть трассировка в окне терминала, показывающая, что приложение было запущено. Для работы приложения сервер должен непрерывно работать в этом консольном окне. Позже, после разработки и редактирования кода приложения, у нас будет возможность воспользоваться этим консольным окном для отслеживания входящих запросов.

Конкретно в последней строчке трассировка запуска показывает, что был запущен веб-сервер с использованием порта 3000. То есть, мы можем получить доступ к приложению, указав веб-браузеру URL-адрес <http://localhost:3000>.

## СОЗДАНИЕ ТАБЛИЦЫ ТОВАРОВ, И ИСПОЛЬЗОВАНИЕ ГЕНЕРАТОРА ВРЕМЕННОЙ ПЛАТФОРМЫ (SCAFFOLD) ДЛЯ УЧЕТА ТОВАРОВ В ЭТОЙ ТАБЛИЦЕ

Как было сказано ранее, все Rails-приложения выполняются с использованием архитектуры *MVC(Model-View-Controller)*, соответственно на структурирование веб-приложений должны накладываться определённые (весьма серьёзные) ограничения, что в данном случае только упрощают их создание и программирование.

То есть, работая в среде Ruby on Rails вы разрабатываете модели, представления и контроллеры отдельными функциональными блоками, после чего они связываются при выполнении вашей программы. В Rails-приложении входящий запрос сначала будет подан маршрутизатору, который решает, в

какое место приложения должен быть отправлен запрос и как должен быть произведен синтаксический разбор этого запроса. На данном этапе в коде контроллера идентифицируется конкретный метод, называемый в Rails *действием*. Действие ищет запрошенные данные, может взаимодействовать с моделью и искать другие действия.

Небольшая историческая справка: впервые новая архитектура интерактивных приложений была предложена в 1979 году Трюгве Реенскауг. По его мнению все приложения разбиваются на компоненты трёх типов: модели, представления и контроллеры. Модель отвечает за поддержку состояния приложения. Представление отвечает за формирование пользовательского интерфейса, который обычно основывается на считывании данных с модели. Контроллеры же организуют работу приложения. Они воспринимают изменения, вводимые пользователем, взаимодействуют с моделью и отображают соответствующее представление для пользователя.

Итак, чтобы создать простейшее приложение, нам нужен код для контроллера и представления, а также связывающий их маршрут. Начнем с контроллера. Определившись, как должно выглядеть меню будущего сайта, необходимо сгенерировать соответствующий контроллер. В данном случае главное меню будет состоять из пунктов Page, Home, About, FAQs, Contacts.

Аналогично тому, как для создания нового Rails-приложения мы воспользовались командой rails, для создания нового контроллера для нашего проекта мы можем также воспользоваться генерирующим сценарием. Для этого выполним следующую команду в терминале:

```
cartapp>rails g controller page home bout faqs contact
```

Теперь, для того, чтобы попасть на страницу каждого из разделов, необходимо в файле application.html.erb (он находится в директории /app/views/layouts/) добавить ссылку путь для каждого из них.

