# **4** РАЗРАБОТКА ПРОГРАММНЫХ МОДУЛЕЙ

## **4.1** Управляющая программа

Raspbian — это официальная операционная система для Raspberry Pi, она разработана специально для этого устройства и имеет все необходимое программное обеспечение. Raspbian основана на ARM версии Debian 8 Jessie и содержит такие программы по умолчанию — офисный пакет LibreOffice, веб-браузер, почтовый клиент — Claws Mail, легкое окружение рабочего стола, а также некоторые инструменты для обучения программированию.

Дистрибутив Raspbian (см. рисунок 4.1) не привязан к конкретной модели Pi, более того, в комплекте с ним поставляются драйверы одобренных Raspberry PI Foundation устройств, включая Wi-Fi-донгл Pi USB, модули камер (v1 и v2), плату расширения Raspberry Pi Sense HAT. Raspbian дополнительно комплектуется проверенным программным обеспечением и утилитами.

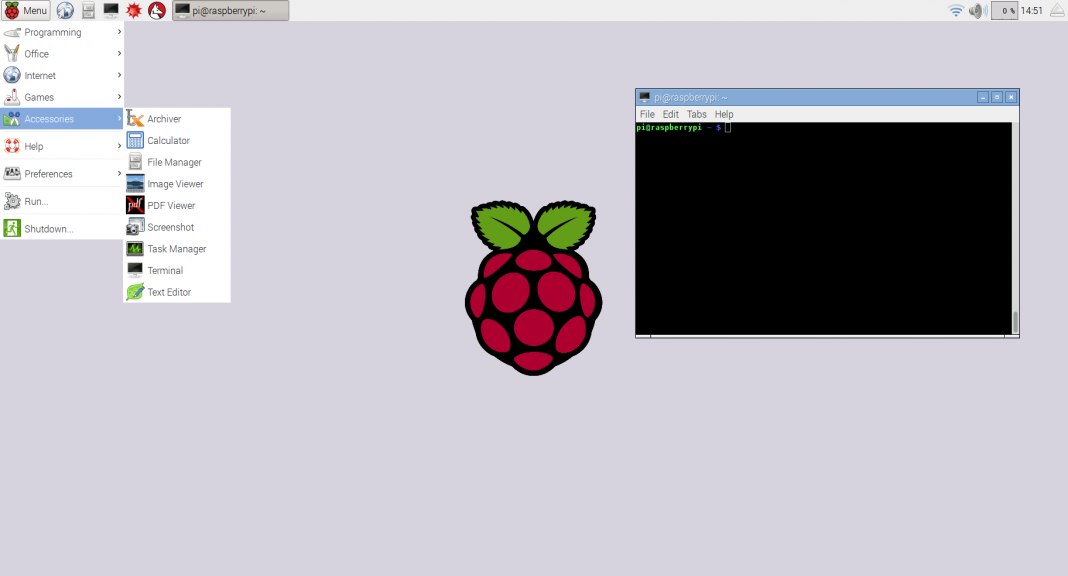


Рисунок 4.1 – Рабочий стол Raspbian OC.

Для установки ОС необходима MicroSD карточка. В данном проекте была использована карточка на 8 GB. Этого вполне достаточно для установки ОС.

Для записи ОС необходимо совершить побайтовую запись скачанного и распакованного образа Raspbian на карту памяти:

$ sudo dd bs=4M if=./2017-11-29-raspbian-stretch-lite.img of=/dev/sdd status=progress conv=fsync

После чего уже карту необходимо вставить в Raspberry Pi, подключить и работать с ней напрямую.

При первой загрузки ОС необходимо совершить ряд первоначальных настроек.

Первое действие после загрузки – это создание пользователя.

