**Реферат**

Пояснительная записка курсового проекта содержит 43 страницы, 25 рисунков, 31 листинга, 5 источников литературы, 4 приложений.

Цель курсовой работы является проектирование и реализация кроссплатформенного серверного приложения «Менеджер ссылок» на базе REST API архитектуры.

Пояснительная записка состоит из введения, шести разделов, заключения.

Во введении представлена общая информация, дающая представление о предстоящей работе, определены цели.

В первом разделе рассматривается архитектура базы данных.

Во втором разделе представлен процесс разработки серверного приложения.

В третьем разделе описывается создание и восстановление базы данных из резервной копии.

В четвёртом разделе описывается создание процедур для экспорта и импорта.

В пятом разделе описана технология применения оптимизации запросов.

В шестом разделе представлено руководство пользователя.

В заключении представлены итоги курсового проектирования и задачи, которые были решены в ходе проектирования и разработки базы данных.

Содержание

[Введение 6](#_Toc514094290)

[1. Разработка модели базы данных 7](#_Toc514094291)

[2. Анализ и проектирование архитектуры приложения 10](#_Toc514094292)

[2.1. Модель 10](#_Toc514094293)

[2.2. Сериализация данных 10](#_Toc514094294)

[2.3. Пагинация ответов 12](#_Toc514094295)

[2.4. Привилегии 13](#_Toc514094296)

[2.5. Представления 13](#_Toc514094297)

[2.6. Тестирование сервера 13](#_Toc514094298)

[2.7. Адресация 14](#_Toc514094299)

[3. WEB приложение. 15](#_Toc514094300)

[3.1. Краткий обзор. 15](#_Toc514094301)

[4. Деплой на удалённый сервер 16](#_Toc514094302)

[4.1. Процедура экспорта 16](#_Toc514094303)

[4.2. Процедура импорта 16](#_Toc514094304)

[Заключение 18](#_Toc514094305)

[Список использованных источников 19](#_Toc514094306)

[ПРИЛОЖЕНИЕ А 20](#_Toc514094307)

[ПРИЛОЖЕНИЕ Б 21](#_Toc514094308)

# Введение

Цель работы изучить и закрепить навыки написания кроссплатформенных распределённых приложений на базе архитектуры REST API. Использовать технологию ORM для работы с базой данных. Научиться автоматизировать тестирование и обновления и разворачивании приложения на удалённом сервере. Овладеть современными средствами создания документации на базе написанного кода. Закрепить навыки написания web-приложений.

В соответствии с заданием курсового проектирования следует разработать кроссплатформенное серверное приложение для управления ссылками на базе REST API архитектуры. Так же должно присутствовать создать клиентское WEB приложение, представляющее собой интерфейс для менеджмента ссылок. Для работы курсового проекта необходим интерпретатор Python 3.6 (Cython) и библиотеки, указанные в приложении А.

# Разработка модели базы данных

Для создания, хранения и управления данными, была использованная реляционная СУБД Postgresql 5.3. Ниже на рисунке 1.1 представлена модель базы данных (таблицы, отмеченные на них первичных и вторичных ключей) курсового проекта. База данных была подвергнута процедуре нормализации данных.

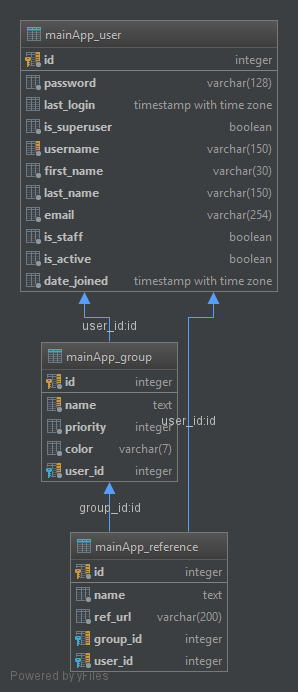


Рисунок 1.1 – Диаграмма модели базы данных

База данных на выше представленной схеме в полной мере описывает необходимые таблицы для решения поставленной задачи данного курсового проекта. Модель базы данных описывается с помощью технологии ORM. Ниже представлены листинги кода, который создаёт её. UML схему можно найти на электронном носителе.

class Group(models.Model):

name = models.TextField()

priority = models.IntegerField()

color = models.CharField(max\_length=7)

user = models.ForeignKey(

'User',

related\_name='groups\_ref',

on\_delete=models.CASCADE

)

class Meta:

unique\_together = (('id', 'name'),)

ordering = ('priority',)

Листинг 1.1 – Код для описания таблицы групп

Выше на листинге 1.1 описана таблица групп ссылок, которая имеет 5 полей:

* столбец id (int, increment (1, 1)) содержит уникальный идентификатор;
* столбец name (text) хранит в себе имя группы;
* столбец color (char (7)) хранит текстовое представление цвета;
* столбец priority (integer) хранит приоритет в группы;
* столбец user (int) хранит внешний ключ на запись в таблице пользователя.

