Урок 2

Регистрация через социальную сеть. Django-ORM: связь «один-к-одному»

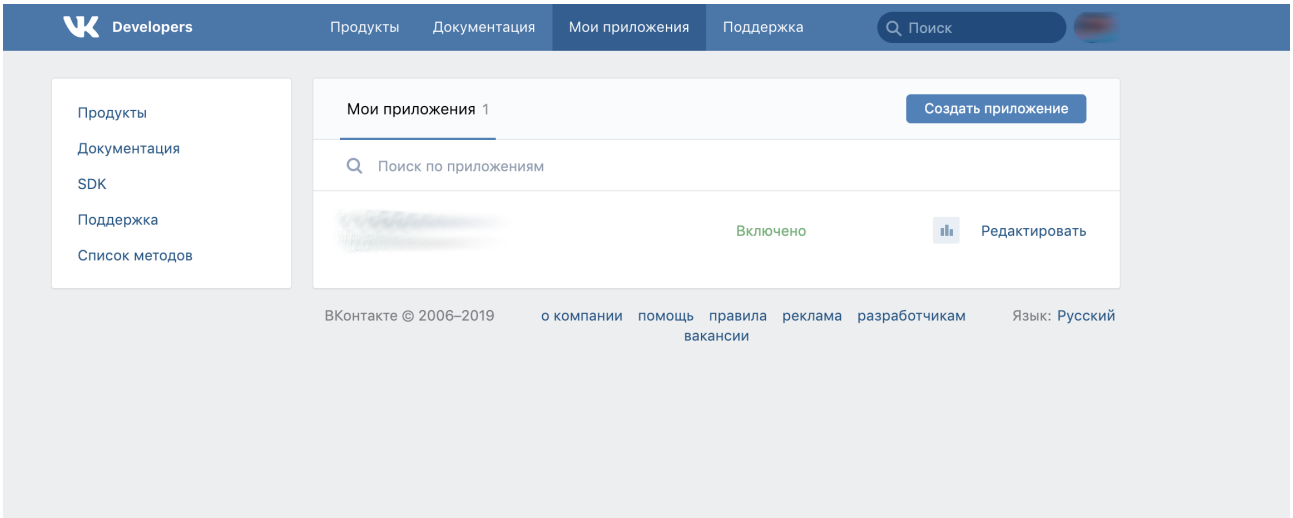
Регистрируемся в магазине через «ВКонтакте». Создаем профиль пользователя и заполняем его данными из социальной сети

Регистрация на сайте через социальную сеть

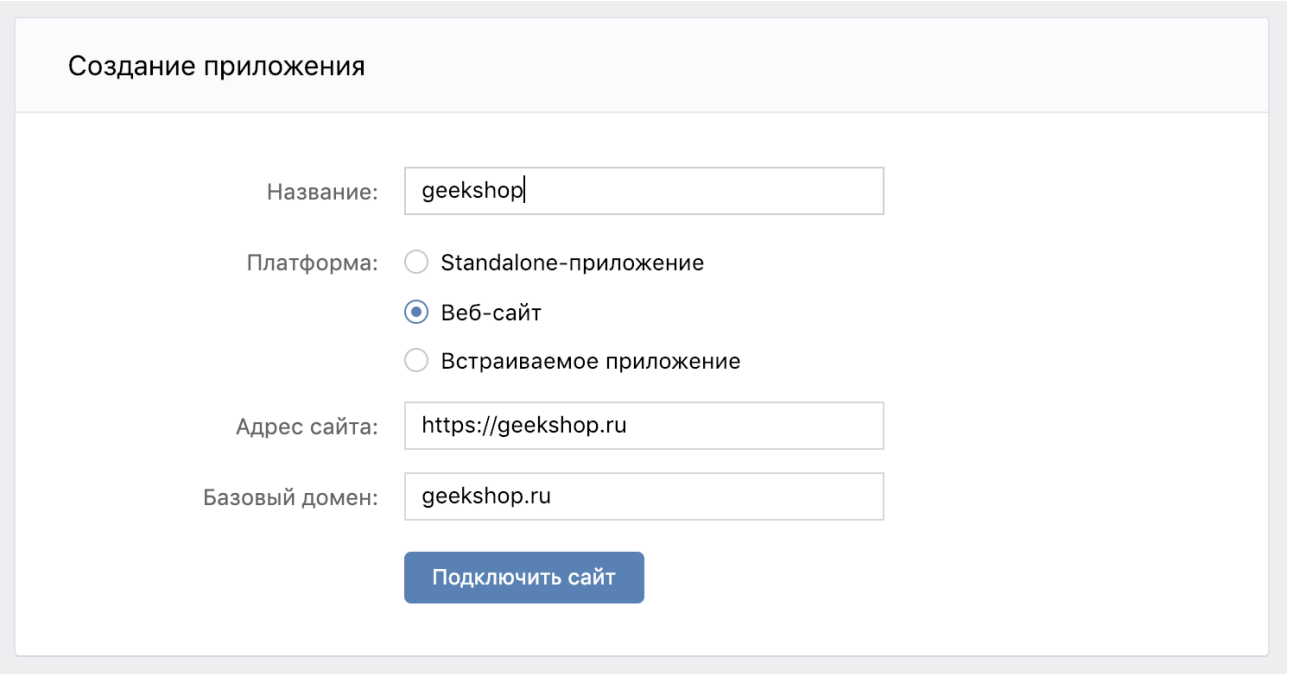
На предыдущем уроке мы расширили функционал регистрации пользователя на сайте, используя подтверждение через электронную почту. Но существует более удобный и быстрый способ аутентификации – через социальную сеть. Рассмотрим его реализацию на примере «ВКонтакте».