# uname –a

# adduser

# usermod -a -G sudo

# userdel -r pi

# apt update && apt upgrade && apt dist-upgrade

# rpi-update && reboot

После совершённых настроек и перезагрузки системы – вторым шагом настройки является настройка сети. Для этого необходимо произвести настройку /etc/network/interfaces файла. Настройка выглядит следующим образом:

source-directory /etc/network/interfaces.d

auto lo

iface lo inet loopback

#iface eth0 inet manual

allow-hotplug eth0

iface eth0 inet static

address 192.168.0.100

netmask 255.255.255.0

gateway 192.168.0.1

# dns-\* options are implemented by the resolvconf package, if installed

dns-nameservers 8.8.8.8 8.8.4.4

allow-hotplug wlan0

iface wlan0 inet manual

wpa-conf /etc/wpa\_supplicant/wpa\_supplicant.conf

После настройки сети для данного дипломного проекта необходимо настроить использование Wi-Fi модуля. Для этого необходимо отредактировать файл wpa\_supplicant.conf.

Ко всему прочему, нужно активизировать ssh протокол для удалённой работы с Raspberry Pi и установить дополнительный пакет для автоматического поднятия правил после перезагрузки системы.

# systemctl enable ssh

# apt install iptables-persistent

# chmod +x /etc/iptables.rules.sh

# /etc/iptables.rules.sh

После настройки сетевого взаимодействия необходимо произвести настройку памяти.

Для увеличения ресурса SD-карты необходимо отключить swap-память.

# phys-swapfile swapoff

# dphys-swapfile uninstall

# systemctl disable dphys-swapfile

Так же отключаем Bluetooth демона, что он не занимал оперативной памяти в системе.

# systemctl stop Bluetooth

Теперь плата готова к работе. Управляющая устройством программа была разработана на языке программирования Python. В операционной системе уже присутствует библиотка PiCamera, которая предоставляет простейшие механизмы работы с камерой Pi NOIR Camera v2, а также библиотека Requests, которая предоставляет механизмы для создания http-запросов. Так же была использована библиотека Time. Все библиотеки в языке Python подключаются с помощью инструкции import, как показано ниже:

import requests

from picamera import PiCamera

from time import sleep

Пользовательские настройки вносятся в управляющую программу посредством изменения значений констант, представленных ниже:

SERVER\_URL = 'http://192.168.43.138:3000'

PHOTO\_FILENAME = 'home\_photo'

DELAY = 300

PREVIEW\_DELAY = 4

IS\_TEMP = True

USER\_ID = 1

TITLE = 'Home camera'

MODE = 'cycle' # or 'single'

* SERVER\_URL – константа, хранящая URL адрес сервера веб-приложения;
* PHOTO\_FILENAME – константа, хранящая шаблон для названия фотографий;
* DELAY – константа, хранящая значение задержки между снимками в секундах;
* PREVIEW\_DELAY – константа, хранящая значение выдержки снимка в секундах;
* IS\_TEMP – константа, в которой указывается, является ли сделанная фотография временной или же необходимо сохранить ее на устройстве;
* USER\_ID– константа, хранящая значение идентификационного номера пользователя, которому принадлежит устройство;
* TITLE – константа, хранящая описание сделанной фотографии;
* MODE – константа, описывающая режим работы программы. Single означает, что программа сделает один снимок и завершится, отправив его на сервер веб-приложения. Cycle означает, что программа будет работать в циклическом режиме, производя фотофиксацию с определенным периодом и отправляя фотографии на сервер;

Точкой входа в управляющую программу является метод \_\_main\_\_, реализация которого представлена ниже:

if \_\_name\_\_ == '\_\_main\_\_':

client = Client(SERVER\_URL, PHOTO\_FILENAME, DELAY, PREVIEW\_DELAY, IS\_TEMP, USER\_ID, TITLE)

if MODE == 'single':

client.single\_capture\_and\_post()

elif MODE == 'cycle':

client.cycle\_capture\_and\_post()

В методе создается объект класса Client, которому, в качестве параметров, в конструктор передаются вышеперечисленные константы. В зависимости от значения константы MODE, запускаются разные режимы программы, которые описаны в методах класса Client, представленного ниже:

class Client:

TEMP\_FILENAME = 'temp\_photo'

def \_\_init\_\_(self, server\_url, photo\_filename=TEMP\_FILENAME, delay=0, preview\_delay=2, is\_temp=True, user\_id=1, title='Title'):

self.server\_url = server\_url

self.photo\_filename = photo\_filename

self.delay = delay

self.preview\_delay = preview\_delay

self.is\_temp = is\_temp

self.user\_id = user\_id

self.title = title

self.my\_camera = MyCamera(self.is\_temp, self.photo\_filename, self.preview\_delay)

self.post\_request = PostRequest(self.server\_url, self.user\_id, self.title)

def single\_capture\_and\_post(self):

filename = self.my\_camera.capture()

self.post\_request.make\_request(filename)

def cycle\_capture\_and\_post(self):

while True:

filename = self.my\_camera.capture()

self.post\_request.make\_request(filename)

sleep(self.delay)

Константа TEMP\_FILENAME хранит в себе временное название файла. Метод \_\_init\_\_ является конструктором объекта. Поле camera является экземпляром класса MyCamera и позволяет работать с методами объекта этого класса. Поле post\_request является экземпляром класса PostRequest и также позволяет работать с методами объекта этого класса. Метод single\_capture\_and\_post(self) вызывает метод capture() объекта camera, который делает снимок и возвращает путь к файлу, который потом передается в метод make\_request(filename) объекта post\_request. Метод cycle\_capture\_and\_post(self) производит аналогичные действия, однако это происходит в бесконечном цикле с задержкой, представленной в виде функции sleep(self.delay), в которую передается поле delay.

Ниже представлена реализация класса MyCamera:

class MyCamera:

def \_\_init\_\_(self, is\_temp, filename, preview\_delay):

self.is\_temp = is\_temp

self.counter = 0

self.filename = filename

self.preview\_delay = preview\_delay

self.camera = PiCamera()

def capture(self):

self.camera.start\_preview()

sleep(self.preview\_delay)

filename = self.filename + str(self.counter) + '.jpg'

self.camera.capture(filename)

self.camera.stop\_preview()

if not self.is\_temp:

self.counter += 1

return filename

Все поля, присутствующие в данном классе, за исключением полей counter и camera, были рассмотрены выше. Поле counter является счетчиком, которое добавляет к названию файла фотографии порядковый номер при режиме сохранения фотографий на устройстве. Поле camera является экземпляром библиотечного класса PiCamera, который предоставляет простейшие механизмы работы с камерой Pi NOIR Camera v2. В методе capture происходит непосредственно процесс фотофиксации, перед которым происходит автоматическая настройка камеры в течении времени, хранящемся в секундах в поле preview\_delay. Если активирован режим хранения фотографий на устройстве, то счетчик counter будет инкрементироваться каждый раз, после создания фотографии.

if not self.is\_temp:

self.counter += 1

Рассмотрим класс PostRequest:

class PostRequest:

def \_\_init\_\_(self, url, user\_id, title):

self.url = url

self.params = (('image[user\_id]', str(user\_id)),('image[title]', str(title)))

def make\_request(self, filename):

self.files = ('image[photo]', open(filename, 'rb'))

requests.post(self.url, files = self.files, data = self.params)

self.files['image[photo]'].close()

Поле params, инициализируемое в конструкторе, представляет собой словарь параметров, которые будут переданы на сервер веб-приложения в http-запросе, а поле files содержит фотографию и так же отправляется в этом запросе. Сам запрос происходит в методе make\_request(filename). Сначала в методе открывается файл фотографии, путь к которому приходит в аргументе filename, в режиме чтения бинарного файла. После этого дескриптор открытого файла присваивается значению в словаре files. Потом вызывается POST метод http-запроса на сервер веб-приложения. После отправки файл фотографии закрывается.

На этом реализация управляющей программы устройства завершена.

## **4.2** Веб-приложение

Разработка веб-приложения велась в редакторе исходного кода Visual Studio Code, который показан на рисунке.4.2. Данный редактор разработала компания Microsoft и позиционирует его как «лёгкий» редактор кода для кроссплатформенной разработки веб- и облачных приложений. Он включает в себя отладчик, инструменты для работы с Git, подсветку синтаксиса IntelliSense, а также средства для рефакторинга.

