WEB APIs

Class-based views e Hiperlinked API

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.

Principais pontos da aula

- Trabalhar com wrappers para escrever API views;
- Declarar relacionamentos entre os models do estudo de caso de jogos;
- Gerenciar serialização e deserialização com relacionamentos e hyperlinks na API;
- Criar class-based views e usar classes genéricas;
- Criar e listar recursos relacionados;

Estudo de caso (relembrando)

- 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.

Expandindo os models e API

- Fizemos uma API para uma tabela simples;
- Precisamos criar uma API para um modelo mais complexo/realista:
  - Registrar os escores dos jogadores para jogos que são agrupados por categoria;
  - Precisamos criar e relacionar mais models;
  - Anteriormente usamos a categoria de um jogo como uma string;
  - Agora precisamos "puxar" todos os jogos de uma categoria, o que gera mais um relacionamento.

Novo modelo

- GameCategory;
- Game;
- Player;
- Score.

Relacionamentos

- Game:
  - id;
  - name;
  - release_date;
  - created;
  - played;
  - game_category: foreign key para GameCategory

Relacionamentos

- Player:
  - id;
  - name;
  - gender;
  - created.

Relacionamentos

- Score:
  - id;
  - score;
  - score_date;
  - game: foreing key para Game;
  - player: foreing key para Player;

Jogos x HTTP

Verbo	URL	Semântica
GET	/games	Obtém todos os jogos ordenados pelo seu nome. Cada jogo deve conter a descrição de sua categoria.
GET	/games/{id}	Obtém um jogo. O jogo deve conter a descrição da sua categoria.
POST	/games	Cria um novo jogo na coleção.
PUT	/games/{id}	Atualiza um jogo.
PATCH	/games/{id}	Atualiza um ou mais campos de um jogo.
DELETE	/game/{id}	Exclui um jogo existente.

Categorias x HTTP

Verbo	URL	Semântica
GET	/game-categories/	Recupera todas as categorias de jogos, ordenadas pelo nome em ordem crescente. Cada categoria deve incluir ainda uma lista de URLs para cada jogo que pertença à categoria.
GET	/game-categories/{id}	Recupera uma categoria de jogo. A categoria deve incluir ainda uma lista de URLs para cada jogo que pertença à categoria.
POST	/game-categories/	Cria uma nova categoria
PUT	/game-categories/{id}	Atualiza uma categoria
PATCH	/game-categories/{id}	Atualiza um ou mais campos de uma categoria
DELETE	/game-categories/{id}	Exclui uma categoria

Jogadores x HTTP

Verbo	URL	Semântica
GET	/players/	Recupera todos os jogadores, ordenados pelo nome em ordem crescente. Cada jogador deve incluir uma lista escores, ordenados em ordem decrescente de pontuação. A lista deve incluir ainda todos os detalhes do escore obtido e o nome do jogo relacionado.
GET	/players/{id}	Recupera um jogador. O jogador deve incluir deve incluir ainda uma lista escores, ordenados em ordem decrescente de pontuação. A lista deve incluir todos os detalhes do escore obtido e nome do jogo relacionado.
POST	/players/	Cria um novo jogador
PUT	/players/{id}	Atualiza um jogador
PATCH	/players/{id}	Atualiza um ou mais campos de um jogador
DELETE	/players/{id}	Exclui um jogador

Escores x HTTP

Verbo	URL	Semântica
GET	/scores/	Recupera todos os escores, ordenados pelo escore em ordem decrescente. Cada escore deve incluir ainda os nomes do jogador e do jogo.
GET	/scores/{id}	Recupera um escore. O escore deve incluir ainda os nomes do jogador e do jogo.
POST	/scores/	Cria um novo escore
PUT	/scores/{id}	Atualiza um escore
PATCH	/scores/{id}	Atualiza um ou mais campos de um escore
DELETE	/scores/{id}	Exclui um escore

Sobre o PUT, POST e PATCH

- Fica subtendido sobre esses métodos o seguinte:
  - PUT e POST: recebem um JSON representando um objeto específico;
  - PATCH: recebe um JSON representando parte de um objeto específico a ser atualizado;
  - Todos retornam os objetos persistidos refletindo as alterações;

Os models

```
# gamesapi/games/models.py
class GameCategory(models.Model):
    name = models.CharField(max_length=200)

    class Meta:
        ordering = ('name',)

    def __str__(self):
        return self.name
```