«ВКонтакте»: создаем и настраиваем приложение

Для работы нам необходим аккаунт «ВКонтакте». Заходим на свою страницу и в левом меню выбираем пункт «Управление»:

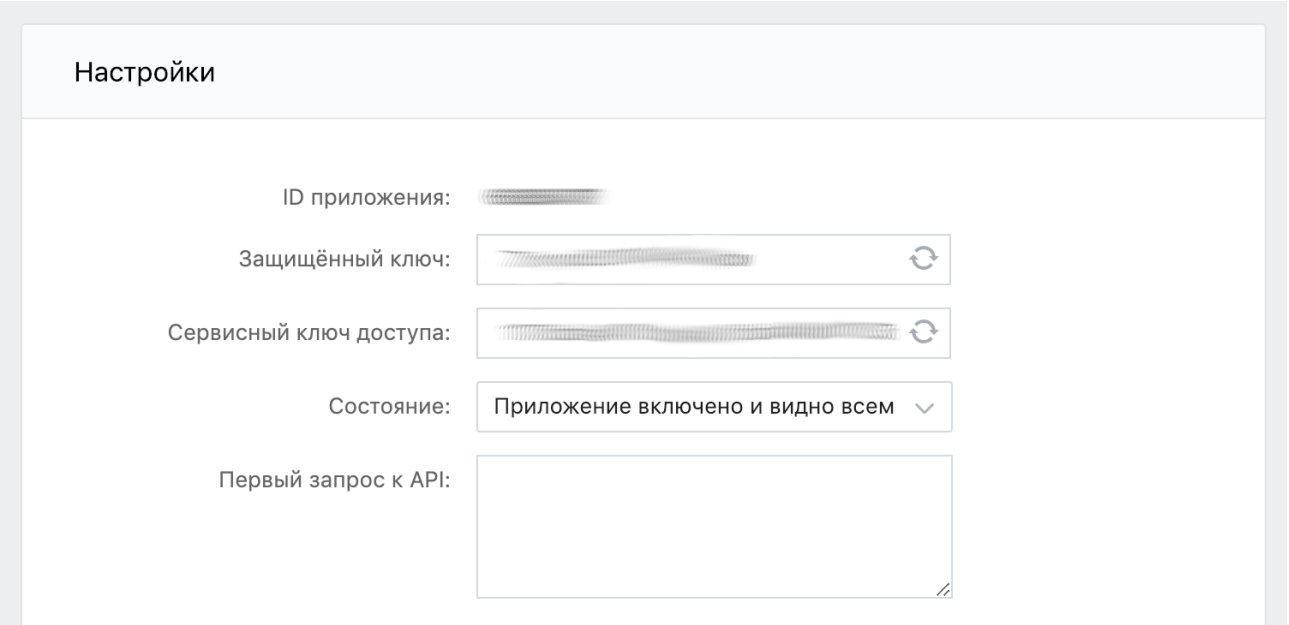


Нажимаем кнопку «Создать приложение». Далее выбираем «Веб-сайт» и вводим данные нашего приложения:

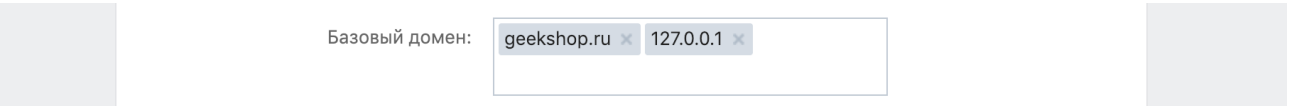


После этого нажимаем «Подключить сайт».

Переходим в настройки созданного приложения:



В дальнейшем нам понадобятся ID приложения и защищенный ключ. Поздравляем! Вы успешно создали приложение «ВКонтакте». Кроме того, для локальной отладки необходимо добавить **127.0.0.1** в список базовых доменов:



Аутентификация через социальные сети в Django при помощи приложения social_django

Процесс аутентификации через социальную сеть сводится к отправке запроса соответствующему сервису (API) и интерпретации ответа. Можно эти действия выполнять как на низком уровне (писать реализацию алгоритмов формирования запроса и анализа ответа), так и на высоком – воспользоваться готовым приложением. Преимущества и недостатки обоих подходов очевидны.

Мы в проекте будем использовать приложение **social_django**. Установка (в Ubuntu pip3):

```
pip install social_auth_app_django
```

Настраиваем конфигурационный файл проекта:

geekshop/settings.py

```
import os, json
...

INSTALLED_APPS = [
    ...
    'social_django',
]
...
AUTHENTICATION_BACKENDS = (
    'django.contrib.auth.backends.ModelBackend',
    'social_core.backends.vk.VKOAuth2',
)

# Загружаем секреты из файла
with open('geekshop/vk.json', 'r') as f:
    VK = json.load(f)

SOCIAL_AUTH_VK_OAUTH2_KEY = VK['SOCIAL_AUTH_VK_OAUTH2_KEY']
SOCIAL_AUTH_VK_OAUTH2_SECRET = VK['SOCIAL_AUTH_VK_OAUTH2_SECRET']
```

В список установленных приложений добавляем **social_django**, создаем константу с кортежем бэкендов аутентификации **AUTHENTICATION_BACKENDS** и прописываем в нее встроенный бэкенд Django и бэкенд VK:

```
'django.contrib.auth.backends.ModelBackend',
'social_core.backends.vk.VKOAuth2'
```

Последняя *обязательная* настройка – создать две константы для ID приложения (**SOCIAL_AUTH_VK_OAUTH2_KEY**) и защищенного (**SOCIAL_AUTH_VK_OAUTH2_SECRET**) клиента, которые мы получили ранее для приложения «ВКонтакте». Из соображений конфиденциальности мы в проекте будем хранить эти данные в текстовом файле в формате JSON:

geekshop/vk.json

```
{
    "SOCIAL_AUTH_VK_OAUTH2_KEY": "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx",
    "SOCIAL_AUTH_VK_OAUTH2_SECRET": "XXXXXXXXXXXXXXXXXXXXXXXX"
}
```

Добавляем константу **SOCIAL_AUTH_URL_NAMESPACE** со значением **social**.

