ТЕХНОАТОМ

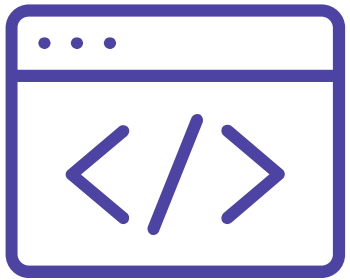
Веб-программирование Python

Лекция 9. Интеграция

Михалев Олег

Сегодня

- Интеграция компонент
- Интеграция приложений
- REST



Ввиду тех благ, что дает ООП и модульная структура приложений на Python, интеграция простых компонент между собой не составляет труда

Немного вспомним

Django Cache
https://docs.djangoproject.com/en/1.10/topics/cache/

Интеграция

ТЕХНОАТОМ

settings.py

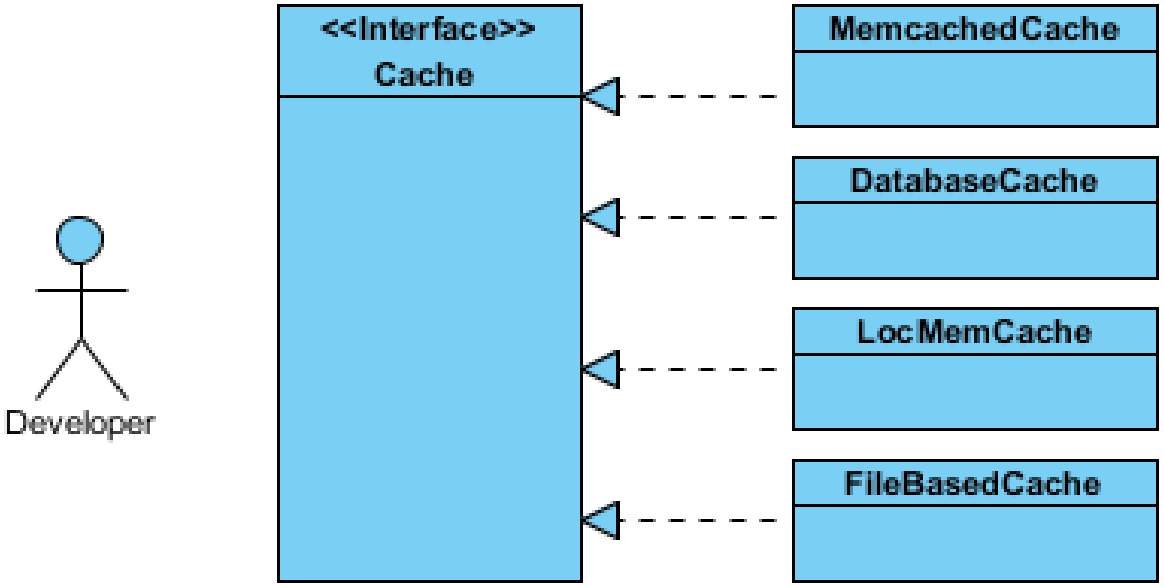
```
1. SESSION_ENGINE = 'django.contrib.sessions.backends.cache'

2. CACHES = {
3.     'default': {
4.         'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',
5.         'LOCATION': [
6.             '172.16.32.54:6379',
7.             '172.16.32.57:6379',
8.         ],
9.         'TIMEOUT': 300,
10.        'OPTIONS': {
11.            'MAX_ENTRIES': 10000
12.        }
13.    },
14.}
```

Мы используем общий интерфейс сессий, несмотря на конечную реализацию сессий

Мы используем общий интерфейс сессий, несмотря на конечную реализацию кэша

Мы работаем с абстракциями, абстракции не зависят от деталей реализации



Инверсия управления

В языках со статической типизацией
используют интерфейсы

Интерфейсы
+
Фабрики
Внедрение зависимостей
Сигналы

В Python динамическая типизация

Если это выглядит как утка, плавает как утка и крякает как утка, то это, возможно, и есть утка