Os models

```
# gamesapi/games/models.py
class Game(models.Model):
    created = models.DateTimeField(auto_now_add=True)
    name = models.CharField(max_length=200)
    game_category = models.ForeignKey(
        GameCategory,
        related_name='games',
        on_delete=models.CASCADE)
    release_date = models.DateTimeField()
    played = models.BooleanField(default=False)

    class Meta:
        ordering = ('name',)

    def __str__(self):
        return self.name
```

Os models

```
# gamesapi/games/models.py
class Player(models.Model):
    MALE = 'M'
    FEMALE = 'F'
    GENDER_CHOICES = ((MALE, 'Male'),(FEMALE, 'Female'),)
    created = models.DateTimeField(auto_now_add=True)
    name = models.CharField(max_length=50, blank=False)
    gender = models.CharField(max_length=2,
                              choices=GENDER_CHOICES,
                              default=MALE,)

    class Meta:
        ordering = ('name',)

    def __str__(self):
        return self.name
```

Os models

```
# gamesapi/games/models.py
class Score(models.Model):
    player = models.ForeignKey(Player,
                               related_name='scores',
                               on_delete=models.CASCADE)
    game = models.ForeignKey(Game,
                             on_delete=models.CASCADE)
    score = models.IntegerField()
    score_date = models.DateTimeField()

    class Meta:
        ordering = ('-score',)
```

Importante

Rode as migrações.

Serialização do modelo

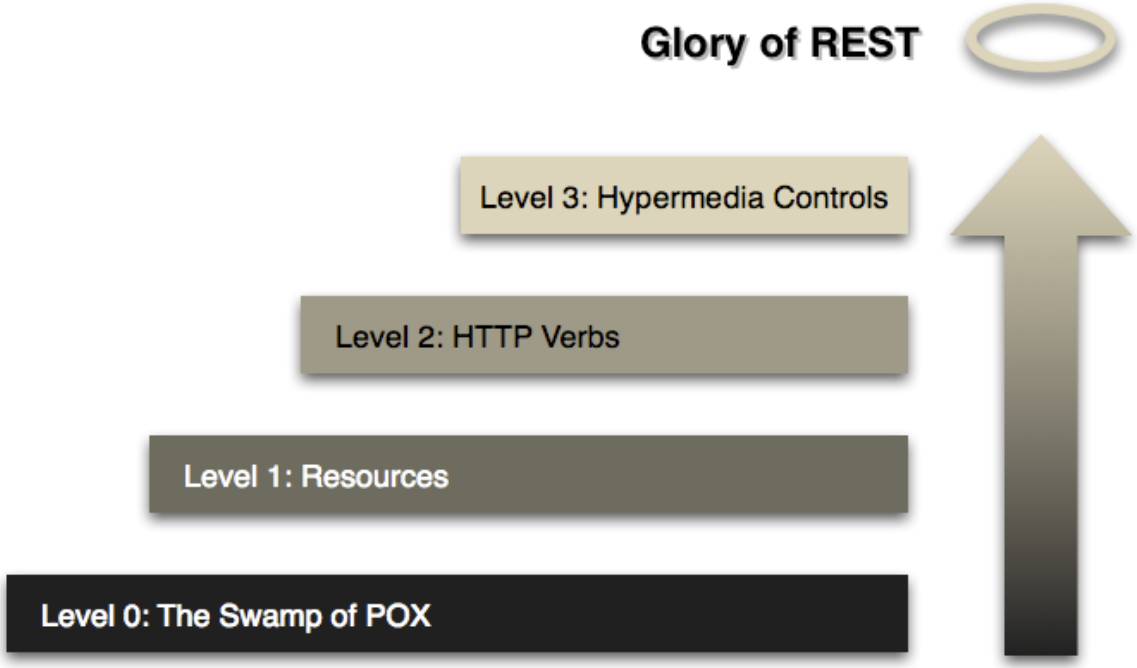
- A serialização deve contemplar também com os relacionamentos;
- Anteriormente usamos a classe ModelSerializer do pacote rest_framework.serializers;
- Os serializadores utilizarão agora a classe HyperlinkedModelSerializer;
  - Utiliza relacionamentos "hiperlinkados" em vez do relacionamento via chave primária;
  - Para isso, há a criação de um atributo 'url' com o caminho para o recurso referenciado;

Por que HyperlinkedModelSerializer?