После установки приложения и настройки необходимо выполнить миграции – в БД проекта появятся новые таблицы с названиями вида **social_...** Нас интересует **social_auth_usersocialauth**: в ней после аутентификации пользователя через социальную сеть будет появляться запись. Посмотрите поля этой таблицы и данные в них. Одновременно будет появляться запись в основной таблице пользователей магазина **authapp_shopuser**. Связь между таблицами – через поле **user_id**.

Следующим шагом подключаем диспетчер URL приложения **social_django**:

geekshop/urls.py

```
...
urlpatterns = [
    ...
    path('', include('social_django.urls', namespace='social')),
    ...
]
```

Теперь ответ сервера «ВКонтакте» автоматически будет обрабатываться приложением **social_django**.

Размещаем на странице входа в систему новую ссылку:

geekshop/authapp/templates/authapp/login.html

```
...
<button class="btn btn-round form-control">
    <a href="{% url 'auth:register' %}" class="">
        Зарегистрироваться
    </a>
</button>
<button class="btn btn-round form-control">
    <a href="{% url 'social:begin' 'vk-oauth2' %}?next=/">
     Вход через ВКонтакте
    </a>
</button>
...
```

Используем пространство имен **social**, указанное в диспетчере URL, и адрес **begin** в этом пространстве. В качестве дополнительного аргумента передаем название протокола аутентификации **vk-oauth2**. В конце адреса дописываем **?next=/** для перехода на главную страницу после аутентификации.

Последний штрих: так как ВКонтакте API не передает в ответе возраст пользователя, пропишем в модели **ShopUser** пользователя магазина значение возраста по умолчанию:

```
age = models.PositiveIntegerField(verbose_name = 'возраст', default=18)
```

Для проверки запускаем сервер Django и переходим по ссылке «Вход через ВКонтакте».

После авторизации, вы должны оказаться на главном окне проекта и увидеть свое имя в строке меню, как и в случае обычной аутентификации по логину и паролю. Если этого не произошло – надо искать ошибку.

Обратите внимание, что после аутентификации в основной таблице пользователей магазина **authapp_shopuser** автоматически заполнились поля с логином, паролем, именем, фамилией и email. Входим в систему под уже существующей учетной записью – все должно работать как раньше. Попробуем создать нового пользователя – при верификации по e-mail получим ошибку в консоли:

```
('You have multiple authentication backends configured and therefore must
provide the `backend` argument or set the `backend` attribute on the user.',)
```

Уточним процесс аутентификации пользователя в контроллере **authapp.views.verify**: при вызове метода **auth.login()** явно зададим бэкенд:

```
auth.login(request, user, backend='django.contrib.auth.backends.ModelBackend')
```

Django-ORM: связь «один-к-одному»

Предположим, что в проекте возникла необходимость хранить дополнительные данные пользователя (его пол, ключевые слова, информация о себе и т. д). При этом не хотелось бы менять структуру модели пользователя магазина **ShopUser**. Решением может быть создание еще одной модели (например **ShopUserProfile**) и связывание их один-к-одному.

Authapp: модель ShopUserProfile

Создадим в приложении **authapp** новую модель с профилем пользователя:

authapp/models.py

```
from django.db.models.signals import post_save
from django.dispatch import receiver
...

class ShopUserProfile(models.Model):
    MALE = 'M'
    FEMALE = 'W'

    GENDER_CHOICES = (
        (MALE, 'М'),
        (FEMALE, 'Ж'),
    )

    user = models.OneToOneField(ShopUser, unique=True, null=False,\
                                db_index=True, on_delete=models.CASCADE)
    tagline = models.CharField(verbose_name='теги', max_length=128, \
                               blank=True)
    aboutMe = models.TextField(verbose_name='о себе', max_length=512, \
                               blank=True)
    gender = models.CharField(verbose_name='пол', max_length=1, \
                              choices=GENDER_CHOICES, blank=True)

    @receiver(post_save, sender=ShopUser)
    def create_user_profile(sender, instance, created, **kwargs):
        if created:
            ShopUserProfile.objects.create(user=instance)


    @receiver(post_save, sender=ShopUser)
    def save_user_profile(sender, instance, **kwargs):
        instance.shopuserprofile.save()
```

Для создания связи «один-к-одному» используем поле **models.OneToOneField**.

Значения его аргументов интуитивно понятны. Поясним только **db_index=True**: для данного поля создается индекс. Атрибуты модели **tagline** и **aboutMe** – обычные поля для хранения текстовых данных. Для атрибута **gender** также создаем текстовое поле, но с аргументом **choices=GENDER_CHOICES** – получаем фиксированный набор значений, которые прописаны в кортеже **GENDER_CHOICES**, содержащем кортежи из пар «значение в БД» – «отображаемое значение». Для вывода этого поля будем использовать метод **.get_gender_display()**.

Добавим в модель два метода для создания и сохранения профиля: **create_user_profile** и **save_user_profile**. При работе со связью «один-к-одному» необходим механизм синхронных действий со связанной моделью. Мы используем декоратор **@receiver**, который при получении определенных сигналов вызывает задекорированный метод. В нашем случае сигналом является сохранение (**post_save**) объекта модели **ShopUser** (**sender=ShopUser**).

Мы видим, что из модели **ShopUser** можно получить доступ к связанной модели по ее имени как к атрибуту:

```
instance.shopuserprofile
```

Выполняем миграции и создаем профили для уже существующих пользователей. С этой целью создаем скрипт **update_db.py** по аналогии с **fill_db.py**:

geekshop/mainapp/management/commands/update_db.py

```
from django.core.management.base import BaseCommand
from authapp.models import ShopUser
from authapp.models import ShopUserProfile


class Command(BaseCommand):
    def handle(self, *args, **options):
        users = ShopUser.objects.all()
        for user in users:
            users_profile = ShopUserProfile.objects.create(user=user)
            users_profile.save()
```

В командной строке в корне проекта выполним:

```
python manage.py update_db
```

Попробуем создать нового пользователя – автоматически должен появиться его профиль.