В Python мы можем обойтись без явных механизмов инверсии управления

Немного посмотрим

Django Middleware
https://docs.djangoproject.com/en/1.10/topics/http/middleware/

Возьмем простой пример, чтобы понять принцип Django Middleware

Интеграция

ТЕХНОАТОМ

middleware.py

```
1. class Middleware(object):

2.     def __init__(self, get_response):
3.         # Make some config
4.         # Save default handler
5.         self.get_response = get_response

6.     def __call__(self, request):
7.         # Pre-process request
8.         # Get response
9.         response = self.get_response(request)
10.        # Post-process response
11.        return response
```

Напоминает уже знакомый WSGI Middleware

Возьмем более сложный пример, чтобы понять принцип инверсии управления

Интеграция

ТЕХНОАТОМ

backends.py

```
1. class MemoryBasedBackend(object):

2.     def __init__(self, options):
3.         self.storage = {}

4.     def set_data(self, key, value):
5.         self.storage[key] = value

6.     def get_data(self, key):
7.         return self.storage.get(key)
```

Интеграция

ТЕХНОАТОМ

backends.py

```
1. from django.core.exceptions import ImproperlyConfigured

2. class FileBasedBackend(object):

3.     def __init__(self, options):
4.         try:
5.             self.location = options['LOCATION']
6.         except (TypeError, KeyError):
7.             raise ImproperlyConfigured(
8.                 'Edit your STORAGE.OPTIONS setting'
9.             )

10.    ...
```

Интеграция

ТЕХНОАТОМ

backends.py

```
1.      ...

2.      def set_data(self, key, value):
3.          value = value.encode('utf-8')
4.          with open(self.location + '/' + key, 'w') as output:
5.              output.write(value)

6.      def get_data(self, key):
7.          data = None
8.          try:
9.              input = open(self.location + '/' + key)
10.             data = input.read().decode('utf-8')
11.         except:
12.             pass
13.         return data
```

Интеграция

ТЕХНОАТОМ

middleware.py

```
1. from django.conf import settings
2. from django.core.exceptions import ImproperlyConfigured

3. from .backends import FileBasedBackend, MemoryBasedBackend

4. ...
```

Интеграция

ТЕХНОАТОМ

middleware.py

```
1. class StorageMiddleware(object):

2.     def __init__(self, get_response):
3.         conf = getattr(settings, 'STORAGE', None)
4.         if not isinstance(conf, dict):
5.             raise ImproperlyConfigured('Edit your STORAGE setting')
6.         engine_type = conf.get('TYPE')
7.         engine_options = conf.get('OPTIONS')
8.         if engine_type == 'FILE':
9.             self.engine = FileBasedBackend(options)
10.        elif engine_type == 'MEMORY':
11.            self.engine = MemoryBasedBackend(options)
12.        else:
13.            raise ImproperlyConfigured('Edit your STORAGE.TYPE setting')
14.        self.get_response = get_reponse
```

Интеграция

ТЕХНОАТОМ

middleware.py

```
1.  def __call__(self, request);
2.          setattr(request, 'storage', self.engine)
3.          return self.get_response(request)
```

Мы используем абстрактный **request.storage**, не обращая внимания на детали реализации

Также (грубо говоря), получается и **request.session**

Назначение middleware остается тем же

Пред- и постобработка сообщений
между клиентом и веб-приложением

Связывание функций через предоставление
интерфейсов компонент веб-приложения

И посмотрим еще

Django Signals

https://docs.djangoproject.com/en/1.10/topics/signals/

https://docs.djangoproject.com/en/1.10/ref/signals/

Сигналы - это события, которые могут быть обработаны диспетчером

Диспетчер маршрутизирует сигналы к их обработчикам

- Тип сигнала
- Компонент посылающий сигнал (sender)
- Компонент принимающий сигнал (receiver)

Встроенные сигналы позволяют реагировать на события внутри фреймворка

Чаще всего применяются сигналы моделей

Интеграция

ТЕХНОАТОМ

models.py

```
1. class Account(models.Model):
2.     '''Bank account'''

3.     number = models.CharField('Account number', max_length=22)
4.     amount = models.DecimalField('Current amount')


5. class Charge(models.Model):
6.     '''Bank transaction'''

7.     account = models.ForeignKey(Account, related_name='charges')
8.     amount = models.DecimalField('Charge amount')
```

