



# Django REST Framework

Michał Wawrzyniak  
kontakt@msites.pl



**Wi** Wydział Informatyki  
Politechnika Białostocka



**OWNED**OUTCOMES



grupa **zpr**  
media

POWERED BY

**NETSTATION**

# Earlier presentations



- Django ORM #31 (08/05/2018)
- Data processing and visualization with PySpark and Apache Zeppelin #26 (19/09/2017)
- Docker for developers #19 (13/09/2016)
- Django projects optimization #17 (17/05/2016)

# Good practices? What's that?



- DRY - Don't Repeat Yourself
- KISS - Keep It Simple Stupid
- PEP8 - Style Guide for Python Code

*E741 - Variables named **I**, **O**, and **l** can be very hard to read. This is because the letter **I** and the letter **l** are easily confused, and the letter **O** and the number **0** can be easily confused.*

# Useful tools



- Flake8
- mccabe
- pylint
- isort

# Before isort



```
from my_lib import Object
print("Hey")
import os
from my_lib import Object3
from my_lib import Object2
import sys
from third_party import lib15, lib1, lib2, lib3, lib4, lib5, lib6, lib7, lib8, lib9, lib10, lib11, lib12
import sys
from __future__ import absolute_import
from third_party import lib3
print("yo")
```

# After isort



```
from __future__ import absolute_import
```

```
import os  
import sys
```

```
from third_party import (lib1, lib2, lib3, lib4, lib5, lib6, lib7, lib8,  
                          lib9, lib10, lib11, lib12)
```

```
from my_lib import Object, Object2, Object3
```

```
print("Hey")  
print("yo")
```

# Most common errors in writing code IMHO



- `from some_package import *`
- Python is not Java, use `sneak_case`
- naming variables meaningless
- writing code without blank lines to separate sections
- overuse of one-liners
- mixing Class-Based Views and Function-Based Views in one app



# Tests



“Code without tests is broken as designed”

Jacob Kaplan-Moss

# Tests



TDD – Test Driven Development

Useful libraries:

- unittest / unittest2
- pytest
- nose
- coverage
- model\_mommy
- FactoryBoy

# Django REST Framework



- Serializers
- Views
- Viewsets
- Routers
- Throttling
- Filtering
- Testing
- and more...

# Example models



```
from django.db import models
```

```
class Article(models.Model):  
    title = models.CharField(max_length=100)  
    rank = models.PositiveIntegerField(default=0)
```

```
class Comment(models.Model):  
    article = models.ForeignKey(  
        Article, on_delete=models.CASCADE, related_name='comments',  
    )  
    text = models.TextField(default="")
```

# DRF Serializers



```
from rest_framework import serializers
```

```
from .models import Article
```

```
class ArticleSerializer(serializers.ModelSerializer):
```

```
    class Meta:
```

```
        fields = ('id', 'title', 'rank')
```

```
        model = Article
```

# DRF Serializers fields



- BooleanField
- CharField
- IntegerField
- DateTimeField
- FileField
- ListField
- ReadOnlyField
- ModelField
- SerializerMethodField
- and many more...

# Nested serializers



```
class CommentSerializer(serializers.ModelSerializer):
```

```
    class Meta:
```

```
        model = Comment
```

```
        fields = ('text', )
```

```
class ArticleSerializer(serializers.ModelSerializer):
```

```
    comments = CommentSerializer(many=True)
```

```
    class Meta:
```

```
        model = Article
```

```
        fields = ('title', 'comments')
```

# Serializers validation



```
from rest_framework.validators import UniqueTogetherValidator
```

```
class ExampleSerializer(serializers.Serializer):
```

```
    # ...
```

```
    class Meta:
```

```
        validators = [  
            UniqueTogetherValidator(  
                queryset=ToDoItem.objects.all(),  
                fields=('list', 'position')  
            )  
        ]
```



# DRF Views



```
from rest_framework.views import APIView
from rest_framework.response import Response
from django.contrib.auth.models import User

class ListUsers(APIView):
    """View to list all users in the system."""
    def get(self, request, format=None):
        """
        Return a list of all users.
        """
        usernames = User.objects.values_list('username', flat=True)
        return Response(usernames)
```

# DRF Views



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

```
@api_view()
def hello_world(request):
    return Response({"message": "Hello, world!"})
```

# Generic views



The generic views provided by REST framework allow you to quickly build API views that map closely to your database models.