class Reference(models.Model):

name = models.TextField()

ref\_url = models.URLField()

group = models.ForeignKey(

'Group',

related\_name='groups',

on\_delete=models.CASCADE,

blank=True,

null=True

)

user = models.ForeignKey(

'User',

related\_name='references',

on\_delete=models.CASCADE

)

class Meta:

ordering = ('id',)

Листинг 1.2 – Код для описания таблицы ссылок

Выше на листинге 1.2 описана таблица групп ссылок, которая имеет 5 полей:

* столбец id (int, increment (1, 1)) содержит уникальный идентификатор;
* столбец name (text) хранит в себе имя ссылки;
* столбец ref\_url (URL) хранит ссылку;
* столбец user (int) хранит внешний ключ на запись в таблице пользователя
* столбец group (int) хранит внешний ключ на запись в таблице групп.

Класс, описывающий таблицу пользователей, наследуется от класса AbstractUser фреймворка Django 2.0. И в нем представлены следующие столбцы:

* столбец id (int, increment (1, 1)) содержит уникальный идентификатор;
* столбец password (varchar (128)) содержит пароль пользователя, который храниться в виде хэша, полученного по алгоритму sha-256;
* столбец username (char (150)) содержит никнейм пользователя;
* столбец first\_name (char (30)) содержит имя пользователя;
* столбец last\_name (char (150)) содержит фамилию пользователя;
* столбец email (email) содержит email пользователя;
* столбец is\_active (boolean) содержит поле отвечающая был ли активирован пользователь;
* столбец date\_joined (dataTime) содержит дату и время последней авторизации пользователя.

# Анализ и проектирование архитектуры приложения

## 2.1. Анализ и краткие сведения о сервере

Серверное приложение согласно заданию курсового проекта должно реализовываться на REST API архитектуре. REST — архитектурный стиль взаимодействия компонентов распределённого приложения в сети. REST представляет собой согласованный набор ограничений, учитываемых при проектировании распределённой гипермедиа-системы. В REST используются различные методы запроса для различных действий. Ниже представлены типы запросов и их функции:

* GET (чтение списка или одного объекта, для него может быть использовано кэширование);
* OPTION (получение списка информации об объекте (только чтение, его тип, обязательно ли указывать и т.д.));
* POST (создание элемента таблицы);
* PUT (изменение элемента таблицы);
* PATCH (частичное изменение элемента таблицы);
* DELETE (удаление элемента таблицы).

Для реализации сервера использовался фреймворк Django 2.0 для технологии python 3.6 (cython). Так же использовался такая библиотека, как Django-rest-framework.

После описания модели данных можно начать проектирование архитектуры запросов сервера. Имеется 3 таблицы данных, 2 из которых связаны с таблицей пользователя, следовательно, они будут завесить от таблицы пользователя. Ниже представлено API (относительные url адреса и их методы).

* /api/users/ (GET, OPTION);
* /api/users/<username>/ (GET, POST, PUT, PATCH, OPTION);
* /api/users/<username>/login (POST);
* /api/users/<username>/change-password (POST);
* /api/users/<username>/groups/ (GET, OPTION);
* /api/users/<username>/groups/<group\_id>/ (GET, POST, PUT, PATCH, OPTION);
* /api/users/<username>/references/ (GET, OPTION);
* /api/users/<username>/references/<references\_id>/ (GET, POST, PUT, PATCH, OPTION);

Приложение содержит несколько слоёв, которые будут описаны ниже.

## 2.2. Модель данных

База данных содержит 3 таблицы, которые подробно описаны в главе 1. В коде они описываются классами. Этот слой предоставляет функционал для работы с базой данной (выборка, вставка, удаление строк из таблиц).

## 2.3. Сериализация данных

Сериализаторы — особые классы, отвечающие за то, какие поля из сущности (строка из таблицы) будут отображаться пользователю при получении ответа сервера и какие необходимо указать чтобы создать объект (изменить) объект.

Ниже представлены листинги классов сериализаторов для всех сущностей курсовой работы.

class UserSerializer(serializers.HyperlinkedModelSerializer):

url = serializers.HyperlinkedIdentityField(

view\_name='user-detail',

lookup\_field='username'

)

class Meta:

model = User

fields = (

'url',

'username',

'first\_name',

'last\_name',

'email',

)

Листинг 2.1. Сериализатор пользователя

Сериализатор класс пользователя позволяет пользователю API увидеть следующие поля:

* никнейм пользователя (поле username);
* имя пользователя (поле first\_name);
* фамилия пользователя (поле last\_name);
* электронная почта пользователя (поле email).