Интеграция

ТЕХНОАТОМ

handlers.py

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

3. from finances.models import Charge

4. @receiver(post_save, sender=Charge)
5. def update_account_on_charge_create(sender, instance, created, **kwargs):
6.     if not created:
7.         return
8.     account = instance.account
9.     account.amount += instance.amount
10.    account.save()
```

Интеграция

ТЕХНОАТОМ

apps.py

```
1. from django.apps import AppConfig

2. class FinancesConfig(AppConfig):

3.     name = 'finances'

4.     def ready(self):
5.         from finances.handlers import *
```

Можно объявлять собственные сигналы

Интеграция

ТЕХНОАТОМ

signals.py

```
1. from django.dispatch import Signal

2. get_millionaire = Signal(
3.     providing_args=['user']
4. )
```

Интеграция

ТЕХНОАТОМ

handlers.py

```
from finances.models import Account
from finances.signals import get_millionaire

@receiver(post_save, sender=Account)
def update_account(sender, instance, created, **kwargs):
    if instance.amount >= 1000000:
        get_millionaire.send(sender=sender, user=instance.user)


@receiver(get_millionaire, sender=Account)
def call_to_federal_tax_service(sender, user, **kwargs):
    ...
```

Удобство очевидно

Особенно, учитывая, что транзакция БД распространяется и на обработчики сигналов

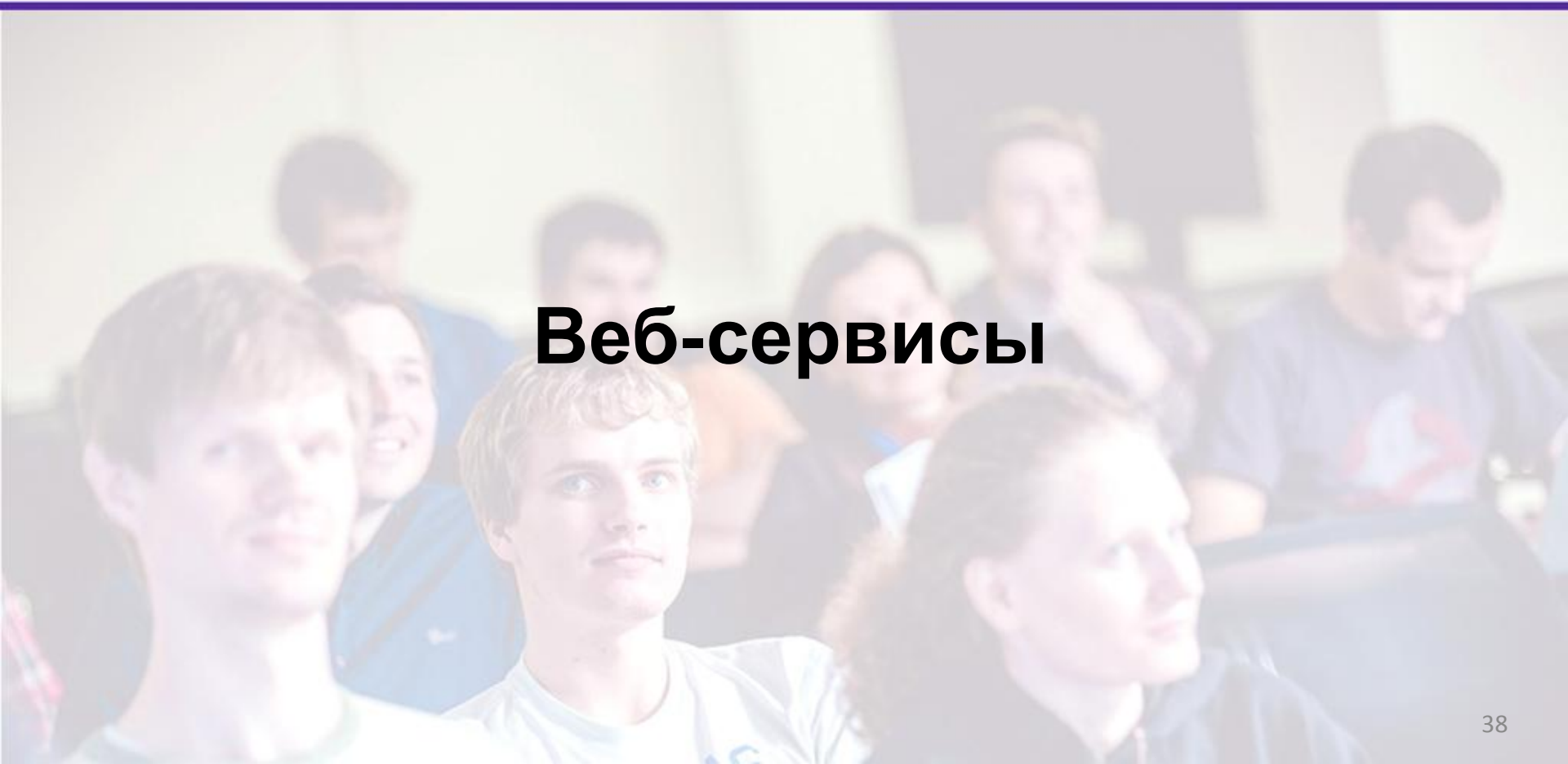
Но мы бы даже не узнали о том,
что нужна транзакция

Сигналы лучше не использовать без весомых аргументов потому, что:

- Логика "размазывается" по разным частям приложения
- Рекурсивные вызовы могут сломать приложение
- Проблемы с дублированием могут сломать вам жизнь

А давайте представим, что нам необходимо интегрироваться со сторонним приложением

Какие проблемы возникнут?
Каковы пути решения?



Веб-сервисы

Под сервис-ориентированной архитектурой (SOA) обычно понимают технологии вызова удаленных процедур на базе формата XML

XML-RPC
Extensible Markup Language Remote Procedure Call

xmlrpclib
https://docs.python.org/3/library/xmlrpc.server.html
https://docs.python.org/3/library/xmlrpc.client.html

SOAP
Simple Object Access Protocol

soaplib, spyne
https://pypi.python.org/pypi/spyne/
http://spyne.io/docs/2.10/
suds, suds-py3
https://pypi.python.org/pypi/suds-py3/
https://github.com/cackharot/suds-py3/

Интеграция

ТЕХНОАТОМ

services.py