Чтобы узнать этот самый путь, можно воспользоваться командой

**rake routes**

Данная команда выведет раскладку всех маршрутов, как на рис.3



	Prefix	Verb	URI Pattern	Controller#Action
new_user_session	GET		/users/sign_in(:format)	devise/sessions#new
user_session	POST		/users/sign_in(:format)	devise/sessions#create
destroy_user_session	DELETE		/users/sign_out(:format)	devise/sessions#destroy
user_password	POST		/users/password(:format)	devise/passwords#create
new_user_password	GET		/users/password/new(:format)	devise/passwords#new
edit_user_password	GET		/users/password/edit(:format)	devise/passwords#edit
	PATCH		/users/password(:format)	devise/passwords#update
	PUT		/users/password(:format)	devise/passwords#update
cancel_user_registration	GET		/users/cancel(:format)	devise/registrations#cancel
user_registration	POST		/users(:format)	devise/registrations#create
new_user_registration	GET		/users/sign_up(:format)	devise/registrations#new
edit_user_registration	GET		/users/edit(:format)	devise/registrations#edit
	PATCH		/users(:format)	devise/registrations#update
	PUT		/users(:format)	devise/registrations#update
	DELETE		/users(:format)	devise/registrations#destroy
cart	GET		/cart(:format)	cart#index
cart_clear	GET		/cart/clear(:format)	cart#clearCart
	GET		/cart/:id(:format)	cart#add
products	GET		/products(:format)	products#index
	POST		/products(:format)	products#create
new_product	GET		/products/new(:format)	products#new
edit_product	GET		/products/:id/edit(:format)	products#edit
product	GET		/products/:id(:format)	products#show
	PATCH		/products/:id(:format)	products#update
	PUT		/products/:id(:format)	products#update
	DELETE		/products/:id(:format)	products#destroy
root	GET		/	page#home
page_about	GET		/page/about(:format)	page#about
page_faqs	GET		/page/faqs(:format)	page#faqs
page_contact	GET		/page/contact(:format)	page#contact

Рис.3

Выбирая необходимый путь, создаём ссылку на него:

```
<%= link_to "Home", root_path %>
```

или

```
<%= link_to "Contacts", page_contact_path %>
```

Подобную операцию следует проделывать для каждого нового раздела на сайте (например, вход, регистрация и т. д.). Это так называемый роутинг в Rails. Роутер Rails распознает URL и соединяет их с экшном контроллера. Он также создает пути и URL, избегая необходимость жестко прописывать строки в ваших представлениях.

Так как будущее приложение должно напоминать интернет-магазин, необходимо создать перечень товаров на сайте. Что конкретно нужно сделать: создать таблицу базы данных и модель Rails, которая позволит нашему приложению использовать эту таблицу, а также создать ряд представлений для формирования пользовательского интерфейса и контроллер, управляющий приложением. Начнем с таблицы Listing products — создадим для неё модель, представления, контроллер и миграцию. Как и во многих других случаях, в Rails всё это можно сделать, воспользовавшись одной командой, попросив

Rails сгенерировать *временную платформу scaffold* для заданной модели. Также можно сразу указать параметры(поля) нашей будущей модели `product` ( в данном случае: описание, адрес изображения, цена, категория, подкатегория) и тип каждого из них . В итоге необходимо выполнить всего одну команду:

```
cartapp> rails g scaffold product title:string description:text image_url:string price:integer category:string subcategory:string
```

Далее, после внесения каких-либо изменений нужно заставить Rails применить эту миграцию к нашей базе данных. Сделать это можно с помощью команды **rake**. Данная команда является надежным помощником реализации любой миграции, которая еще не применялась в базе данных.

```
cartapp> rake db:migrate
```

Rake находит миграции, которые еще не применялись к базе данных, и применяет их. В нашем случае к базе данных, определенной в разделе *development* файла `database.yml` , добавляется таблица **products**.

## ПРОВЕРКА ПРИЕМЛИМОСТИ ДАННЫХ. БЛОЧНОЕ ТЕСТИРОВАНИЕ МОДЕЛЕЙ

Таблица товаров создана и готова к заполнению. Но существенным минусом является то, что новые продукты будут заноситься в базу данных даже несмотря на отсутствие каких-либо данных критически важных полей (например, неправильно указанная цена или отсутствие названия товара). Чтобы исправить это, необходимо ввести в приложение проверку приемлемости данных. Другими словами необходимо сконцентрировать внимание на повышении надёжности модели и преградить ошибкам, допущенным при вводе, путь в базу данных.

Соединяющим звеном между программным кодом и базой данных является не что иное, как уровень модели. Поэтому модель становится идеальным местом для описания проверки. Вне зависимости от источника поступления данных (будь то форма или программные изменения), если модель будет проверять их перед записью - база данных будет защищена от некорректных данных.