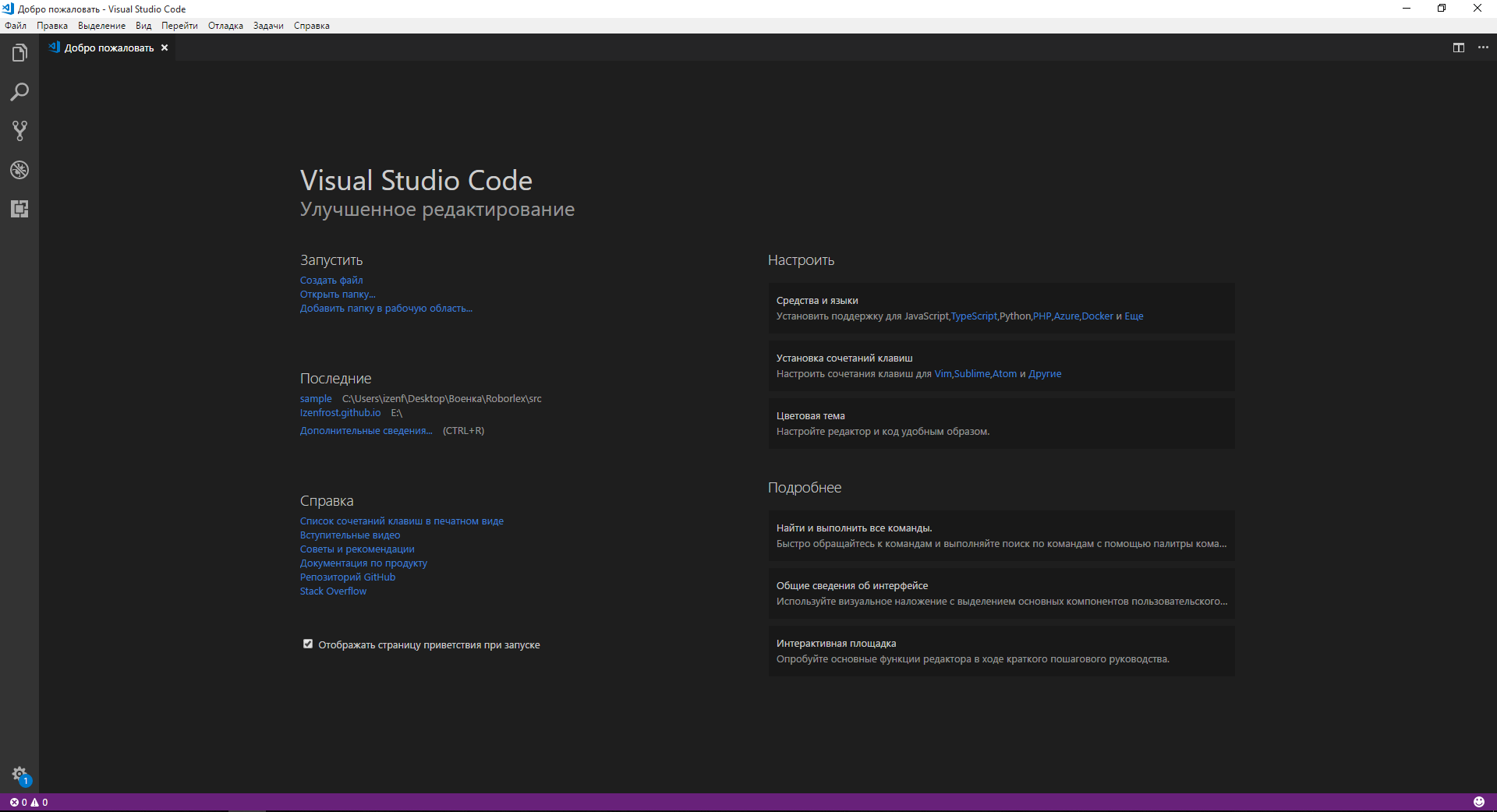


Рисунок 4.2 – Загрузочный экран Visual Studio Code.

Для поддержки работы с различными языками программирования были использованы плагины для языка Ruby, на котором было разработано веб-приложение. Плагины создаются сообществом редактора и являются свободными для загрузки и использования. Окно магазина плагинов показано на рисунке 4.3.