```
from spyne.model import ComplexModel, Array, primitive

class Account(ComplexModel):
    number = primitive.String
    amount = primitive.Float


class SearchRequest(ComplexModel):
    number = primitive.String

class SearchResponse(ComplexModel):
    results = Array(Account)

...
```

services.py

```
...

class AccountService(ServiceBase):

    @rpc(SearchRequest, _returns=SearchResponse)
    def search(self, account_request):
        result = [ ]
        for model in search_accounts(account_request.number):
            account = Account( )
            account.number = model.number
            account.amount = model.amount
            results.append(account)
        response = SearchResponse( )
        response.results = results
        return response
```

Интеграция

ТЕХНОАТОМ

server.py

```
from spyne.server.wsgi import WsgiApplication
from spyne.application import Application as SoapApplication

from services import AccountService

from wsgiref.simple_server import make_server


application = WsgiApplication(
    SoapApplication([AccountService], 'services')
)

if __name__ == '__main__':
    server = make_server('', 80, application)
    server.serve_forever( )
```

WSDL + XML Schema
http://localhost/?wsdl

SOAP Port
http://localhost/?wsdl

Интеграция

ТЕХНОАТОМ

client.py

```
from suds.client import Client, WebFault

if __name__ == '__main__':
    client = Client('http://localhost/?wsdl')
    request = client.factory.create('SearchRequest')
    request.number = '9876543210'
    print('Try to search: %s' % request.number)
    try:
        response = client.service.search(request)
    except WebFault:
        print('Result: Error')
    else:
        for result in response.results.Account:
            account = (result.number, result.amount)
            print('Found %s: %.2f RUR' % account)
```

XML тяжелый, но развитый

Описание данных (**XML Schema**)
Обращение к структуре (**DOM**)
Обращение к данным (**XPath**, **XQuery**)
Преобразование данных (**XSLT**)
Описание сервисов (**WSDL**)
Обращение к сервисам (**SOAP, XML-RPC**)
Описание процессов (**BPMN**, **BPEL**)

И это далеко (очень далеко) не все

Многие крупные системы используют SOAP

REST
Representational State Transfer

Fielding, Roy Thomas
Architectural Styles and the Design of Network-based Software Architectures
http://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf

REST - это архитектурный стиль, оперирующий абстракциями

Холивары "REST vs SOAP" несостоятельны

Ключевой абстракцией в REST является **ресурс** – сущность предметной области, имеющая идентификатор

Во взаимодействии обязательно участвуют две роли - **клиент** и **сервер**

Коллекция - некоторая совокупность ресурсов

Ресурсы и коллекции не содержат промежуточных состояний

Вся необходимая для выполнения запроса информация передается клиентом

Ресурсы и коллекции имеют уникальные идентификаторы URI (URL)

Единообразие интерфейсов доступа гарантирует прозрачность и позволяет организовать кэширование

Ресурс может иметь несколько форматов представлений (MIME)

Вся необходимая информация о формате представления передается сервером

Клиент указывает предпочитаемый формат представления в запросе

Взаимодействие с ресурсами и их коллекциями осуществляется средствами транспортного протокола (HTTP)

Взаимодействие с ресурсами и их коллекциями по сути повторяет деятельность Интернет-пользователя

Последнее утверждение обязывает сервер отвечать корректными по смыслу HTTP-состояниями

Типовые операции коллекции

- Получение списка членов коллекции (метод **GET**)
- Создание нового ресурса (метод **POST**)
- Обновление всей коллекции (метод **PUT**)
- Удаление всей коллекции (метод **DELETE**)

Типовые операции ресурса

- Получение представления ресурса (метод **GET**)
- Обновление ресурса (метод **PUT**)
- Удаление ресурса (метод **DELETE**)

На примерах станет понятнее

Счета пользователя (коллекция)

GET /api/accounts.json

Получение списка счетов пользователя

Ответ

200 OK | 403 Forbidden

JSON

Счета пользователя (коллекция)

POST /api/accounts.json
Создание счета пользователя

Ответ
201 Created | 403 Forbidden
JSON*

Заголовки ответа
Location: /api/accounts/1.json

Счет пользователя (ресурс)

GET /api/accounts/1.json
Получение счета пользователя

Ответ
200 OK | 403 Forbidden | 404 Not Found
JSON

Счет пользователя (ресурс)

PUT /api/accounts/1.json
Обновление счета пользователя

Ответ
200 OK | 403 Forbidden | 404 Not Found
JSON*

Счет пользователя (ресурс)

DELETE /api/accounts/1.json

Удаление счета пользователя

Ответы

204 No Content | 403 Forbidden | 404 Not Found

JSON*

Спецификаций на REST нет,
но можно ориентироваться на идеал

Django REST framework

https://pypi.python.org/pypi/djangorestframework/

http://www.django-rest-framework.org/

Необходимо добавить **rest_framework** в **INSTALLED_APPS** settings.py

Рекомендуется настроить параметр **REST_FRAMEWORK**

Интеграция

ТЕХНОАТОМ

settings.py

```
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': [
        'rest_framework.authentication.SessionAuthentication',
    ],
    'DEFAULT_PERMISSION_CLASSES': [
        'rest_framework.permissions.IsAuthenticated',
    ],
    'DEFAULT_PARSER_CLASSES': [
        'rest_framework.parsers.JSONParser',
    ],
    'DEFAULT_RENDERER_CLASSES': [
        'rest_framework.renderers.JSONRenderer',
    ],
}
```

serializers.py

```
from rest_framework.serializers import Serializer, ModelSerializer
...
```

За приведение данных ресурса к необходимому содержанию отвечают сериализаторы

serializers.py

```
from rest_framework.validators import UniqueValidator

class AccountSerializer(serializers.ModelSerializer):

    number = serializers.CharField(
        max_length=22,
        validators=[
            UniqueNumberValidator(queryset=Account.objects.all())
        ]
    )

    class Meta:
        model = Account
        fields = ['number', 'amount']
```

Собственные валидации описываются вызываемыми объектами, зачастую функторами