If the generic views don't suit the needs of your API, you can drop down to using the regular `APIView` class, or reuse the mixins and base classes used by the generic views to compose your own set of reusable generic views.

# Generic views



```
from django.contrib.auth.models import User
from myapp.serializers import UserSerializer
from rest_framework import generics
```

```
class UserList(generics.ListCreateAPIView):
    queryset = User.objects.all()
    serializer_class = UserSerializer
```

# Mixins



- ListModelMixin
- CreateModelMixin
- RetrieveModelMixin
- UpdateModelMixin
- DestroyModelMixin

# Concrete View Classes



- CreateAPIView
- ListAPIView
- RetrieveAPIView
- DestroyAPIView
- UpdateAPIView
- ListCreateAPIView
- and more...

# DRF Viewsets and routers



Django REST framework allows you to combine the logic for a set of related views in a single class, called a `ViewSet`.

In other frameworks you may also find conceptually similar implementations named something like 'Resources' or 'Controllers'.

# DRF Viewsets and routers



```
from rest_framework import viewsets
from rest_framework.routers import DefaultRouter
```

```
from .serializers import ArticleSerializer
```

```
class ArticleViewSet(viewsets.ModelViewSet):
```

```
    queryset = Article.objects.all()
    serializer_class = ArticleSerializer
```

```
router = DefaultRouter()
router.register(r'articles', ArticleViewSet, basename='article')
```





# Routers mapping

URL Style	HTTP Method	Action	URL Name
{prefix}/	GET	list	{basename}-list
	POST	create	
{prefix}/{url_path}/	GET, or as specified by `methods` argument	`@action(detail=False)` decorated method	{basename}-{url_name}
{prefix}/{lookup}/	GET	retrieve	{basename}-detail
	PUT	update	
	PATCH	partial_update	
	DELETE	destroy	
{prefix}/{lookup}/{url_path}/	GET, or as specified by `methods` argument	`@action(detail=True)` decorated method	{basename}-{url_name}

# Filtering



```
from rest_framework import filters
```

```
class UserListView(generics.ListAPIView):  
    queryset = User.objects.all()  
    serializer_class = UserSerializer  
    filter_backends = (filters.SearchFilter,)  
    search_fields = ('username', 'email')
```

GET <http://example.com/api/users?search=russell>

# Ordering



```
class UserListView(generics.ListAPIView):  
    queryset = User.objects.all()  
    serializer_class = UserSerializer  
    filter_backends = (filters.OrderingFilter,)  
    ordering_fields = ('username', 'email')
```

GET <http://example.com/api/users?ordering=username>

# Serializer context



There are some cases where you need to provide extra context to the serializer in addition to the object being serialized.

```
serializer = AccountSerializer(account, context={'request': request})
```

```
serializer.data
```

```
# {'id': 6, 'owner': u'denvercoder9', 'created': datetime.datetime(2013, 2, 12, 09, 44, 56, 678870), 'details': 'http://example.com/accounts/6/details'}
```

# Overriding to\_representation() method



```
class HighScoreSerializer(serializers.BaseSerializer):
```

```
    def to_representation(self, obj):  
        return {  
            'score': obj.score,  
            'player_name': obj.player_name  
        }
```

# When you need more serializers ... use get\_serializer



```
class ArticleViewSet(viewsets.ModelViewSet):
```

```
    def get_serializer_class(self):  
        if self.action == 'create':  
            return CreateArticleSerializer  
        elif self.action == 'list':  
            return ListArticleSerializer  
        return ArticleSerializer
```