- HATEOAS: Hypermedia as the Engine of Application State;
- É uma constraint arquitetural de aplicações REST;
- Provê informações que permitem navegar entre seus endpoints de forma dinâmica;
- Uma API para ser RESTful, deve incluir links junto às respostas para navegação entre entidades;

RESTful

- O **Richardson Maturity Model** define que HATEOAS é o último nível de maturidade de uma API:
  - https://martinfowler.com/articles/richardsonMaturityModel.html
  - http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven



GameCategorySerializer

- Nosso requisito:

  "Cada categoria deve incluir ainda uma lista de URLs para cada jogo que pertença à categoria";

```
# gamesapi/games/serializers.py
class GameCategorySerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
        model = GameCategory
        fields = ('url', 'pk', 'name', 'games')
```

GameSerializer

```
# gamesapi/games/serializers.py
class GameSerializer(serializers.HyperlinkedModelSerializer):

    class Meta:
        model = Game
        fields = (
            'url',
            'game_category',
            'name',
            'release_date',
            'played'
        )
```

GameSerializer

- Esta classe antes tinha o atributo categoria definido como uma string;
- Agora, ela tem um relacionamento com **GameCategory**;
- Nosso requisito:

  "Cada jogo deve conter a descrição de sua categoria";

- Se apenas deixarmos como está o serializer, ele exibirá um link para a categoria:

```
{
    "url": "http://localhost/games/1/",
    "game_category": "http://localhost/game-categories/3/",
    "name": "FIFA 17",
    "release_date": "2017-08-07T00:00:00Z",
    "played": true
},
```

GameSerializer

- Podemos criar um campo "slug" no serializer que para cada jogo busque sua categoria e exiba apenas o nome;
- Um campo do tipo SlugRelatedField tem com parâmetros:
  - Uma QuerySet que informa como consultar o objeto;
  - O nome do campo que será exibido.

GameSerializer

```
# gamesapi/games/serializers.py
class GameSerializer(serializers.HyperlinkedModelSerializer):
      game_category =
         serializers.SlugRelatedField(queryset=GameCategory.objects.all(),
                                      slug_field='name')
       class Meta:
         model = Game
         fields = (
             'url',
             'game_category',
             'name',
             'release_date',
             'played'
         )
```

```
{
    "url": "http://localhost/games/1/",
    "game_category": "Esporte",
    "name": "FIFA 17",
    "release_date": "2017-08-07T00:00:00Z",
    "played": true
},
```

ScoreSerializer

```
# gamesapi/games/serializers.py
class ScoreSerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
        model = Score
        fields = (
            'url',
            'pk',
            'score',
            'score_date',
            'player',
            'game',
            )
```

ScoreSerializer

- Nosso requisito:

  "Cada escore deve exibir os nomes do jogador e do jogo";

- Da forma como está o serializer, serão exibidos apenas os links para jogador e jogo;
- Para exibirmos os dados adicionaremos campos 'slug' atributo.

ScoreSerializer

```
# gamesapi/games/serializers.py
class ScoreSerializer(serializers.HyperlinkedModelSerializer):
    game =
        serializers.SlugRelatedField(queryset=Game.objects.all(),
                                     slug_field='name')
    player =
        serializers.SlugRelatedField(queryset=Player.objects.all(),
                                     slug_field='name')
    class Meta:
        model = Score
        fields = (
            'url',
            'pk',
            'score',
            'score_date',
               'player',
            'game',
            )
```

PlayerSerializer

- Nosso requisito:

  "Cada jogador deve listar uma coleção de escores, ordenados em ordem decrescente de pontuação."

- Devemos declarar um campo para trazer os escores como sendo o ScoreSerializer:
  - Como esse campo poderá trazer vários scores, o atributo many deve ser setado para True;
  - Ela deve ser uma propriedade readonly, para que alterações no jogador não reflitam diretamente nos escores.