Редактирование профиля пользователя

Пока профили пользователей пустые. Для редактирования создадим форму и выведем ее параллельно с формой редактирования модели пользователя.

Форма на основе класса **forms.ModelForm**:

geekshop/authapp/forms.py

```
...
from .models import ShopUserProfile
...
class ShopUserProfileEditForm(forms.ModelForm):
    class Meta:
        model = ShopUserProfile
        fields = ('tagline', 'aboutMe', 'gender')

    def __init__(self, *args, **kwargs):
        super(ShopUserProfileEditForm, self).__init__(*args, **kwargs)
        for field_name, field in self.fields.items():
            field.widget.attrs['class'] = 'form-control'
```

Добавляем в контроллер:

geekshop/authapp/views.py

```
...
from django.db import transaction
from authapp.forms import ShopUserProfileEditForm
...
@transaction.atomic
def edit(request):
    title = 'редактирование'

    if request.method == 'POST':
        edit_form = ShopUserEditForm(request.POST, request.FILES, \
                                     instance=request.user)
        profile_form = ShopUserProfileEditForm(request.POST, \
                                     instance=request.user.shopuserprofile)
        if edit_form.is_valid() and profile_form.is_valid():
            edit_form.save()
            return HttpResponseRedirect(reverse('auth:edit'))
    else:
        edit_form = ShopUserEditForm(instance=request.user)
        profile_form = ShopUserProfileEditForm(
            instance=request.user.shopuserprofile
        )

    content = {
        'title': title,
        'edit_form': edit_form,
        'profile_form': profile_form
    }

    return render(request, 'authapp/edit.html', content)
```

Особенность работы с формой **ShopUserProfileEditForm** в том, что она заполняется данными из связанной модели:

```
ShopUserProfileEditForm(instance=request.user.shopuserprofile)
```

Обратите внимание, что сохраняем только форму пользователя:

```
edit_form.save()
```

Профиль сохранится автоматически благодаря использованию в модели декоратора **@receiver(post_save, sender=ShopUser)**.

Так как теперь изменения сохраняются в двух моделях, для обеспечения целостности данных применяем к контроллеру декоратор **@transaction.atomic**. Теперь, если произойдет ошибка записи данных в базу внутри контроллера, никакие данные вообще не записываются. Тем самым мы исключаем ситуацию, когда в модели пользователя изменения сохранились, а в модели профиля из-за ошибки – нет.

Не забываем добавить форму в шаблон:

geekshop/authapp/templates/authapp/edit.html

```
{% extends 'authapp/base.html' %}
{% load staticfiles %}

{% block content %}
    <form class="form-horizontal" action="{% url 'auth:edit' %}" method="post"
enctype="multipart/form-data">
        {% csrf_token %}
        {{ edit_form.as_p }}
        {{ profile_form.as_p }}
        <input class="form-control" type="submit" value="сохранить">
    </form>
    <button class="btn btn-round form-control last">
        <a href="{% url 'main' %}" class="">
            на главную
        </a>
    </button>
    <div class="user_avatar"><img src="/media/{{ user.avatar }}"></div>
{% endblock %}
```

Итак, мы расширили функционал проекта – создали профиль пользователя и обеспечили атомарность его редактирования вместе с записью самого пользователя.

*Продвинутая аутентификация пользователя через социальную сеть

В нашем проекте появился профиль пользователя. Возникает вопрос: можем ли мы заполнять его автоматически при аутентификации через социальную сеть? Для реализации этого функционала добавим код в файл настроек проекта:

geekshop/settings.py

```
...
DEBUG = False

#ALLOWED_HOSTS = []
ALLOWED_HOSTS = ['127.0.0.1']
...
MIDDLEWARE = [
          ...
    'social_django.middleware.SocialAuthExceptionMiddleware',
]
...
TEMPLATES = [
    {
                  ...
        'OPTIONS': {
            'context_processors': [
                ...
                'social_django.context_processors.backends',
                'social_django.context_processors.login_redirect',
            ],
        },
    },
]
...
LOGIN_ERROR_URL = '/'
...
SOCIAL_AUTH_VK_OAUTH2_IGNORE_DEFAULT_SCOPE = True
SOCIAL_AUTH_VK_OAUTH2_SCOPE = ['email']


SOCIAL_AUTH_PIPELINE = (
    'social_core.pipeline.social_auth.social_details',
    'social_core.pipeline.social_auth.social_uid',
    'social_core.pipeline.social_auth.auth_allowed',
    'social_core.pipeline.social_auth.social_user',
    'social_core.pipeline.user.create_user',
    'authapp.pipeline.save_user_profile',
    'social_core.pipeline.social_auth.associate_user',
    'social_core.pipeline.social_auth.load_extra_data',
    'social_core.pipeline.user.user_details',
)
```

Для проверки корректности обработки исключений в проекте Django иногда необходимо отключать режим отладки (константа **DEBUG**). При этом список допустимых хостов должен быть заполнен:

```
ALLOWED_HOSTS = ['127.0.0.1']
```

Добавляем в список **MIDDLEWARE** слой обработки исключений приложения **social_django**:

```
'social_django.middleware.SocialAuthExceptionMiddleware',
```

В будущем могут потребоваться контекстные процессоры приложения **social_django**. Добавим их:

```
'social_django.context_processors.backends',
'social_django.context_processors.login_redirect'
```

Константа **LOGIN_ERROR_URL** необходима для корректной переадресации при обработке исключений бэкендами аутентификации.