Начнём с исходного кода класса модели (app/models/product.rb)

```
class Product < ActiveRecord::Base
```

```
#добавив следующую строчку мы гарантируем проверку наличия данных в  
# указанных полях
```

```
validates :title, :description, :image_url, presence: true
```

```
end
```

Метод `validates()` является в Rails стандартным средством проверки (валидатором). Он будет проверять одно или несколько полей модели на соблюдение одного или нескольких условий. Часть инструкции `presence: true` преписывает данному методу проверять наличие данных в каждом из указанных полей. На рис. можно увидеть, что получится, если мы попробуем отправить сведения о новом товаре, не заполнив ни одного поля. Картина будет весьма впечатляющей: поля с ошибками будут выделены, а сводка об ошибках помещена в красочном списке в верхней части формы. Неплохо для всего одной строки кода.

Home Shop About FAQs Contacts

## New product

3 errors prohibited this product from being saved:

- Title can't be blank
- Description can't be blank
- Image url can't be blank

Title

Description

Image url

Price

Рис.4

Также можно заметить, что после редактирования и сохранения файла `product.rb` перезапускать приложение для проверки изменений не пришлось — перезагрузка страницы, заставившая ранее Rails заметить изменения в схеме данных, всегда приводит к тому, что используется самая последняя версия кода. Нужно также проверить, что цена имеет допустимое, положительное числовое значение. Для проверки воспользуемся методом с выразительным именем `numericality`. Мы также передадим несколько многословному методу `greater_than_or_equal_to` (больше чем или равно) значение `0.01`:

```
validates :price, numericality: { greater_than_or_equal_to: 0.01 }
```

Теперь, если добавить товар с недопустимой ценой, появится соответствующее уведомление, показанное на рис.

The screenshot shows a web form titled "New product" with a dark navigation bar at the top containing links for Home, Shop, About, FAQs, and Contacts. The form fields are as follows:

- Title:** A text input field containing "New product".
- Description:** A text area containing "simple text simple text simple textsimple textsimple text".
- Image url:** A text input field containing "image.url".
- Price:** A number input field containing "hello!". This field is highlighted with a red border, and a red tooltip message "Пожалуйста, введите число." (Please enter a number.) is displayed below it.
- Subcategory:** An empty text input field.
- Create Product:** A button to submit the form.
- Back:** A link to return to the previous page.

Рис. 5

Почему проверка велась относительно значения 0.01, а не нуля? Потому что можно ведь ввести в это поле и такое значение, как 0.001. Поскольку база данных сохраняет только две цифры после десятичной точки, в ней окажется нуль, даже если поле пройдет проверку на ненулевое значение. Проверка того, что число, по крайней мере, равно 0.01, гарантирует, что будет сохранено только допустимое значение.

В итоге, добавив подобные проверки для каждого из полей, обновленная модель Product должна приобрести следующий вид:

```
class Product < ActiveRecord::Base

  validates :title, :description, :image_url, presence: true

  validates :price, numericality: {greater_than_or_equal_to: 0.01}

  validates :title, uniqueness: true

  validates :image_url, allow_blank: true, format: {
    with: %r{\.(gif|jpg|png)\Z}i,
    message: 'must be a URL for GIF, JPG or PNG image.'

    # URL должен указывать на изображение формата GIF, JPG
    #или PNG
  }

end
```

Как видно, простое добавление проверки приемлемости данных придало страницам ведения перечня товаров более внушительный вид. Прежде чем двигаться дальше, воспользуемся еще одной командой:

```
car tapp>rake test
```

В выводе этой команды должны присутствовать две строки, в каждой из которых сообщается: 0 failures, 0 errors (0 сбоев и 0 ошибок). Данный тест относится к тестам модели и контроллера, которые Rails генерирует вместе с созданием временной платформы. Запуск этой команды поможет выявить и отследить ошибки.