# ORM optimization in DRF



```
class ArticleSerializer(serializers.ModelSerializer):  
    comments = serializers.SerializerMethodField()
```

```
class Meta:  
    fields = ('id', 'title', 'comments', 'rank')  
    model = Article
```

```
def get_comments(self, obj):  
    return obj.comments.count()
```



# ORM optimization in DRF



```
from django.db.models import Count
```

```
class ArticleViewSet(viewsets.ModelViewSet):  
    queryset = Article.objects.annotate(comment_count=Count('comments')).all()
```

```
class ArticleSerializer(serializers.ModelSerializer):  
    comments = serializers.IntegerField(source='comment_count')
```

```
class Meta:  
    fields = ('id', 'title', 'comments', 'rank')  
    model = Article
```

# Throttling



Throttling is similar to permissions, in that it determines if a request should be authorized. Throttles indicate a temporary state, and are used to control the rate of requests that clients can make to an API.

# Throttling



```
REST_FRAMEWORK = {  
    'DEFAULT_THROTTLE_CLASSES': (  
        'rest_framework.throttling.AnonRateThrottle',  
        'rest_framework.throttling.UserRateThrottle'  
    ),  
    'DEFAULT_THROTTLE_RATES': {  
        'anon': '100/day',  
        'user': '1000/day'  
    }  
}
```

# Permissions in DRF



Together with authentication and throttling, permissions determine whether a request should be granted or denied access.

```
REST_FRAMEWORK = {  
    'DEFAULT_PERMISSION_CLASSES': (  
        'rest_framework.permissions.IsAuthenticated',  
    )  
}
```

# Permissions in DRF



```
from rest_framework.permissions import IsAuthenticated
from rest_framework.response import Response
from rest_framework.views import APIView
```

```
class ExampleView(APIView):
    permission_classes = (IsAuthenticated,)
```

```
    def get(self, request, format=None):
        content = {
            'status': 'request was permitted'
        }
        return Response(content)
```

# Authentication



Authentication is the mechanism of associating an incoming request with a set of identifying credentials, such as the user the request came from, or the token that it was signed with.

The permission and throttling policies can then use those credentials to determine if the request should be permitted.

# Authentication



```
REST_FRAMEWORK = {  
    'DEFAULT_AUTHENTICATION_CLASSES': (  
        'rest_framework.authentication.BasicAuthentication',  
        'rest_framework.authentication.SessionAuthentication',  
    )  
}
```

# Authentication



```
from rest_framework.authentication import SessionAuthentication,  
BasicAuthentication
```

```
from rest_framework.permissions import IsAuthenticated
```

```
class ExampleView(APIView):  
    authentication_classes = (SessionAuthentication, BasicAuthentication)  
    permission_classes = (IsAuthenticated,)  
  
    def get(self, request, format=None):  
        content = {  
            'user': unicode(request.user), # `django.contrib.auth.User` instance.  
        }  
        return Response(content)
```



# Cache results of time-consuming operations



```
class BasketSerializer(serializers.ModelSerializer):
```

```
...
```

```
def get_sum_with_tax(self, obj):
```

```
    if not hasattr(self, '_sum'):
```

```
        self._sum = obj.calc_sum()
```

```
    return self._sum * TAX_RATE
```

# Testing in DRF

```
from django.urls import reverse
from rest_framework import status
from rest_framework.test import APITestCase
from myproject.apps.core.models import Account


class AccountTests(APITestCase):
    def test_create_account(self):
        """Ensure we can create a new account object."""
        url = reverse('account-list')
        data = {'name': 'DabApps'}
        response = self.client.post(url, data, format='json')
        self.assertEqual(response.status_code, status.HTTP_201_CREATED)
        self.assertEqual(Account.objects.count(), 1)
        self.assertEqual(Account.objects.get().name, 'DabApps')
```

# How to extend DRF?



- DRF Writable Nested
- Dynamic Serializer Fields
- DRF Nested routers
- Django filters and DRF filters

# DRF Writable Nested



```
class CreateArticleSerializer(serializers.ModelSerializer):  
    comments = CommentSerializer(many=True)
```

```
class Meta:  
    model = Article  
    fields = ('title', 'comments')
```

**AssertionError:** The `.create()` method does **not** support writable nested fields by default.

Write an explicit `.create()` method **for** serializer

`articles.serializers.CreateArticleSerializer`, **or set** `read_only=True` on nested serializer fields.