Отключаем масштаб (**scope**) данных пользователя по умолчанию:

```
SOCIAL_AUTH_VK_OAUTH2_IGNORE_DEFAULT_SCOPE = True
```

Добавляем свой масштаб (**scope**):

```
SOCIAL_AUTH_VK_OAUTH2_SCOPE = ['email']
```

Также мы включаем свой метод **authapp.pipeline.save_user_profile** в конвейер процесса аутентификации пользователя **SOCIAL_AUTH_PIPELINE** (подробнее про SOCIAL_AUTH_PIPELINE). В конвейере прописана последовательность действий. После того как пользователь создан (**social_core.pipeline.user.create_user**), мы обращаемся к ВКонтакте API и получаем дополнительную информацию о пользователе. Для этого нам понадобится библиотека **requests**:

```
pip install requests
```

geekshop/authapp/pipeline.py

```
from collections import OrderedDict
from datetime import datetime
from urllib.parse import urlencode, urlunparse

import requests
from django.utils import timezone
from social_core.exceptions import AuthForbidden

from authapp.models import ShopUserProfile


def save_user_profile(backend, user, response, *args, **kwargs):
    if backend.name != 'vk-oauth2':
        return

    api_url = urlunparse(('https',
                          'api.vk.com',
                          '/method/users.get',
                          None,
                          urlencode(OrderedDict(fields=','.join(('bdate', 'sex',
'about')),

access_token=response['access_token'],
                                                v='5.92')),
```

```
                            None
                            ))

    resp = requests.get(api_url)
    if resp.status_code != 200:
        return

    data = resp.json()['response'][0]
    if data['sex']:
        user.shopuserprofile.gender = ShopUserProfile.MALE if data['sex'] == 2
else ShopUserProfile.FEMALE

    if data['about']:
        user.shopuserprofile.aboutMe = data['about']

    if data['bdate']:
        bdate = datetime.strptime(data['bdate'], '%d.%m.%Y').date()

        age = timezone.now().date().year - bdate.year
        if age < 18:
            user.delete()
            raise AuthForbidden('social_core.backends.vk.VKOAuth2')

    user.save()
```

После авторизации пользователя мы делаем дополнительный запрос к ВКонтакте API, чтобы получить дополнительные данные. Вы можете посмотреть описание API здесь. После получения ответа обрабатываем полученные данные и сохраняем их профиль пользователя. Кроме того, мы осуществляем валидацию на возраст. Если проверка не пройдена – удаляем пользователя, созданного на предыдущем шаге конвейера, и выбрасываем исключение **AuthForbidden** приложения **social_django**. Это исключение будет обработано слоем **social_django.middleware.SocialAuthExceptionMiddleware**, и произойдет переход по адресу, который прописан в константе **LOGIN_ERROR_URL**. Если в настройках проекта включен режим отладки (**DEBUG=True**), исключение *не будет обработано*, и мы увидим отладочную информацию на экране.

Если же проверка по возрасту пройдена – сохраняем модель пользователя (напоминаем, что профиль сохранится автоматически).

Практическое задание

1. Реализовать в проекте простой вариант аутентификации пользователя через социальную сеть «ВКонтакте».
2. Поработать со связью моделей «один-к-одному»: создать профиль пользователя и обеспечить возможность его редактирования.
3. Реализовать автоматическое заполнение профиля пользователя при аутентификации через социальную сеть.
4. Проверить работу исключения **AuthForbidden**, например, задав при проверке минимальный возраст 100 лет.
5. *Получить и сохранить язык и URL-адрес страницы пользователя в социальной сети «ВКонтакте».

Дополнительные материалы

Все то, о чем сказано в методичке, но подробнее:

1. social_django.
2. python-social-auth (GIT).
3. Поля модели.
4. Сигналы в Django.
5. Транзакции в Django.
6. SOCIAL_AUTH_PIPLINE.

Используемая литература

1. Официальная документация.
2. python-social-auth.
3. social-examples: django.

Материал для студентов, которые прошли курс раннее*

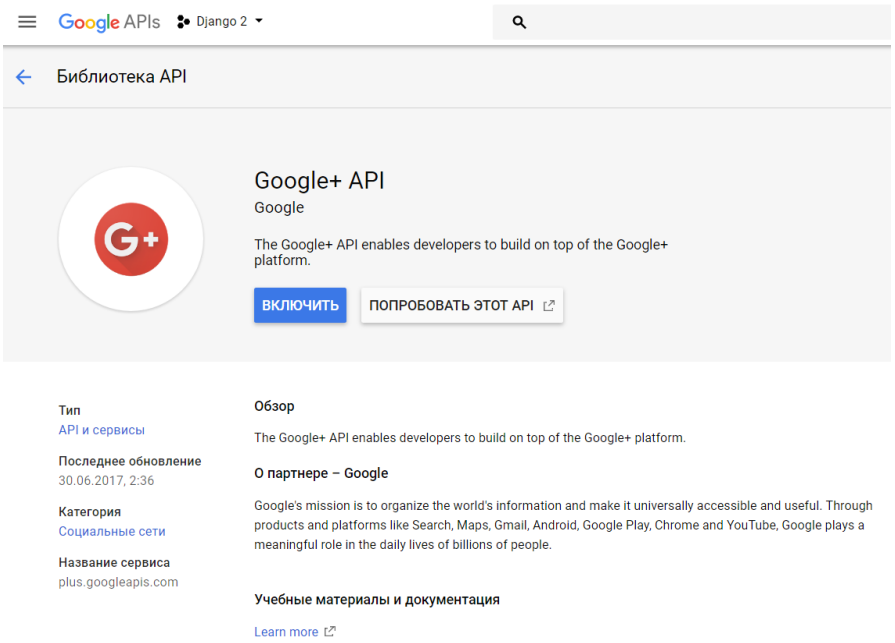
*Данный материал предназначен для студентов, которые уже прошли курс и хотят освежить знания. Новым студентам мы рекомендуем настроить в проекте регистрацию через VK.

Материал ниже не рассматривается на курсе по причине закрытия профилей в социальной сети Google+ после 2 апреля 2019 года.