```
class UniqueNumberValidator(object):

    def __init__(self, queryset):
        self.queryset = queryset

    def __call__(self, value):
        if self.queryset.filter(number=value).exists():
            raise serializers.ValidationError('This field must be a unique.')
```

serializers.py

```
class ChargeSerializer(serializers.ModelSerializer):

    account = AccountSerializer(source='account')

    class Meta:
        model = Charge
        fields = [
            'id',
            'account',
            'amount',
        ]
        read_only_fields = [
            'id',
        ]
```

Сериализаторам возможно переопределить методы **create**, **update** или **save**

Для создания представления используется декоратор **api_view**

```
from rest_framework import status
from rest_framework.decorators import api_view
from rest_framework.response import Response

@api_view(['GET'])
def get_charges(request, account):
    charges = Charge.objects.filter(account__number=account)
    serializer = ChargeSerializer(charges, many=True)
    return Response(serializer.data, status=status.HTTP_200_OK)
```

views.py

```
from rest_framework.parsers import JSONParser
from rest_framework.renderers import JSONRenderer
...
```

За приведение данных ресурса к необходимому формату представления отвечают парсеры и рендереры

serializers.py

```
@api_view(['GET'])
@parser_classes(['JSONParser'])
@renderer_classes(['JSONRenderer'])
def get_charge(request, pk):
    try:
        charge = Charge.objects.get(pk=pk)
    except Charge.DoesNotExist:
        return Response(status=status.HTTP_404_NOT_FOUND)
    serializer = ChargeSerializer(charge)
    return Response(serializer.data)
```

Удобнее использовать представления, основанные на классах

```
from rest_framework import generics

class AccountSimpleResource(generics.RetrieveAPIView):

    serializer_class = AccountSerializer

    def get_object(self):
        return get_object_or_404(Account, **self.kwargs)
```

Методы-обработчики классов называются совсем как методы HTTP, для комбинации методов можно использовать примеси

- **get**
- **post**
- **put**
- **delete**

Еще более полно ресурс помогут описать viewsetы

```
class ChargeViewSet(viewsets.ViewSet):

    def list(self, request):
        pass
    def create(self, request):
        pass
    def retrieve(self, request, pk=None):
        pass
    def update(self, request, pk=None):
        pass
    def partial_update(self, request, pk=None):
        pass
    def destroy(self, request, pk=None):
        pass
```

Подключать такие представления необходимо в **urlpatters**

```
urlpatterns = [
    ... ,
    url(
        r'^api/account/(?P<number>/d+)/.json$', AccountSimpleResource.as_view( )
    ),
    url(
        r'^api/charges/.json$', ChargeViewSet.as_view({'get': 'list'})
    ),
    url(
        r'^api/charges/(?P<pk>/d+)/.json$', ChargeViewSet.as_view({'get': 'retrieve'})
    ),
]
```

Аутентификацию возможно включать параметром **authentication_classes**

По умолчанию для веб-приложений рекомендуется выставлять **SessionAuthentication**

Для мобильных приложений может потребоваться аутентификация по токену

Для этого необходимо подключить **rest_framework.authtoken** в **INSTALLED_APPS** settings.py и выполнить миграции

Важно помнить, что токены нужно выдавать и периодически обновлять

Права доступа возможно включить параметром **permission_classes**

В нашем случае мы оградим API приложения от посторонних классом **IsAuthenticated**

Собственные проверки прав доступа можно определять классами

```
class AdminPermission(permissions.BasePermission):

    def has_permission(self, request, view):
        return request.user.is_staff
```

Возможно использовать контекстные права доступа

```
class IsOwner(permissions.BasePermission):

    def has_object_permission(self, request, view, obj):
        return obj.owner == request.user
```

В случае неудачи авторизации пользователь получит сообщение с описанием ошибки

Фреймворк оборачивает исключительные ситуации **PermissionDenied** и **Http404** Django

Лекция заканчивается, но не заканчивается DRF

До дна шляпы еще далеко
Читайте документацию

Спасибо за внимание!

Михалев Олег
mailto:mhalairt@gmail.com

Домашнее задание

ТЕХНОАТОМ

Подключить Django REST Framework и реализовать сервисы моделей Account и Charge, а также сервис статистики.