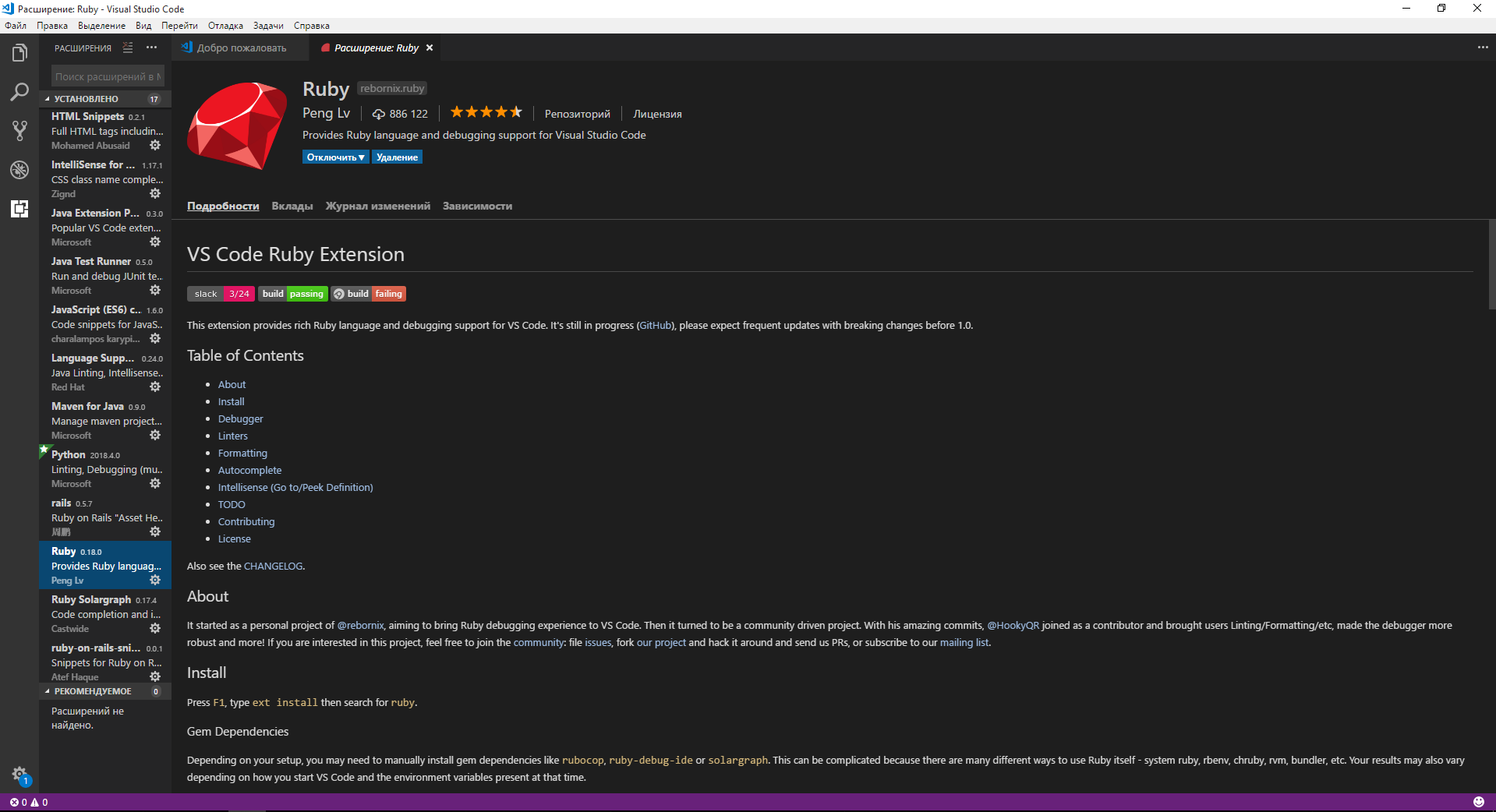


Рисунок 4.3 – Магазин плагинов Visual Studio Code.