Пользователь не сможет получить пароль или последнее время входа, так как это информация ему предоставляется не должна. Так же он пользователь содержит HATEOAS на самого себя.

class GroupSerializer(serializers.HyperlinkedModelSerializer):

user = serializers.HyperlinkedRelatedField(

view\_name='user-detail',

read\_only=True,

lookup\_field='username'

)

class Meta:

model = Group

fields = (

'id',

'name',

'priority',

'color',

'user'

)

Листинг 2.2. Сериализатор группы

Сериализатор класс группы позволяет пользователю API увидеть следующие поля:

* уникальный идентификатор группы (поле id);
* имя группы (поле name);
* приоритет группы при выводе (поле priority);
* цвет группы (поле color);
* HATEOAS ссылка на пользователя, которому принадлежит группа (поле user).

class ReferenceSerializer(serializers.HyperlinkedModelSerializer):

user = serializers.HyperlinkedRelatedField(

view\_name='user-detail',

read\_only=True,

lookup\_field='username')

group = serializers.SlugRelatedField(

slug\_field='id',

read\_only=False,

queryset=Group.objects.filter().all())

class Meta:

model = Reference

fields = (

'id',

'ref\_url',

'name',

'user',

'group')

Листинг 2.3. Сериализатор ссылки

Сериализатор класс группы позволяет пользователю API увидеть следующие поля:

* уникальный идентификатор ссылки (поле id);
* имя ссылки (поле name);
* url ссылки (поле ref\_url);
* id группы (поле group);
* HATEOAS ссылка на пользователя, которому принадлежит ссылка (поле user).

Сериализаторы в некотором смысле являются отображениями.

## 2.4. Пагинация ответов

Все представления, которые описаны в этой главе ниже, используют класс пагинатор. В данном курсовом проекте используется один из пагинаторов реализованных в Django-rest-framework — PageNumberPagination. Максимальное возможное количество элементов на странице — 100. Параметр при GET запросе указывающий количество элементов на странице — «limit», а на номер страницы — «page».

## 2.5. Привилегии

Каждое представление, описанное ниже, использует один или несколько классов, описывающих и контролирующих права доступа к данным, т.е. есть ли у данного пользователя права выполнить запрос. Ниже на листинге 2.4 представлены 2 пользовательских класса привилегий.

class IsUserOwner(permissions.BasePermission):

def has\_object\_permission(self, request, view, obj):

if request.method in permissions.SAFE\_METHODS:

return True

return obj == request.user

class IsOwnerObj(permissions.BasePermission):

def has\_permission(self, request, view):

return request.parser\_context['kwargs']['users\_username'] == request.user.username

def has\_object\_permission(self, request, view, obj):

return obj.user == request.user

Листинг 2.4. Пользовательские классы привилегий

Первый класс проверяет является ли пользователь владельцем записи о нём. Если да, то даёт права на чтение и запись (POST, DELETE, PUT, PATCH методы запроса), иначе только на чтение.

Второй класс проверяет принадлежит ли ссылка или группа пользователю. Если не принадлежит запрещает все операции чтения и записи этих сущностей.

Так же все классы представлений используют IsAuthenticated привилегию, которая проверяет был ли авторизован пользователь.

## 2.6. Представления

Все представления представляют собой классы, наследованные от класса ModelViewSet фреймворка Django-rest-api. Представления реализуют функции работы над сущностью: создание, удаление, изменение, частичное изменение, чтение (CRUD операции). Листинги с кодом реализующие преставления находятся в приложении Б.

Представление пользователя содержит два дополнительных роута для авторизации и смены пароля пользователя.

## 2.7. Тестирование сервера

Тестирование проводится автоматически при загрузке и установке приложения сервера на Heroku. При тестировании создаётся зеркальная тестовая копия базы данных, заполняется данными, которая будет удалена по окончании тестирования. Все тестовые классы находятся в модуле tests сервера приложения mainApp. Все классы тестирования наследуются от класса TestCase. Необходимо переопределить метод setUp, который заполняет данные при инициализации каждого теста.

Тестирование полностью симулирует работу клиента, его запросы, работу с базой данных. Ниже приведены некоторые листинги тестов.

def setUp(self):

self.factory = APIRequestFactory()

self.user = User.objects.create\_user(

username='banifest', email='banifest@gmail.com', password='adminadmin')

self.group = Group.objects.create(color='GREEN', priority=1, name='test', user=self.user)

self.ref = Reference.objects.create(name='GREEN', ref\_url='http://123.com', group=self.group, user=self.user)

self.client = APIClient()

self.client.force\_login(user=self.user)

Листинг 2.5. Инициализация тестов

На листинге 2.5 можно увидеть инициализации тестов связанных с работой с ссылками. Здесь создаётся пользователь, клиент, логиниться клиент, создаётся группа и ссылка, привязанная к ней.

def test\_detail\_ref\_update\_auth(self):

response = self.client.patch(

'/api/users/banifest/references/{0}/'.format(self.ref.id),

{

'name': 'NotTest',

'ref\_url': 'http://Not-test.com',

}, format='json')

self.assertEqual(response.status\_code, 200)

self.assertEqual(response.data['name'], 'NotTest')

self.assertEqual(response.data['ref\_url'], 'http://Not-test.com')

Листинг 2.6. Тестирование частичного изменения ссылки

На листинге выше представлено тестирование частичного изменения ссылки.

