WEB APIs

Django Rest Framework, Verbos HTTP, Status codes, serializadores e API Views

Ely – elydasilvamiranda@gmail.com

Referências e projeto

- Esses slides usam como base a seguintes fonte:
  - Capítulo 2 do livro Building RESTful Python Web Services.
- Baixe o esqueleto do projeto no drive da disciplina.

Software recomendável

- Python 3.5
- Instaláveis via pip:
  - Django
  - Django Restframework (DRF);
  - Markdown;
  - Django-filter.

Alguns requisitos de framework REST

- Serialização e deserialização;
- Paginação;
- Validação;
- Autenticação e autorização;
- Queries personalizadas;
- Tratamento de respostas HTTP personalizadas;
- Paginação;
- Cache;
- Throttling;
- Tests.

Django REST Framework (DRF)

- Pacote do Django, estende o framework;
- Views, autenticação e utilitários para criar WEB APIs;
- Fácil configuração e "low boilerplate".

DRF

- Principais elementos:
  - Serializers: provêem serialização/deserialização simplificados;
  - Views: permitem implementa de forma desacoplada os métodos HTTP
  - URLs: possui um padrão de URLs prático e elementos de roteamento;
  - Autenticação plugável.

Estudo de caso

- Um sistema armazena dados de jogos, jogadores e seus escores.
- Game:
  - Definido por um id, name, data de lançamento e uma categoria (3D, RPG, 2D Arcade) e uma data de cadastro;
- Player:
  - id, nome, gênero e data de cadastro;
- Score:
  - Id, jogador, jogo, o valor do escore e a data em que o valor foi alcançado.

Jogos e os verbos HTTP

Verbo	Semântica	URL
GET	Obtém todos os jogos ordenados pelo seu nome	http://localhost/games
GET	Obtém um jogo	http://localhost/games/id
POST	Cria um novo jogo na coleção	http://localhost/games
PUT	Atualiza um jogo	http://localhost/games/id
DELETE	Exclui um jogo existente	http://localhost/games/id

Jogos e os verbos HTTP

Verbo	URL	Status Code	Situação
GET	/games	200 (OK)	Retorna os jogos
GET	/games/id	200 (OK)	Retorna o jogo
		404 (Not found)	O jogo não foi encontrado
POST	/games/	201 (Created)	Retorna o jogo criado
PUT	/games/id	200 (OK)	Atualiza e retorna o jogo atualizado
		400 (Bad request)	Se os dados do jogo não foram corretamente fornecidos
		404 (Not found)	Jogo não encontrado
DELETE	/games/id	204 (No content)	Jogo excluído
		404 (Not found)	Jogo não encontrado

Jogos e os verbos HTTP

- GET:
  - Aparece duas vezes em diferentes escopos;
  - Solicitação GET em http://localhost/games retorna a coleção de jogos em JSON;
  - GET em http://localhost/games/1 retorna o jogo cujo id é 1 em JSON
- POST:
  - Solicitação http://localhost/games + JSON: deve persistir um novo jogo;
  - Retorna status code 201 (CREATED);
  - Retorna também em JSON o jogo criado.

Jogos e os verbos HTTP

- PUT:
  - URL: http://localhost/games/{id} + JSON: deve alterar/substituir um jogo cujo id foi passado pelo conteúdo do jogo em JSON;
  - Caso tudo esteja ok, retorna o status code 200;
  - Caso os dados do novo jogo estejam incompletos, retorna o status 400;
  - Se o servidor não encontrar o jogo pelo id passado, o status 404 é retornado;

Jogos e os verbos HTTP

- DELETE:
  - Requisição na URL http://localhost/games/{id} deve excluir um jogo cujo id foi passado pelo conteúdo do jogo em JSON;
  - Caso tudo esteja ok, retorna o status code 204;
  - Se o servidor não encontrar o jogo pelo id passado, o status 404 é retornado;

Model

```
# gamesapi/games/models.py
from django.db import models
class Game(models.Model):
    created = models.DateTimeField(auto_now_add=True)
    name = models.CharField(max_length=200, blank=True,
                            default='')
    release_date = models.DateTimeField()
    game_category = models.CharField(max_length=200, blank=True,
                                     default='')
    played = models.BooleanField(default=False)

    class Meta:
        ordering = ('name',)
```

Configurações

```
# gamesapi/gamesapis/settings.py
INSTALLED_APPS = [
     #...
    'rest_framework',
    'games'
]
ROOT_URLCONF = '.urls'

# gamesapi/gamesapis/urls.py
from django.urls import path
from games import views

urlpatterns = [
    path('games/', views.game_list),
    path('games/<int:id>', views.game_detail)
]
```

Inserindo jogos

- Execute as migrações;
- Inclua pela linha de comando alguns jogos:

```
>>> python manage.py shell
from datetime import datetime
from django.utils import timezone
from games.models import Game

gamedatetime = timezone.make_aware(datetime.now(),
timezone.get_current_timezone())
game1 = Game(name='Smurfs Jungle', release_date=gamedatetime,
             game_category='2D mobile arcade', played=False)
game1.save()
game2 = Game(name='Angry Birds RPG', release_date=gamedatetime,
             game_category='3D RPG', played=False)
game2.save()
```

Serializers

- São os responsáveis por converter os models no formato escolhido para enviar ao cliente;
- Também fazem a conversão inversa (deserialização);
- Na prática, um serializer transforma um objeto em JSON e vice-versa;
- São classes que estendem as presentes no pacote rest_framework.serializers;
- As principais classes são:
  - Serializer;
  - ModelSerializer;
  - HiperlinkedModelSerializer.