Все библиотеки, которые были использованы в разработке находятся в файле Gemfile, в корневой директории проекта, содержание которого приведено ниже:

source 'https://rubygems.org'

git\_source(:github) { |repo| "https://github.com/#{repo}.git" }

ruby '2.4.1'

gem 'rails', '~> 5.2.0'

gem 'pg', '>= 0.18', '< 2.0'

gem 'puma', '~> 3.11'

gem 'sass-rails', '~> 5.0'

gem 'uglifier', '>= 1.3.0'

gem 'coffee-rails', '~> 4.2'

gem 'turbolinks', '~> 5'

gem 'jbuilder', '~> 2.5'

gem 'bootsnap', '>= 1.1.0', require: false

gem 'devise'

gem 'cancancan'

gem 'carrierwave'

group :development, :test do

gem 'byebug', platforms: [:mri, :mingw, :x64\_mingw]

end

group :development do

gem 'web-console', '>= 3.3.0'

gem 'listen', '>= 3.0.5', '< 3.2'

gem 'spring'

gem 'spring-watcher-listen', '~> 2.0.0'

end

gem 'tzinfo-data', platforms: [:mingw, :mswin, :x64\_mingw, :jruby]

Кроме служебных библиотек следует отметить библиотеки Devise, CanCanCan и CarrierWave. Devise предоставляет механизмы для регистрации, авторизации и управления сессией пользователей. CanCanCan позволяет устанавливать уровень привелегий для различных типов пользователей. Бибилотека CarrierWave предоставляет механизмы для загрузки изображений, посредством mulptipart/form-data http-запросов. Библиотека Puma предоставляет реализацию сервера, который прослушивает указанный порт и принимает входящие запросы, сервер конфигурируется в файле puma.rb, который находится в директории config:

threads\_count = ENV.fetch("RAILS\_MAX\_THREADS") { 5 }

threads threads\_count, threads\_count

# Specifies the `port` that Puma will listen on to receive requests; default is 3000.

#

port ENV.fetch("PORT") { 3000 }

# Specifies the `environment` that Puma will run in.

#

environment ENV.fetch("RAILS\_ENV") { "development" }

plugin :tmp\_restart

Количество потоков, которые позволяется создавать серверу указано в переменной threads.count. Порт, который прослушивает сервер указывается в качестве аргумента метода port. Окружение, в котором будет работать веб-приложение конфигурируется методом environment.

После получения входящего запроса, сервер перенаправляет его в соответствующий контроллер, согласно файлу, который выполняет роль маршрутизатора, routes.rb. Содержание приеведено ниже:

Rails.application.routes.draw do

resources :images

get 'home/index'

devise\_for :users

root to: "home#index"

end

Инструкция reosources, создает для контроллера ImagesController сразу все маршруты: на создание, получение, изменение и удаление данных. Метод get, создает маршрут на получение данных, который привязан к методу index контроллера HomeController. Метод devise\_for создает стандартные маршруты библиотеки Devise для модели user. Метод root определяет точку входа на веб-приложение, в данном случае точкой входа является метод index, контроллера HomeController. Реализация данного контроллера представлена ниже:

class HomeController < ApplicationController

skip\_before\_action :authenticate\_user!, only: [:index]

def index

end

end

В методе индекс с моделями данных ничего не происходит, однако он по умолчанию вернет соответствующий вью-файл, который приведен ниже:

<% if user\_signed\_in? %>

<!-- <% if can? :manage, User %> -->

<%= link\_to('Show users list', index\_admin\_user\_path) %>

<!-- <% end %> -->

<%= link\_to('Show images', images\_path) %>

<%= link\_to('Logout', destroy\_user\_session\_path, :method => :delete) %>

<% else %>

<%= link\_to('Register', new\_user\_registration\_path) %>

<%= link\_to('Login', new\_user\_session\_path) %>

<% end %>

Данный вью-файл определяет, авторизован пользователь или нет, если нет, то ему предлагается зарегистрироваться в системе или авторизоваться. Если авторизуется администратор, то ему доступен список пользователей. Если пользователь уже авторизован, то он может перейти к своим фотографиям, которые были присланы с устройства или выйти из системы. За это действие отвечает метод index контроллера ImagesController. Код контроллера представлен ниже:

class ImagesController < ApplicationController

before\_action :set\_image, only: [:show, :edit, :update, :destroy]