## Адресация

За адресацию api в приложении отвечает роутеры, обеспечивающие правильное перенаправление запросов. Их код представлен на листинге ниже.

router = DefaultRouter()

router.register(r'users', views.UserViewSet)

group\_router = NestedSimpleRouter(router, r'users', lookup='users')

group\_router.register(r'groups', views.GroupViewSet, base\_name='groups')

ref\_router = NestedSimpleRouter(router, r'users', lookup='users')

ref\_router.register(r'references', views.ReferenceViewSet, base\_name='references')

urlpatterns = [

url(r'^swagger(?P<format>\.json|\.yaml)$', schema\_view.without\_ui(cache\_timeout=None), name='schema-json'),

url(r'^swagger/$', schema\_view.with\_ui('swagger', cache\_timeout=None), name='schema-swagger-ui'),

url(r'^redoc/$', schema\_view.with\_ui('redoc', cache\_timeout=None), name='schema-redoc'),

url(r'^api-auth/', include('rest\_framework.urls')),

url(r'^api/', include(router.urls)),

url(r'^api/', include(group\_router.urls)),

url(r'^api/', include(ref\_router.urls)),

]

Листинг 2.7. Адресация на сервере

Как за адресацию отвечает список urlpatterns, который содержит регулярное выражение, по которому сопоставляются url запросов, и отображения, которые соответствуют регулярным выражениям.

## Документация

Приложение поддерживает два вида документации, которая генерируется на основе реализованных представлений. Примеры можно увидеть на рисунках ниже.

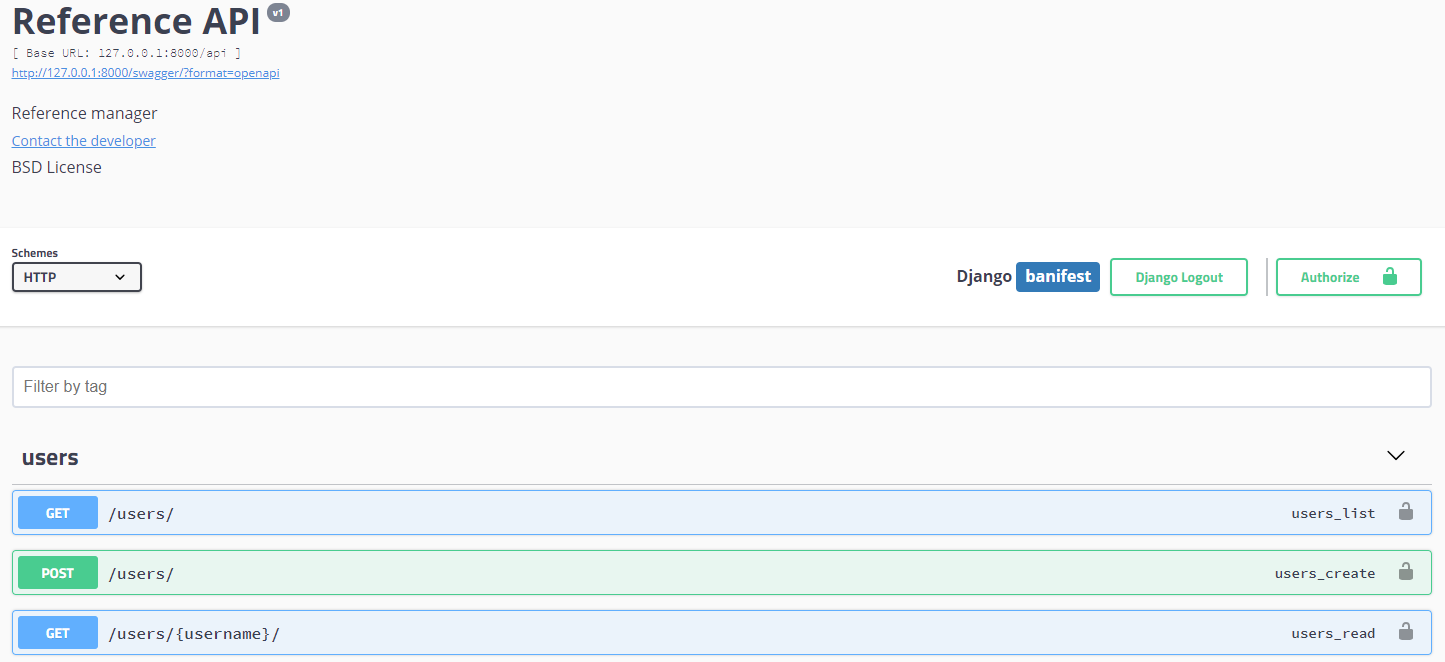


Рисунок 2.1. Часть Swagger документации

Выше представлен пример swagger документации данной курсовой работы.