GameSerializer

- Cada ModelSerializer possui uma classe meta que possui dois atributos:
  - Diz qual a classe model para a qual ele funciona;
  - Quais campos são utilizados;

```
# gamesapi/games/serializers.py
from rest_framework import serializers
from game.models import Game
class GameSerializer(serializers.ModelSerializer):
    class Meta:
        model = Game
        fields = ('id', 'name', 'release_date', 'game_category')
```

Views

- Assim como as views do Django, no DRF temos views para tratar as requisições e respostas;
- Podemos implementar views independentes do DRF, porém temos as seguintes views para extensão:
  - Anotações @**api_view**
  - Class based views;
  - GenericAPIView;
  - ModelViewSet.

Usando a notação @api_view

- Definimos funções isoladas para tratar os verbos HTTP;
- Em cada função, anotamos com a @api_view que verbos o método atende;
- Caso a função atenda a mais de um verbo, é necessário testar qual foi invocado.

```
@api_view(['GET', 'POST'])
def game_list(request):
    if request.method == 'GET':
        # ...
    elif request.method == 'POST':
        # ...
```

Implementando o GET (listagem)

- Passos:
  - Solicite todos os objetos do mecanismo ORM;
  - Instancie um serializer passando a QuerySet obtida;
  - Retorne um objeto Response tendo como parâmetro o objeto data do Serializer.
    - Este objeto é a representação em JSON do resultado;

View do GET

```
# gamesapi/games/views.py
from rest_framework.parsers import JSONParser
from rest_framework import status
from rest_framework.decorators import api_view
from rest_framework.response import Response
from games.models import Game
from games.serializers import GameSerializer

@api_view(['GET', 'POST'])
def game_list(request):
    if request.method == 'GET':
        games = Game.objects.all()
        games_serializer = GameSerializer(games, many=True)
        return Response(games_serializer.data)
    # ...
```

Implementando o POST

- Passos:
  - Instancie um objeto a partir da requisição:
    - Na requisição vem um JSON em request.data;
  - Teste se o serializer é válido e caso seja:
    - Delegue ao serializer através do método save() a responsabilidade de salvar o model;
    - Retorne o objeto recém-criado em JSON e o código 201 (CREATED);
  - Caso não seja válido, retorne um status 400 (Bad request).

View do POST

```
# gamesapi/games/views.py

@api_view(['GET', 'POST'])
def game_list(request):
    # ...

    elif request.method == 'POST':
        game_serializer = GameSerializer(data=request.data)
        if game_serializer.is_valid():
            game_serializer.save()
            return Response(game_serializer.data,
                            status=status.HTTP_201_CREATED)
        return Response(game_serializer.errors,
                        status=status.HTTP_400_BAD_REQUEST)
```

Implementando o GET (único objeto)

- Passos:
  - Consulte o objeto do mecanismo ORM a partir do id passado pela URL;
  - Caso o objeto não exista, retorne um código 404 (Not found)
  - Do contrário, instancie um serializer a partir do objeto;
  - Retorne um objeto Response tendo como parâmetro o atributo data do Serializer.

View do GET

```
# gamesapi/games/views.py (cont.)
@api_view(['GET', 'PUT', 'DELETE' ])
def game_detail(request, pk):
    try:
        game = Game.objects.get(pk=pk)
    except Game.DoesNotExist:
        return Response(status=status.HTTP_404_NOT_FOUND)

    if request.method == 'GET':
        game_serializer = GameSerializer(game)
        return Response(game_serializer.data)

    # ...
```

Implementando o PUT

- Passos:
  - Consulte o objeto do mecanismo ORM a partir do id passado pela URL;
  - Caso o objeto não exista, retorne um código 404 (Not found)
  - Do contrário, instancie um serializer a partir do objeto passa do no request em JSON;
  - Teste se o serializer é válido e caso seja:
    - Delegue ao serializer através do método save() a responsabilidade de salvar o model;
    - Retorne um objeto Response tendo como parâmetro o atributo data do Serializer;
  - Caso não seja válido, retorne um status 400 (Bad request).