**Замечание.** При использовании базы данных, отличной от SQLite3 , тестирование может не пройти.

## ДОБАВЛЕНИЕ ВОЗМОЖНОСТИ АВТОРИЗАЦИИ ПОЛЬЗОВАТЕЛЯ

Далее необходимо создать систему авторизации пользователя. Разумеется, можно реализовывать все необходимые функции самостоятельно, но лучше сэкономить время и воспользоваться готовым модулем Devise. Данный гем является одним из лучших для аутентификации в rails-приложениях и требует некоторой настройки (<https://github.com/plataformatec/devise>). После установки гема, он еще никак не взаимодействует с разрабатываемым приложением. Для завершения установки необходимо запустить один из встроенных генераторов, имеющихся в арсенале Devise:

```
car tapp>rails generate devise:install
```

Этот генератор установит инициализатор, в котором описаны все конфигурационные настройки Devise, необходимые для работы, а также файл с базовой локалью (английский язык). Также установщик предложит нам выполнить базовую настройку. Далее необходимо выполнить базовую пользовательскую настройку гема (обозначить путь перенаправления при входе в систему и т.д.). В итоге мы имеем готовую систему аутентификации пользователя, которая хранит регистрационные данные в базе и сопоставляет их с данными, вводимыми пользователем. Добавив в файл роутера (/config/routes.rb) строку

```
devise_for :users
```

и перейдя по адресу «*localhost:3000/users/sign\_up*», мы перейдем на страницу, где производится регистрация пользователей.

Если позаботится также о внешнем виде будущего приложения, добавив таблицу стилей в файл `cartapp/app/assets/stylesheets/application.css` и соответствующие изменения в `cartapp/app/views/layouts/application.html.erb`, в итоге получится примерно такой результат:

The image displays two screenshots of a web application interface, likely for a shopping cart. Both screenshots feature a dark navigation bar at the top with links: Home, Shop, About, FAQs, Contacts, Sign In, and Sign Up.

The top screenshot shows the "Sign up" form. It includes input fields for "Email", "Password (6 characters minimum)", and "Password confirmation". A "Sign up" button is present, along with a "Log in" link.

The bottom screenshot shows the "Log in" form. It includes input fields for "Email" and "Password". There is a checkbox for "Remember me" and a "Log in" button. Additionally, there are links for "Sign up" and "Forgot your password?".

Рис.6

## СОЗДАНИЕ КОРЗИНЫ ПОКУПАТЕЛЯ

Следующим важным этапом будет добавление на сайт корзины покупателя. По сути, нам необходимо создать новый контроллер, который будет отвечать за создание списка выбранных товаров и считать их общую стоимость. Необходимый нам файл называется `cart_controller.rb` (и находится в директории `/cartapp/app/controllers/`). Следующий код создаёт простейшую корзину покупателя:

```
class CartController < ApplicationController
```

```
  def add
```

```
    id= params[:id]
```

```
    #если корзина покупателя уже была создана, используется текущая, иначе создаётся #новая
```

```
    if session[:cart] then
```

```
      cart = session[:cart]
```

```
    else
```

```
      session[:cart] = {}
```

```
      cart= session[:cart]
```

```
    end
```

```
    # если продукт уже был добавлен в корзину, увеличивает его счётчик на 1, иначе устанавливает его значение равным единице
```

```
    if cart[id] then
```

```
      cart[id] = cart[id] + 1
```

```
    else
```

```
      cart[id] = 1
```

```
    end
```

```
    redirect_to :action => :index
```

```
  end
```

```
  def clearCart
```

```
    session[:cart] = nil
```

```
    redirect_to :action => :index
```

```
  end
```

```
  def index
```

```
    #если существует корзина, вывести её содержимое на страницу, иначе вывести #пустую корзину
```



```

    if session[:cart] then
      @cart = session[:cart]
    else
      @cart = {}
    end
  end
end
end