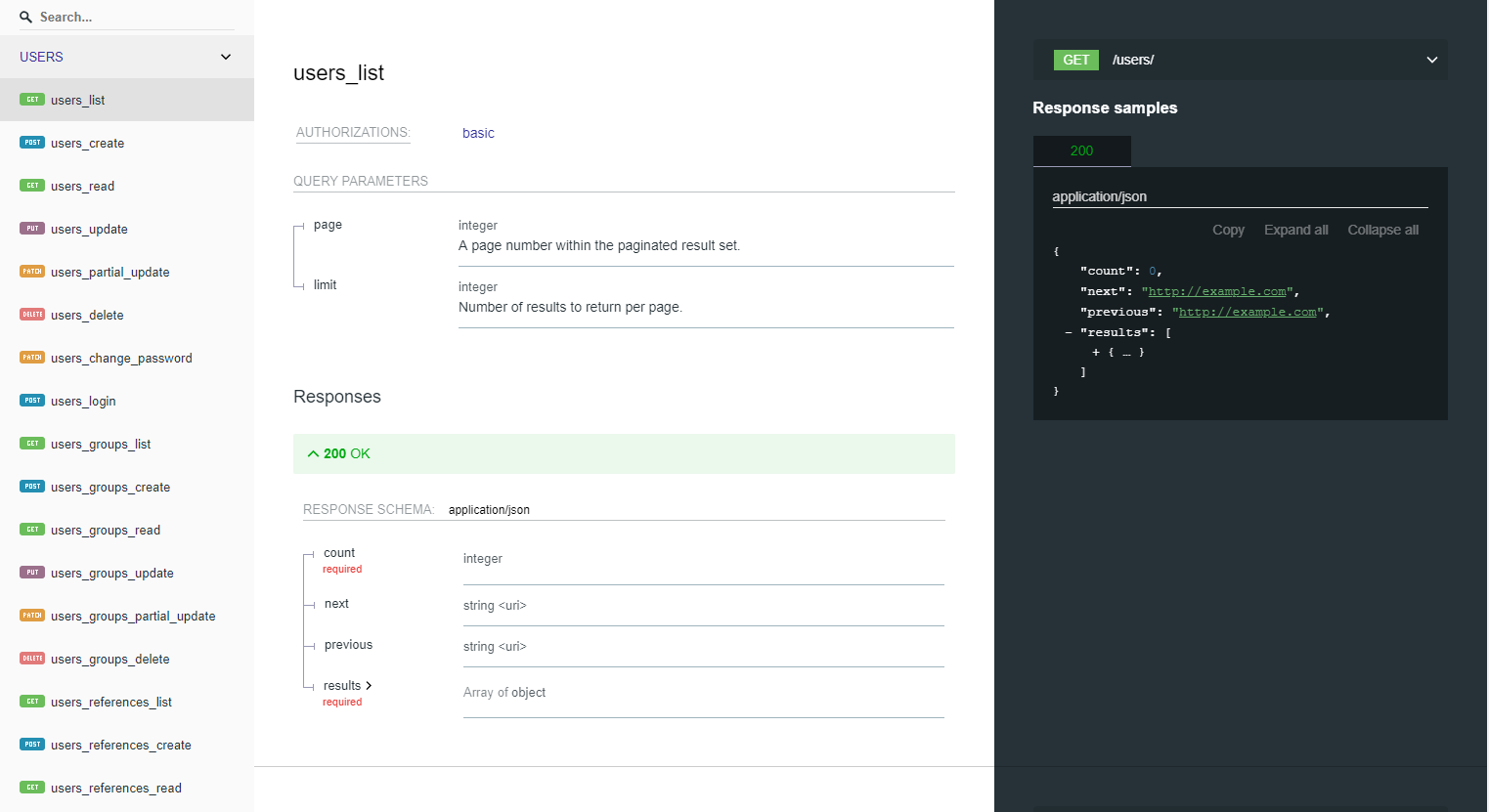


Рисунок 2.2. Часть Redoc документации

Выше представлен пример redoc документации данной курсовой работы.

## Аутентификация

В данной курсовой работе используется basic способ аутентификации, пароли хранятся на сервере в закешированном виде (тип кэширования sha-256), а при передачи используется протокол STL (https). Пароли кэшируются непосредственно на сервере.

Аутентификация происходит следующим образом:

* клиент присылает на url “/api/users/<username>/login/” POST запрос содержащий пароль пользователя, указанного в адресе;
* если у пользователя есть cookies sessionid и csrftoken, то они удаляются;
* если пароль не совпал, то присылаем сообщение о невозможности авторизоваться;
* если проверка пароля прошла успешно, сервер возвращает ответ с хедерами set-cookie содержащий sessionid и csrftoken со временем жизни до конца сессии;
* при всех последующих запросах клиент может отдавать cookies для подтверждения авторизации.