View do PUT

```
# gamesapi/games/views.py (cont.)
@api_view(['GET', 'PUT', 'POST', 'DELETE' ])
def game_detail(request, pk):
    try:
        game = Game.objects.get(pk=pk)
    except Game.DoesNotExist:
        return Response(status=status.HTTP_404_NOT_FOUND)
    # ...
    elif request.method == 'PUT':
        game_serializer = GameSerializer(game, data=request.data)
        if game_serializer.is_valid():
            game_serializer.save()
            return Response(game_serializer.data)
        return Response(game_serializer.errors, status=status.HTTP_400_BAD_REQUEST)

    # ...
```

Implementando o DELETE

- Passos:
  - Consulte o objeto do mecanismo ORM a partir do id passado pela URL;
  - Caso o objeto não exista, retorne um código 404 (Not found) Django;
  - Retorne um objeto Response com o status 204 (No content).

View de GET, PUT e DELETE

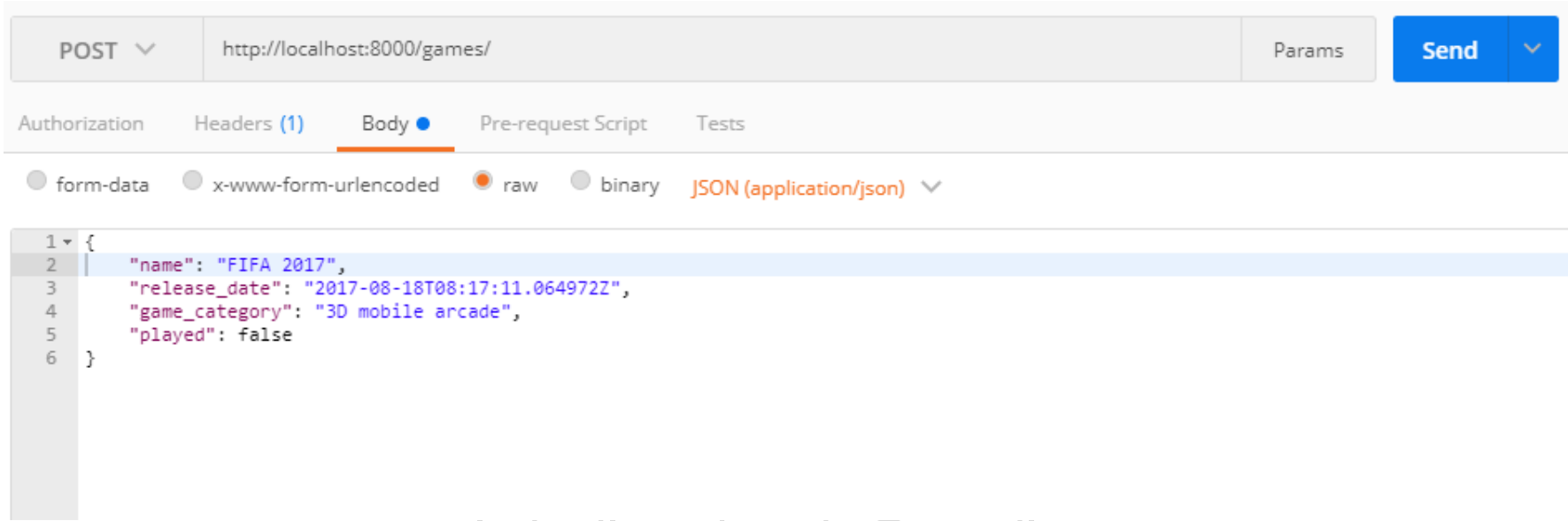
```
# gamesapi/games/views.py (cont.)
@api_view(['GET', 'PUT', 'POST', 'DELETE'])
def game_detail(request, pk):
    try:
        game = Game.objects.get(pk=pk)
    except Game.DoesNotExist:
        return Response(status=status.HTTP_404_NOT_FOUND)
    # ...
    elif request.method == 'DELETE':
        game.delete()
        return Response(status=status.HTTP_204_NO_CONTENT)
    # ...
```

Testando a API

- Rode o nosso servidor:

  >>> python manage.py runserver

- Teste a api a partir de http://localhost:8000/games/
- Utilize: A WEB View, curl ou postman.



Atividade

- Pesquise qual o melhor padrão para validações extras em no DRF. Na própria view ou no serializer?
- Faça as seguintes validações na API proposta:
  - [POST, PUT]: Jogos não podem ter campos vazios;
  - [POST, PUT]: Jogos não podem ter nomes repetidos;
  - [DELETE]: Somente jogos que ainda não foram lançados podem ser excluídos;
- Que status codes você usaria? Ou usaria os mesmos? Pesquise e implemente;
- Retorne também uma informação descritiva ao retornar um erro de validação.

Entrega

- Individual;
- Prazo: para a próxima aula;
- Vale até 0,5 ponto;
- Entrega via commit do código da atividade e publicação do link do github no gurpo do telegram;

WEB APIs

Django Rest Framework, Verbos HTTP, Status codes, serializadores e API Views

Ely – elydasilvamiranda@gmail.com