# DRF Writable Nested



This is a writable nested model serializer for Django REST Framework which allows you to create/update your models with related nested data.

The following relations are supported:

- OneToOne (direct/reverse)
- ForeignKey (direct/reverse)
- ManyToMany (direct/reverse excluding m2m relations with through model)
- GenericRelation (this is always only reverse)

# DRF Writable Nested



```
from drf_writable_nested import WritableNestedModelSerializer
```

```
class CreateArticleSerializer(WritableNestedModelSerializer):  
    comments = CommentSerializer(many=True)
```

```
class Meta:  
    model = Article  
    fields = ('title', 'comments')
```

# Dynamic Serializer Fields



```
GET /articles/
```

```
GET /articles/?fields=id,title
```

```
GET /articles/?omit=rank
```

```
from drf_dynamic_fields import DynamicFieldsMixin
```

```
class ArticleSerializer(DynamicFieldsMixin, serializers.ModelSerializer):
```

```
    class Meta:
```

```
        fields = ('id', 'title', 'rank')
```

```
        model = Article
```

# DRF Nested routers



```
from rest_framework_nested import routers
from .views import ArticleViewSet, CommentViewSet
```

```
router = routers.SimpleRouter()
router.register(r'articles', ArticleViewSet)
```

```
comment_router = routers.NestedSimpleRouter(router, r'articles', lookup='article')
comment_router.register(r'comments', CommentViewSet,
base_name='article-comments')
```



# DRF Nested routers



`^articles/$ [name='article-list']`

`^articles/(?P<pk>[^/\.]+)/$ [name='article-detail']`

`^articles/(?P<article_pk>[^/\.]+)/comments/$ [name='article-comments-list']`

`^articles/(?P<article_pk>[^/\.]+)/comments/(?P<pk>[^/\.]+)/$ [name='article-comments-detail']`

# Django filter and DRF Filters



```
from django_filters.rest_framework import DjangoFilterBackend
```

```
class ProductList(generics.ListAPIView):  
    queryset = Product.objects.all()  
    serializer_class = ProductSerializer  
    filter_backends = (DjangoFilterBackend,)  
    filter_fields = ('category', 'in_stock')
```

GET [http://example.com/api/products?category=clothing&in\\_stock=True](http://example.com/api/products?category=clothing&in_stock=True)

# JSON Web Token Authentication support for DRF



JSON Web Token Authentication support for Django REST Framework

# Django REST Auth



- User Registration with activation
- Login/Logout
- Retrieve/Update the Django User model
- Password change
- Password reset via e-mail
- Social Media authentication

# Links



- <https://pycodestyle.readthedocs.io/en/latest/intro.html#error-codes>
- <https://www.djangoproject.com/>
- <https://www.django-rest-framework.org/>
- <https://github.com/carltongibson/django-filter>
- <https://github.com/django-guardian/django-guardian>
- <https://github.com/alanjds/drf-nested-routers>
- <https://github.com/dbrgn/drf-dynamic-fields>
- <https://github.com/beda-software/drf-writable-nested>
- <https://github.com/Tivix/django-rest-auth>
- <https://github.com/GetBlimp/django-rest-framework-jwt>
- [https://github.com/berinhard/model\\_mommy](https://github.com/berinhard/model_mommy)
- [https://github.com/FactoryBoy/factory\\_boy](https://github.com/FactoryBoy/factory_boy)

---

# Questions?