# 3. WEB приложение.

## 3.1. Краткий обзор.

WEB-приложение написано на языке python 3.6 с использованием фреймворка django 2.0. Содержит 4 станицы: авторизации, регистрации, главной (на которой и доступно взаимодействие с пользователем), настроек пользователя. Для рендеринга страниц используется шаблонизатор jinja 2. Приложение для получения и манипуляции данных использует сервер, описанный в предыдущей главе. Web-приложение использует представления для получения и обработки данных.

На каждой странице присутствует хедер и футер. Первый содержит кнопки доступные пользователю, которые меняются в зависимости от страницы и статуса авторизации пользователя, так же в футере подключаются стили и js библиотеки, использующиеся глобально и сообщение об ошибке (если есть). Футер содержит информацию о сайте.

## 3.2. Страница аутентификации

Ниже представлен скриншот страницы аутентификации.

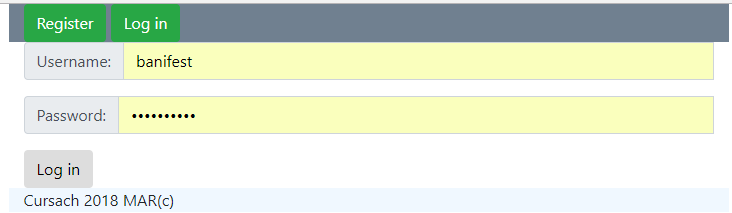


Рисунок 3.1. Страница аутентификации

Взаимодействие на всех страницах и этой, в частности, осуществляется следующим образом:

* на странице находятся html формы которые отправляют post запросы на эту же страницу;
* сервер web-приложения из параметров запроса определяет его тип и отправляет на сервер с api;
* дождавшись ответа сервер решает, что делать дальше, если произошла ошибка авторизации, то происходит автоматический log out;
* если ответ положительный, то формирует ответ и возвращает его клиенту.

Если аутентификация прошла успешно, то происходит переадресация на главную страницу. После аутентификации всегда можно разлогиниться удалив cookies или нажать кнопку “Log out”.

## 3.3. Главная страница

Ниже на рисунке 3.2 представлен пример главной страницы.



Рисунок 3.2. Главная страница

На главной странице представлен основной функционал приложения. Так как это менеджер ссылок, то здесь можно просмотреть сохранённые ссылки, создать новую в какой-либо группе, изменить или удалить. Настроить цвет фона отображения группы в приложении. Так же реализован функционал удаления из группы. Так же присутствует возможность скачать json файл с группами, которые содержат ссылки.

## 3.4. Страница настроек аккаунта пользователя

Ниже на рисунке 3.3 представлен пример страницы настроек пользователя.

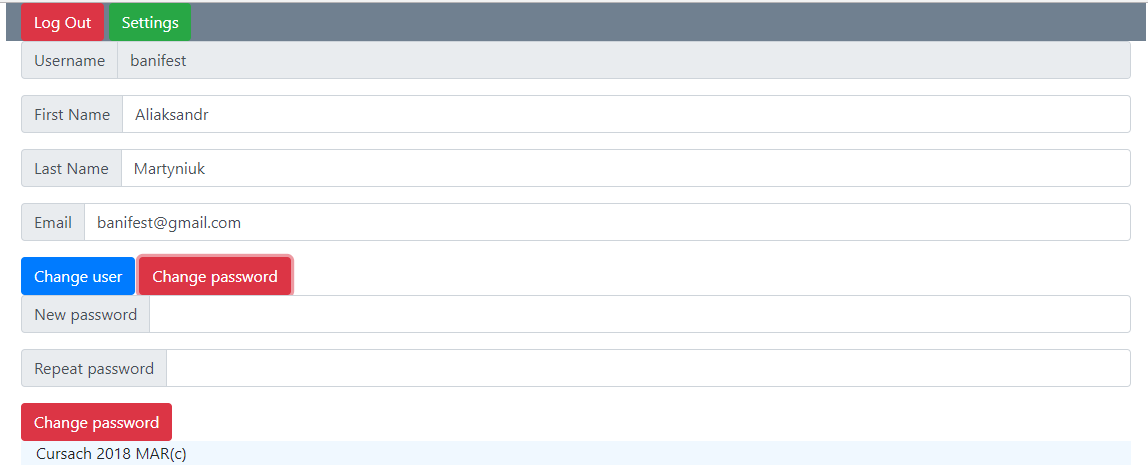


Рисунок 3.3. Страница настроек пользователя

На странице пользователя ему предоставляются полномочия изменить имя, фамилию, электронную почту, или пароль.

## 3.5. Страница регистрации пользователя

Ниже представлен пример страницы регистрации на рисунке 3.4.