```

Следующим шагом будет создание страницы, отображающей саму корзину и её содержимое. В данном примере был модифицирован файл *index.html.erb*

```

<h1>You cart</h1>
#Если корзина пустая — вывести соответствующее сообщение
<% if @cart.empty? %>
  <p>Your cart is currently empty</p>
<% else %>
  # Возможность удаления товаров из корзины одним действием
  <%= link_to 'Empty Your Cart', cart_clear_path %>
<% end %>
<br><br><br>

#Подсчёт суммарной стоимости товаров
<% total = 0 %>
<ul>
<% @cart.each do | id,quantity | %>
  <% product = Product.find_by_id(id) %>
  <li>
    <%= link_to product.title, product %>
    <p><%= product.description %></p>

```

```

<p><%=number_to_currency(product.price,:unit=>'$'%></p>
<p>Quantity: <%= quantity %> </p>
</li>
<% total += quantity * product.price %>
<% end %>

<p><b><%= number_to_currency(total, :unit => '$' ) %></b></p>
</ul>

```

В итоге может получиться нечто подобное:

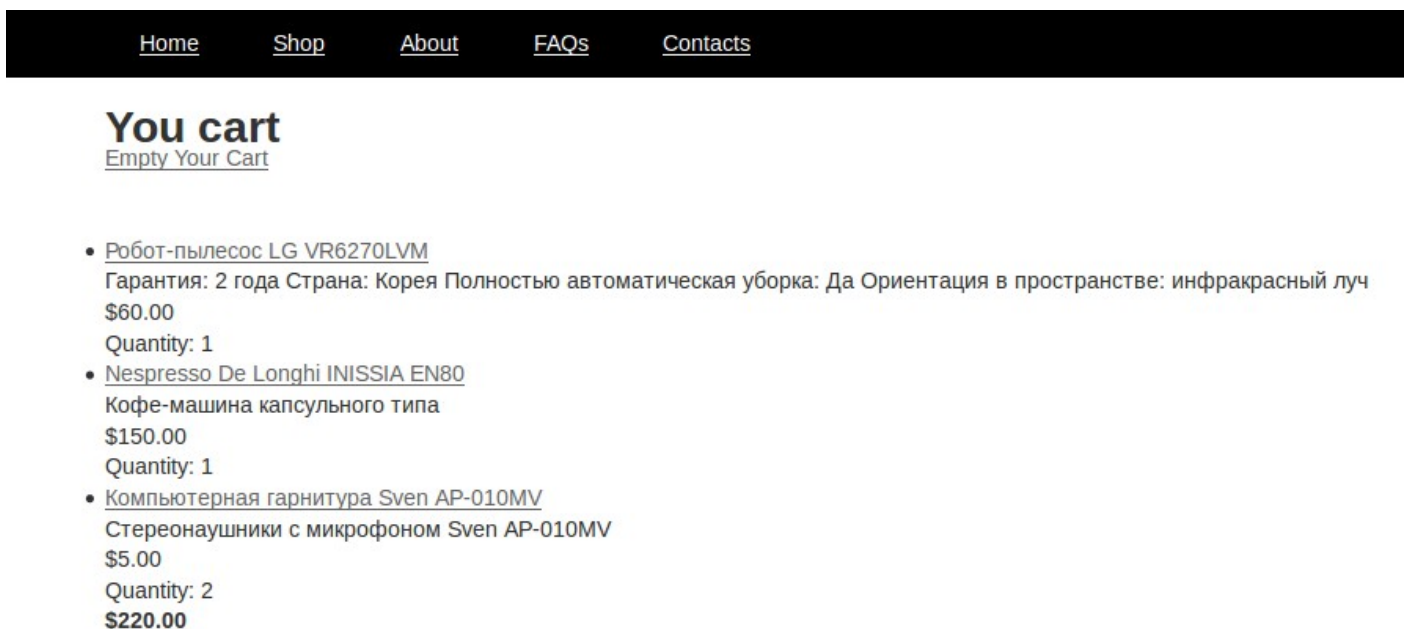


Рис.7

Теперь, для того, чтобы интернет-магазин работал корректно, необходимо учесть такие детали, как запрет покупки для неавторизованных пользователей. То есть, при входе на сайт без авторизации, ссылка *Add To Cart*, должна направить пользователя на страницу авторизации. Для этого в файле `cart_controller.rb` добавим следующую строку:

```
before_action :authenticate_user!, except: [:index]
```