# GET /images

# GET /images.json

def index

@images = Image.all

end

# GET /images/1

# GET /images/1.json

def show

end

# GET /images/new

def new

@image = Image.new

end

# GET /images/1/edit

def edit

end

# POST /images

# POST /images.json

def create

@image = Image.new(image\_params)

respond\_to do |format|

if @image.save

format.html { redirect\_to @image, notice: 'Image was successfully created.' }

format.json { render :show, status: :created, location: @image }

else

format.html { render :new }

format.json { render json: @image.errors, status: :unprocessable\_entity }

end

end

end

# PATCH/PUT /images/1

# PATCH/PUT /images/1.json

def update

respond\_to do |format|

if @image.update(image\_params)

format.html { redirect\_to @image, notice: 'Image was successfully updated.' }

format.json { render :show, status: :ok, location: @image }

else

format.html { render :edit }

format.json { render json: @image.errors, status: :unprocessable\_entity }

end

end

end

# DELETE /images/1

# DELETE /images/1.json

def destroy

@image.destroy

respond\_to do |format|

format.html { redirect\_to images\_url, notice: 'Image was successfully destroyed.' }

format.json { head :no\_content }

end

end

private

# Use callbacks to share common setup or constraints between actions.

def set\_image

@image = Image.find(params[:id])

end

# Never trust parameters from the scary internet, only allow the white list through.

def image\_params

params.require(:image).permit(:user\_id, :title, :photo)

end

end

В методе index из базы данных вычитываются все объекты модели Image, метод create позволяет создать новый объект модели, метод update изменить аттрибуты объекта модели с указанным идентификационным номером, а метод destroy удаляет объект модели с указанным идентификационным номером. Исходный код модели Image представлен ниже:

class Image < ApplicationRecord

belongs\_to :user

mount\_uploader :photo, PhotoUploader

end

Аттрибуты класса не представлены в коде модели напрямую, так как с помощью механизмов представляемого родительским классом ApplicationRecord поля из соответствующей таблицы базы данных подключаются как аттрибуты модели. Инструкция belongs\_to указывает на связь модели Image с моделью User. Метод mount\_uploader подключает к модели Image загрузчик, который является реализацией предоставляемых классом CarrierWave::Uploader::Base механизмов для загрузки изображения. Реализация класса загрузчика PhotoUploader представлена ниже:

class PhotoUploader < CarrierWave::Uploader::Base

include CarrierWave::RMagick

storage :file

def store\_dir

"uploads/#{model.class.to\_s.underscore}/#{mounted\_as}/#{model.id}"

end

def default\_url(\*args)

"/images/fallback/" +

[version\_name,"default.png"].compact.join('\_')

end

def scale(width, height)

RMagick.scale width, height

end

version :thumb do

process resize\_to\_fit: [50, 50]

end

def extension\_whitelist

%w(jpg jpeg gif png)

end

def filename

model.title if original\_filename

end

end

Так директива include позволяет подключить библиотек RMagick, которая предоставляет средства обработки изображений. Поле storage хранит ключ, который указывает, какое хранилище для файлов будет использовано, а метод store\_dir генерирует название директории для нового файла. После прохождения всех этих шагов, метод index контроллера ImagesController рендерит HTML-файл из вью-файла index.html.erb, исходный код которого приведен ниже:

<p id="notice"><%= notice %></p>

<h1>Images</h1>

<table>

<thead>

<tr>

<th>Title</th>

<th>Photo</th>

<th colspan="3"></th>

</tr>

</thead>

<tbody>

<% @images.each do |image| %>

<tr>

<td><%= image.title %></td>

<td><%= image\_tag(image.photo.url) %></td>

<td><%= link\_to 'Show', image %></td>

<td><%= link\_to 'Edit', edit\_image\_path(image) %></td>

<td><%= link\_to 'Destroy', image, method: :delete, data:

{ confirm: 'Are you sure?' } %>

</td>

</tr>

<% end %>

</tbody>

</table>

Данный файл представляет собой описание разметки HTML-страницы с представлением названия изображения, изображения и кнопок взаимодействия с ним в форме таблицы. С каждым новым изображением таблица автоматически дополняется.