Рисунок 3.4. Страница регистрации пользователя

Данная страница предназначена для регистрации новых пользователей. С чем успешно справляется.

# 4. Загрузка и запуск приложений на удалённом сервере

## 4.1. Система контроля версий

Для написания сервера с api, и web-приложения использовалась система контроля версий GitHub. Репозиторий api сервера можно найти по следующей ссылке <https://github.com/Banifest/PSCPACourseWork>. Репозиторий web-приложения можно найти по следующей ссылке <https://github.com/Banifest/WebUICourseWork>.

## 4.2. Удалённый сервер

Два приложения разворачивались на облачной платформе Heroku. Веб приложение не содержит никаких аддонов в отличии от api сервера, который имеет postgresql аддон. Для деплоя на языке python необходимо создать файл requiments.txt содержащий набор необходимых библиотек и их версии. А также файл «Prockfile», содержащий набор команд, которые должны выполниться после деплоя на облачную платформу. Web-приложение будет отправляться на сервер после каждого push’а на github. Api сервер будет должен пройти тестирование для последующего деплоя на heroku, этот процесс описан ниже.

## 4.3. Тестирование api сервера

Тестирование происходит на Travis CI. Travis CI — распределённый веб-сервис для сборки и тестирования программного обеспечения, использующий GitHub в качестве хостинга исходного кода. Чтобы travis ci корректно определил нужный набор действий необходимо написать конфигурационный файл с именем «.travis.yml», который приведён на листинге 4.1 ниже.

sudo: true

language: python

python:

- "3.6"

services:

- postgresql

install:

- pip install -r requirements.txt

before\_script:

- psql -c 'create database TestPostgres;' -U postgres

script:

- python manage.py makemigrations

- python manage.py migrate

- python manage.py test --debug-sql

deploy:

provider: heroku

api\_key: 7b182a2c-0bed-443c-9395-2069d64217f7

app: pscaserv

Листинг 4.1. Yml код настроек деплоя на travis ci

Пояснение вышеприведённого листинга. Вначале указывается язык, который будет использоваться на сервере в инструкции language. Затем версия интерпретатора языка в инструкции python. Потом устанавливаются необходимые библиотеки из файла «requirements.txt». В инструкции before\_script создаётся тестовая база данных. Потом указываются скрипты в порядке их выполнения. И последний шаг — указание инструкций по деплою на удалённый сервер.

# Заключение

В данном курсовом проекте была разработана база данных для учреждений менеджера ссылок. Построено REST API приложение на основе базы данных. Разработано web-приложение, предоставляющее простой для работы с приложением интерфейс. Настроен автоматический деплой на удалённый сервер с предварительным тестированием

Выполненные цели:

* хранение ссылок;
* группировка по различным критериям или выдача меток;
* экспорт ссылок в формат json;
* настройка цвета отображение групп.

# Список использованных источников

1. Оптимизация запросов производительности [Электронный ресурс] msdn.microsoft.com – Режим доступа: <https://technet.microsoft.com/ru>-ru/library/ms172984(v=sql.110).aspx Дата доступа 07.12.2017.

2. Benjamin Nevarez. Inside the SQL Server Query Optimizer / – First published by Simple Talk Publishing 2010 – 252 с.

3. Оптимизация SQL запросов [Электронный ресурс] <http://ts-soft.ru/blog/sql-optimization-1>. Дата доступа 07.12.2017.

4. Оптимизация SQL запросов [Электронный ресурс] [http://ts-soft.ru/blog/sql-optimization-2. Дата доступа 07.12.2017](http://ts-soft.ru/blog/sql-optimization-2.%20Дата%20доступа%2007.12.2017).

5. Разностные резервные копии (SQL Server) [Электронный ресурс] <https://docs.microsoft.com/ru-ru/sql/relational-databases/backup-restore/differential-backups-sql-server>. Дата доступа 08.12.2017.

# ПРИЛОЖЕНИЕ А

**Необходимые библиотеки и их версии для работы сервера с API**

certifi==2018.1.18

chardet==3.0.4

click==6.7

coreapi==2.3.3

coreschema==0.0.4

coverage==4.5.1

dj-database-url==0.5.0

Django==2.0

django-filter==1.1.0

django-heroku==0.3.1

django-rest-swagger==2.1.2

djangorestframework==3.8.2

drf-extensions==0.3.1

drf-nested-routers==0.90.2

drf-yasg==1.7.1

flex==6.13.1

future==0.16.0

gunicorn==19.8.1

idna==2.6

inflection==0.3.1

itypes==1.1.0

Jinja2==2.10

jsonpointer==1.14

jsonschema==2.6.0

Markdown==2.6.11

MarkupSafe==1.0

openapi-codec==1.3.2

permission==0.4.1

psycopg2==2.7.4

Pygments==2.2.0

pytz==2018.4

PyYAML==3.12

requests==2.18.4

rfc3987==1.3.7

ruamel.yaml==0.15.37

simplejson==3.13.2

six==1.11.0

strict-rfc3339==0.7

swagger-spec-validator==2.1.0

uritemplate==3.0.0

urllib3==1.22

validate-email==1.3

whitenoise==3.3.1

# ПРИЛОЖЕНИЕ Б

**Частичные листинги приложения**

Листинги представлений.