Также, если выйти из своей учётной записи на сайте, нажав на ссылку **Log out**, и зайти в корзину — она будет отображаться как пустая. Ну и вконец-концов, если нам необходимо, чтобы неавторизованный пользователь имел минимальный доступ к нашему интернет-магазину, необходимо внести изменения в файл контроллера страницы (`/app/controllers/page_controller`)

```
before_action :authenticate_user!, only: [:contact]
```

Таким образом, гость сайта не имеет доступа к странице контактов интернет-магазина.

Заключительным этапом будет создание формы для ввода информации о заказе. Вначале мы создадим модель заказов `order`:

```
cartapp> rails generate scaffold Order name address:text email pay_type
```

Далее необходимо создать кнопку «Оформить заказ», добавить проверку корзины на пустоту при оформлении, и конечно же создать новое действие для ввода персональной информации: имени, почтового и электронного адреса и способа оплаты. Добавленный код будет отправлять пользователя обратно в каталог, если в его корзине ничего нет. Поля ввода самой формы нужно связать с соответствующими атрибутами объекта модели Rails, поэтому нам нужно создать в действии `new` пустой объект модели, чтобы дать этим полям что-нибудь, с чем они будут работать. Нужно выполнить обычную для HTML-форм задачу: заполнить начальными значениями поля формы, а затем извлечь эти значения в наше приложение, когда пользователь щелкнет на кнопке передачи данных.

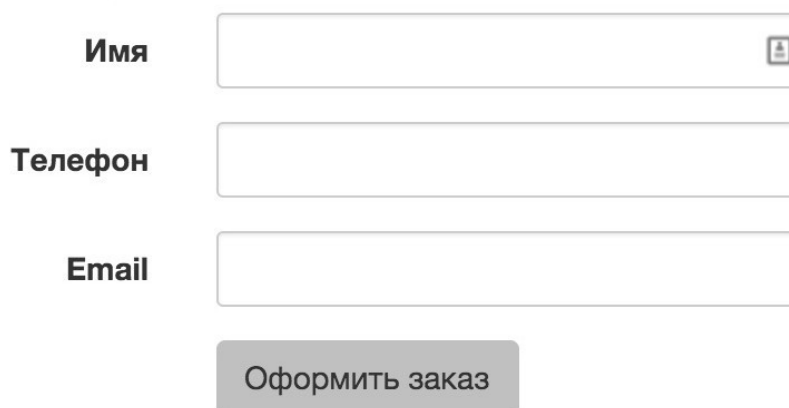
Для ссылки на новый объект модели `Order` в контроллере объявляется новая переменная экземпляра `@order`. Наши действия обусловлены тем, что представление заполняет форму, используя данные этого объекта. Само по себе это обстоятельство не представляет интереса: поскольку модель новая, все поля будут пустыми. Но рассмотрим какой-нибудь общий случай. Возможно, нам захочется отредактировать существующий заказ. Или пользователь может попытаться подтвердить заказ, но его данные не пройдут проверку. В таких случаях нам захочется, чтобы данные, существующие в модели, были показаны пользователю при отображении формы. Передача на данной стадии пустого объекта модели сохраняет совместимость со всеми этими случаями — представление всегда сможет иметь доступ к этому объекту модели. Затем,