Регистрация через Google+

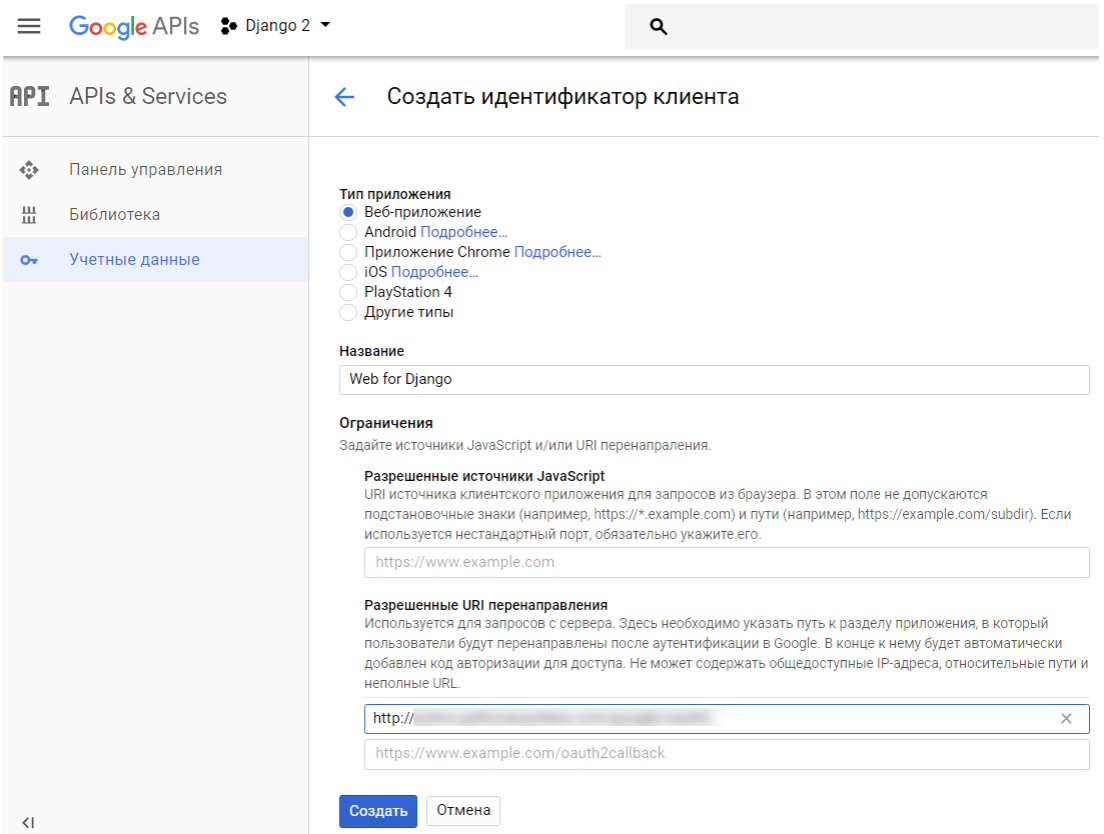
Google+: создаем и настраиваем приложение

Нам необходим аккаунт Google – создаем новую или пользуемся имеющейся почтой @gmail.com. После входа в аккаунт Google находим «Google APIs» и пункт «Библиотека API» (console.developers.google.com), в нем – «Google+ API» и включаем:



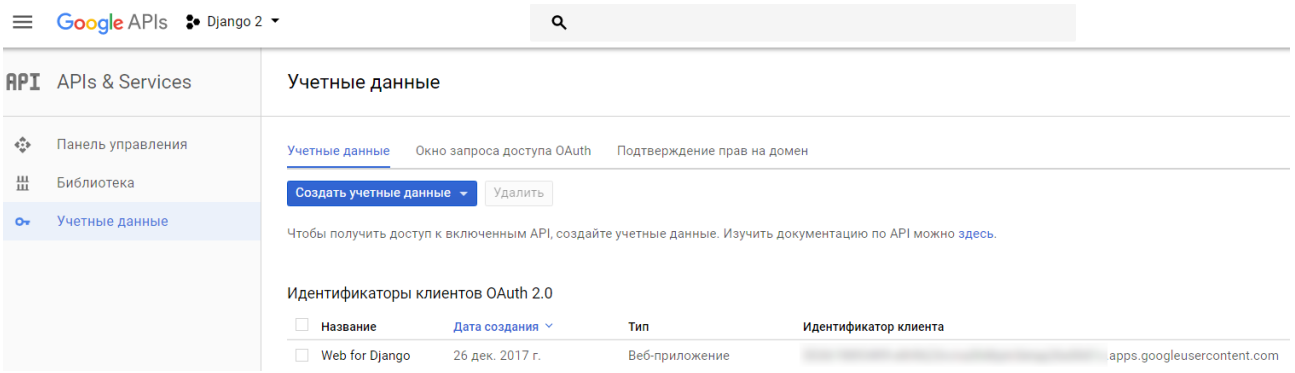
Для дальнейшей работы необходимо настроить идентификатор клиента. Для этого на странице «Google APIs» выбираем пункт «Учетные данные». В окне «Создать идентификатор клиента» выполняем следующие шаги:

- выбираем тип приложения – «Веб-приложение»;
- задаем название идентификатора – «Web for Django»;
- добавляем адрес в «Разрешенные URI перенаправления» (на него пользователь будет перенаправлен после аутентификации Google, для учебного сайта пропишем: http://localhost:8000/auth/verify/google/oauth2/);
- нажимаем кнопку «Создать».