class UserViewSet(viewsets.ModelViewSet):

queryset = User.objects.all()

serializer\_class = UserSerializer

lookup\_field = 'username'

permission\_classes = (

permissions.IsAuthenticated,

IsUserOwner,

)

def get\_permissions(self):

if self.action == 'create':

permission\_classes = [permissions.AllowAny]

elif self.action == 'login':

permission\_classes = [permissions.AllowAny]

else:

permission\_classes = [

IsUserOwner,

permissions.IsAuthenticated,

]

return [permission() for permission in permission\_classes]

@action(methods=['POST'], detail=True, url\_path='login')

def login(self, request: HttpRequest, username=None):

user = authenticate(username=username, password=request.data['password'])

if user is not None:

if user.is\_active:

login(request, user)

return HttpResponse(

content=json.dumps({'status': 'success'}),

status=201,

content\_type='application/json'

)

else:

return HttpResponse(

json.dumps({'detail': "don't right login or password"}),

status=401,

content\_type='application/json'

)

else:

return HttpResponse(

json.dumps({'detail': "don't right login or password"}),

status=401,

content\_type='application/json'

)

@action(methods=['PATCH'], detail=True, url\_path='change-password')

def change\_password(self, request: HttpRequest, username=None):

user = User.objects.filter(username=username).first()

if user == request.user:

user.set\_password(self.request.data['password'])

user.save()

return Response('{"detail": "password change successful"}')

else:

res = Response('{"detail": "bad request"}')

res.status\_code = 400

return res

def retrieve(self, request, \*args, \*\*kwargs):

pk = kwargs['username']

queryset = User.objects.filter(username=pk)

instance = get\_object\_or\_404(queryset, username=pk)

serializer = self.get\_serializer(instance)

return Response(serializer.data)

def perform\_create(self, serializer):

serializer.save()

user = User.objects.filter(username=self.request.data['username']).first()

user.set\_password(self.request.data['password'])

user.save()

Листинг Б.1. Пользовательский класс представлений

class GroupViewSet(viewsets.ModelViewSet):

permission\_classes = (

IsOwnerObj,

permissions.IsAuthenticated,

)

queryset = Group.objects.all()

serializer\_class = GroupSerializer

def retrieve(self, request, \*args, \*\*kwargs):

pk = kwargs['pk']

users\_pk = kwargs['users\_username']

user = User.objects.get(username=users\_pk)

queryset = Group.objects.filter(pk=pk, user=user)

instance = get\_object\_or\_404(queryset, pk=pk)

serializer = self.get\_serializer(instance)

return Response(serializer.data)

def list(self, request, \*args, \*\*kwargs):

users\_pk = kwargs['users\_username']

user = User.objects.get(username=users\_pk)

queryset = Group.objects.filter(user=user)

page = self.paginate\_queryset(queryset)

if page is not None:

serializer = self.get\_serializer(page, many=True)

return self.get\_paginated\_response(serializer.data)

else:

serializer = self.get\_serializer(queryset, many=True)

return Response(serializer.data)

def perform\_create(self, serializer):

serializer.save(

user=self.request.user

)

Листинг Б.2. Класс представлений группы

class ReferenceViewSet(viewsets.ModelViewSet):

queryset = Reference.objects.all()

serializer\_class = ReferenceSerializer

permission\_classes = (

IsOwnerObj,

permissions.IsAuthenticated,

)

pagination\_class = CustomPagination

def retrieve(self, request, \*args, \*\*kwargs):

pk = kwargs['pk']

users\_pk = kwargs['users\_username']

user = User.objects.get(username=users\_pk)

queryset = Reference.objects.filter(pk=pk, user=user)

instance = get\_object\_or\_404(queryset, pk=pk)

serializer = self.get\_serializer(instance)

return Response(serializer.data)

def list(self, request, \*args, \*\*kwargs):

users\_pk = kwargs['users\_username']

user = User.objects.get(username=users\_pk)

queryset = Reference.objects.filter(user=user)

page = self.paginate\_queryset(queryset)

if page is not None:

serializer = self.get\_serializer(page, many=True)

return self.get\_paginated\_response(serializer.data)

else:

serializer = self.get\_serializer(queryset, many=True)

return Response(serializer.data)

def perform\_create(self, serializer):

try:

serializer.save(user=self.request.user,

group=Group.objects.get(id=self.request.data['group']))

except:

serializer.save(user=self.request.user,

group=Group.objects.first())

Листинг Б.3. Класс представлений группы