когда пользователь щелкнет на кнопке передачи информации, желательно, чтобы новые данные извлекались из формы и, возвращаясь контроллеру, передавались в объект модели. В Rails имеются помощники для разных HTML-элементов формы. В данном примере для получения имени, а также почтового и электронного адресов клиента использовались помощники `text_field`, `email_field` и `text_area`. Единственная сложность связана со списком выбора. Подразумевается, что список возможных вариантов оплаты является свойством модели `Order`. Определим в файле модели `order.rb` (`/app/models/order.rb`) дополнительный массив:

```
PAYMENT_TYPES = [ "Check", "Credit card", "Purchase order" ]
```

Форма готова к использованию.

## Оформление заказа



The form consists of three input fields stacked vertically, each with a label to its left. The first field is labeled 'Имя' (Name) and has a small icon of a person in the bottom right corner. The second field is labeled 'Телефон' (Phone). The third field is labeled 'Email'. Below these fields is a gray button with the text 'Оформить заказ' (Place order).

Рис.8

В заключении создадим документацию проделанной работы. Для создания привлекательной документации программиста Rails упрощает запуск имеющейся в Ruby утилиты `RDoc` 1 в отношении всех имеющихся в приложении исходных файлов. Но перед генерированием этой документации нам, наверное, нужно создать привлекательную начальную страницу, чтобы будущее поколение разработчиков знало, чем занимается наше приложение. Для этого нужно отредактировать файл `README.rdoc` и ввести в него все, что вы считаете полезным. Для создания документации в HTML-формате можно воспользоваться следующей командой `rake`:

```
cartapp> rake doc:app
```

С ее помощью будет сгенерирована документация, которая будет помещена

в каталог `doc/app`. И наконец, может заинтересовать объем созданного программного кода. Для этого в `Rake` также имеется отдельная задача.

```
cartapp> rake stats
```

Если все это проанализировать, получится, что при весьма скромном объеме программного кода выполнен довольно большой объем работы. Более того, весьма существенная часть этого кода была для вас сгенерирована. В этом и заключается основное преимущество `Rails`.

## ЗАКЛЮЧЕНИЕ

В ходе учебной (научно-исследовательской) практики были достигнуты следующие результаты:

1. Проведён анализ основных технологий и программных средств среды Ruby on Rails, используемых для разработки веб-приложений .
2. Проведено исследование Rails-разработки на основе создания корзины покупателя интернет-магазина. Конкретно были проиллюстрированы следующие пункты: создание простых страниц обслуживания покупателей, как они привязываются к таблицам базы данных, как обрабатываются сессии и создаются простые формы.
3. Установка, настройка и работа с библиотеками дополнительных модулей Rails.
4. На основе принципов архитектуры MVC был построен функционал сайта и настроена работа с пользователем.
5. Проведено блочное тестирование всех полей соответствующих форм, настроена правильная обработка данных, вводимых пользователем.
6. Разработаны элементы пользовательского интерфейса для анализа и модификации полученных данных.
7. Создана документация проделанной работы.

## ИСПОЛЬЗУЕМАЯ ЛИТЕРАТУРА

1. «Rails. Гибкая разработка веб-приложений». Сэм Руби, Дэйв Томас, Дэвид Хэнсон.
2. Статья, посвященная дополнительному модулю Devise —  
URL: <https://habrahabr.ru/post/208056/>
3. Официальная страница с инструкцией по установке и использованию гема Devise — URL: <https://github.com/plataformatec/devise>
4. Официальная страница русского сообщества Ruby on Rails —  
URL: <http://rusrails.ru/>
5. Страница загрузки последней стабильной версии Ruby —  
URL: <https://www.ruby-lang.org/ru/downloads>
6. Официальный сайт TIOBE – URL: [http://www.tiobe.com/tiobe\\_index](http://www.tiobe.com/tiobe_index)
7. Официальный сайт фреймворка Ruby on Rails – URL: <http://rubyonrails.org>













