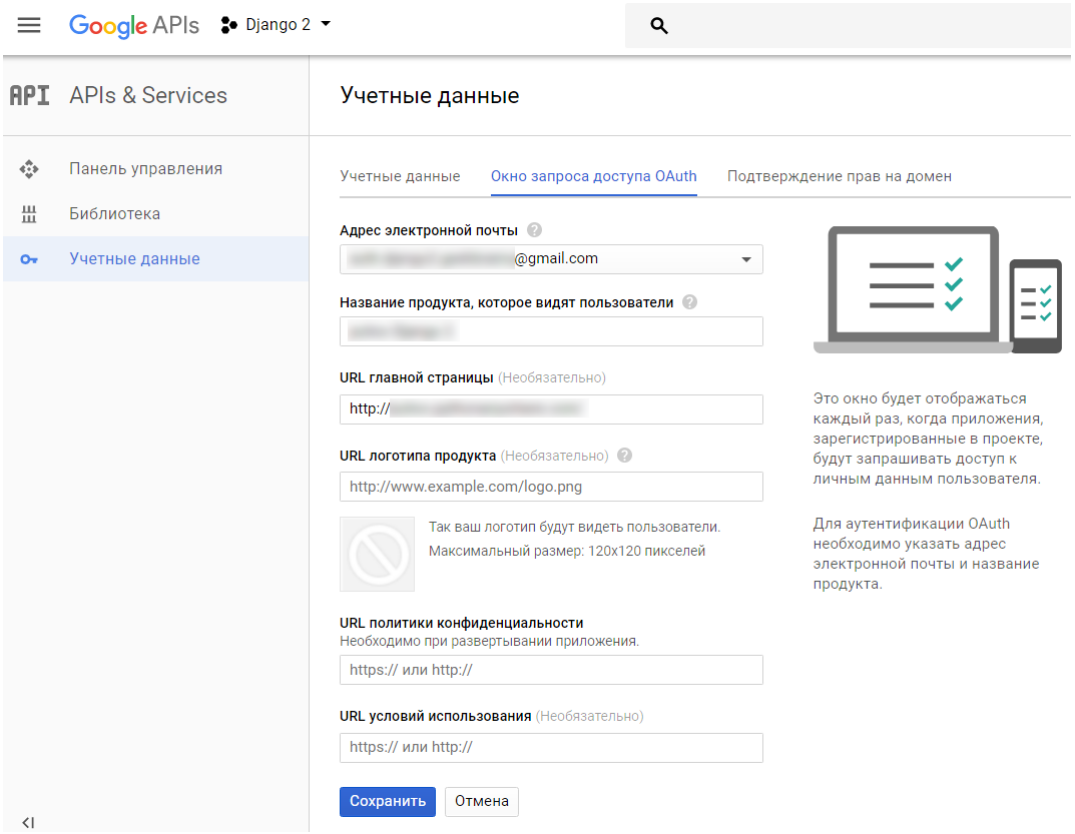


В результате мы получим сгенерированные идентификатор (закрашенная часть, левее имени **.apps.googleusercontent.com**) и секрет клиента:

Их необходимо скопировать в отдельный текстовый файл (они понадобятся позже). В дальнейшем их можно будет посмотреть во вкладке «Учетные данные»:

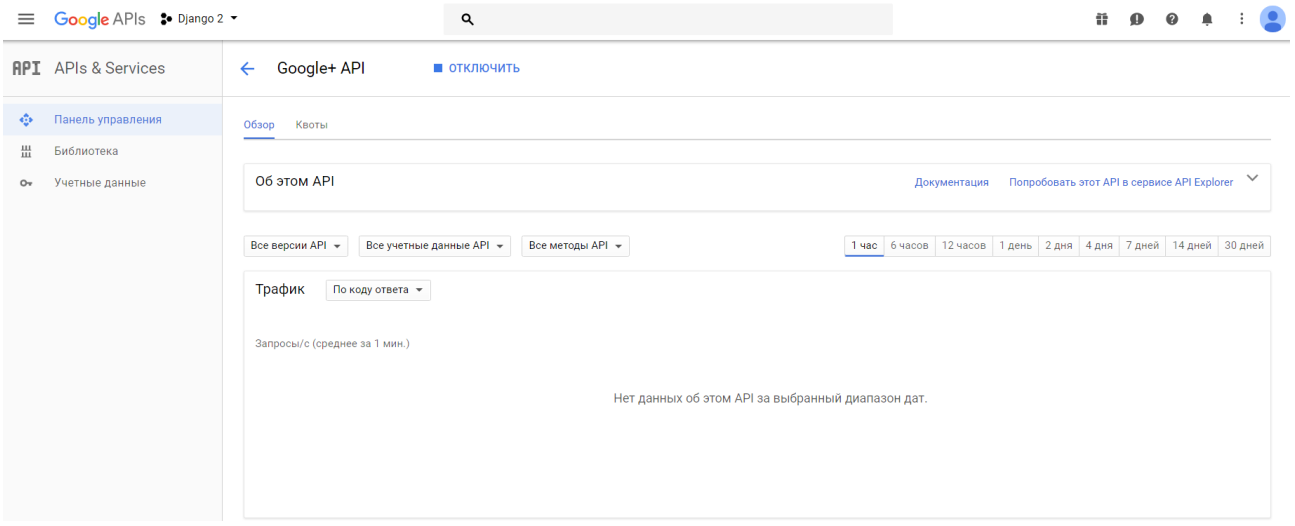


Следующим шагом переходим во вкладку «Окно запроса доступа OAuth» и задаем адрес электронной почты (нашего аккаунта Google) и название ресурса, которое увидят пользователи при попытке аутентификации через Google+ (URL главной страницы можно не вводить).



На этом настройка приложения Google+ завершена.

*Для отладки можно зайти в пункт «Панель управления» – здесь мы видим данные о трафике и ссылку «Попробовать этот API в сервисе API Explorer».



Замечание: рекомендуем попробовать сервис «plus.people.get» для собственного адреса почты @gmail.com – вы увидите, что он может прислать API Google+ в ответ на запрос.

Аутентификация через социальные сети в Django при помощи приложения social_django

Процесс аутентификации через социальную сеть сводится к отправке запроса соответствующему сервису (API) и интерпретации ответа. Можно эти действия выполнять как на низком уровне (писать реализацию алгоритмов формирования запроса и анализа ответа), так и на высоком – воспользоваться готовым приложением. Преимущества и недостатки обоих подходов очевидны.

Мы в проекте будем использовать приложение **social_django**. Установка (в Ubuntu pip3):

```
pip install social_auth_app_django
```

Настраиваем конфигурационный файл проекта:

geekshop/settings.py

```
import os, json
...

INSTALLED_APPS = [
    ...
    'social_django',
]
...
AUTHENTICATION_BACKENDS = (
    'django.contrib.auth.backends.ModelBackend',
    'social_core.backends.google.GoogleOAuth2',
)

SOCIAL_AUTH_URL_NAMESPACE = 'social'

# Можно хранить секреты прямо в файле настроек
# SOCIAL_AUTH_GOOGLE_OAUTH2_KEY = 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx'
# SOCIAL_AUTH_GOOGLE_OAUTH2_SECRET = 'XXXXXXXXXXXXXXXXXXXXXXX'

# Загружаем секреты из файла
with open('geekshop/google+.json', 'r') as f:
    GOOGLE_PLUS = json.load(f)

SOCIAL_AUTH_GOOGLE_OAUTH2_KEY = GOOGLE_PLUS['SOCIAL_AUTH_GOOGLE_OAUTH2_KEY']
SOCIAL_AUTH_GOOGLE_OAUTH2_SECRET =
GOOGLE_PLUS['SOCIAL_AUTH_GOOGLE_OAUTH2_SECRET']
```

В список установленных приложений добавляем **social_django**, создаем константу с кортежем бэкендов аутентификации **AUTHENTICATION_BACKENDS** и прописываем в нее встроенный бэкенд Django и бэкенд Google+:

```
'django.contrib.auth.backends.ModelBackend',
      'social_core.backends.google.GoogleOAuth2'
```

Последняя *обязательная* настройка – создать две константы для идентификатора (**SOCIAL_AUTH_GOOGLE_OAUTH2_KEY**) и секрета (**SOCIAL_AUTH_GOOGLE_OAUTH2_SECRET**) клиента, которые мы получили ранее для приложения Google+. Из соображений конфиденциальности мы в проекте будем хранить эти данные в текстовом файле в формате **JSON**:

geekshop/google+.json

```
{
    "SOCIAL_AUTH_GOOGLE_OAUTH2_KEY": "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx",
    "SOCIAL_AUTH_GOOGLE_OAUTH2_SECRET": "XXXXXXXXXXXXXXXXXXXXXXX"
}
```

Добавляем константу **SOCIAL_AUTH_URL_NAMESPACE** со значением **social**.

После установки приложения и настройки необходимо выполнить миграции – в БД проекта появятся новые таблицы с названиями вида **social_...** Нас интересует **social_auth_usersocialauth**: в ней после аутентификации пользователя через социальную сеть будет появляться запись. Посмотрите поля этой таблицы и данные в них. Одновременно будет появляться запись в основной таблице пользователей магазина **authapp_shopuser**. Связь между таблицами через поле **user_id**.

Следующим шагом подключаем диспетчер URL приложения **social_django**:

geekshop/urls.py

```
...
urlpatterns = [
    ...
    re_path(r'^auth/verify/google/oauth2/', include("social_django.urls",
namespace="social"))
]
```

Теперь ответ сервера Google+ автоматически будет обработан приложением **social_django**.

Размещаем на странице входа в систему новую ссылку:

geekshop/authapp/templates/authapp/login.html

```
...
<button class="btn btn-round form-control">
    <a href="{% url 'auth:register' %}" class="">
        зарегистрироваться
    </a>
</button>
<button class="btn btn-round form-control">
    <a href="{% url 'social:begin' 'google-oauth2' %}?next=/">
     Google+ sign in
    </a>
</button>
...
```

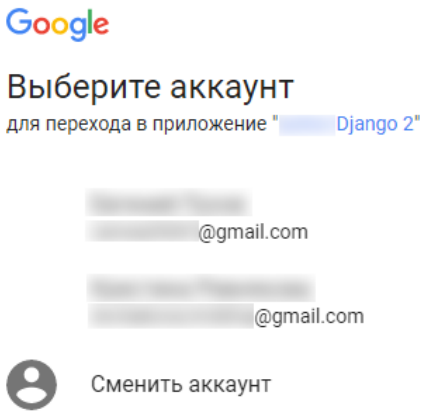
Используем пространство имен **social**, указанное в диспетчере URL, и адрес **begin** в этом пространстве. В качестве дополнительного аргумента передаем название протокола аутентификации

google-oauth2. В конце адреса дописываем **?next=/** для перехода на главную страницу после аутентификации.

Последний штрих: так как API Google+ не передает в ответе возраст пользователя, пропишем в модели **ShopUser** пользователя магазина значение возраста по умолчанию:

```
age = models.PositiveIntegerField(verbose_name = 'возраст', default=18)
```

Для проверки запускаем сервер Django и переходим по ссылке **Google+ sign in** – мы должны увидеть следующее:



После выбора аккаунта и подтверждения разрешений, вы должны оказаться на главном окне проекта и увидеть свое имя в строке меню, как и в случае обычной аутентификации по логину и паролю. Если этого не произошло, надо искать ошибку.

Обратите внимание, что после аутентификации в основной таблице пользователей магазина **authapp_shopuser** автоматически заполнились поля с логином, паролем, именем, фамилией и электронной почтой пользователя.

Входим в систему под уже существующей учетной записью – все должно работать как раньше. Попробуем создать нового пользователя – при верификации по e-mail получим ошибку в консоли:

```
('You have multiple authentication backends configured and therefore must
provide the `backend` argument or set the `backend` attribute on the user.',)
```

Уточним процесс аутентификации пользователя в контроллере **authapp.views.verify** – при вызове метода **auth.login()** явно зададим бэкенд:

```
  auth.login(request, user, backend='django.contrib.auth.backends.ModelBackend')
```