PlayerSerializer

```
# gamesapi/games/serializers.py
class PlayerSerializer(serializers.HyperlinkedModelSerializer):
    scores = ScoreSerializer(many=True, read_only=True)


    class Meta:
        model = Player
        fields = (
            'url',
            'name',
            'gender',
            'scores',
            )
```

Views e classes genéricas

- Anteriormente:
  - Utilizamos a anotação @api_view;
  - Implementamos o tratamento de cada método;
- Quando pensamos em uma API com vários recursos, fica repetitivo escrever daquela forma;
- Podemos utilizar Views genéricas para o comportamento padrão;
- Nos casos em que o tratamento for muito específico, podemos sobrescrever métodos específicos.

Classes genéricas do DRF

- Em rest_framework.generics estão declaradas:
  - ListCreateAPIView: implementa o GET (listagem) e o POST;
  - RetrieveUpdateDestroyAPIView: Implementa os métodos GET (único objeto), PUT, PATCH, DELETE;
- Elas são formadas pela herança múltipla de outros recursos pré-implementados do DRF.

Classes genéricas do DRF

- A classe ListCreateAPIView é declarada como: implementa o GET (listagem) ae o POST;

```
class ListCreateAPIView(mixins.ListModelMixin,
                        mixins.CreateModelMixin,
                        GenericAPIView):
```

- Já a classe RetrieveUpdateDestroyAPIView:

```
class RetrieveUpdateDestroyAPIView(
            mixins.RetrieveModelMixin,
            mixins.UpdateModelMixin,
            mixins.DestroyModelMixin,
            GenericAPIView):
```

https://github.com/encode/django-rest-framework/blob/master/rest_framework/generics.py

Generic Views

- Utilizaremos as classes anteriores para cada um dos models do sistema;
- Para usar uma view genérica precisamos fornecer:
  - Uma QuerySet para resgatar os models;
  - Um serializer;
  - E um nome/"apelido" para a view;
- Caso necessário, os métodos referentes aos métodos GET, POST... Podem ser sobrescritos.

Generic views

```
# gamesapi/games/views.py
class GameCategoryList(generics.ListCreateAPIView):
    queryset = GameCategory.objects.all()
    serializer_class = GameCategorySerializer
    name = 'gamecategory-list'

class GameCategoryDetail(generics.RetrieveUpdateDestroyAPIView):
    queryset = GameCategory.objects.all()
    serializer_class = GameCategorySerializer
    name = 'gamecategory-detail'

class GameList(generics.ListCreateAPIView):
    queryset = Game.objects.all()
    serializer_class = GameSerializer
    name = 'game-list'

class GameDetail(generics.RetrieveUpdateDestroyAPIView):
    queryset = Game.objects.all()
    serializer_class = GameSerializer
    name = 'game-detail'
```

Generic views

```
# gamesapi/games/views.py
class PlayerList(generics.ListCreateAPIView):
    queryset = Player.objects.all()
    serializer_class = PlayerSerializer
    name = 'player-list'

class PlayerDetail(generics.RetrieveUpdateDestroyAPIView):
    queryset = Player.objects.all()
    serializer_class = PlayerSerializer
    name = 'player-detail'

class ScoreList(generics.ListCreateAPIView):
    queryset = Score.objects.all()
    serializer_class = ScoreSerializer
    name = 'score-list'

class ScoreDetail(generics.RetrieveUpdateDestroyAPIView):
    queryset = Score.objects.all()
    serializer_class = ScoreSerializer
    name = 'score-detail'
```

Definindo um endpoint

- Podemos definir um endpoint para a raiz da nossa API: http://localhost/
- Para isso, basta sobrescrever o método get da classe generics.GenericAPIView;
- A resposta desse método deve ser um "JSON" com:
  - Chaves: o nome dos recursos;
  - Valores: o link para a listagem dos recursos.

Definindo um endpoint

```
# gamesapi/games/views.py
class ApiRoot(generics.GenericAPIView):
    name = 'api-root'
    def get(self, request, *args, **kwargs):
        return Response({
            'players': reverse(PlayerList.name,
                               request=request),
            'game-categories': reverse(GameCategoryList.name,
                                       request=request),
            'games': reverse(GameList.name,
                             request=request),
            'scores': reverse(ScoreList.name,
                              request=request)
            })
```

urls.py

```
urlpatterns = [
    path('games/', views.GameList.as_view(),
         name=views.GameList.name),
    path('games/<int:pk>/', views.GameDetail.as_view(),
         name=views.GameDetail.name),
    path('game-categories/', views.GameCategoryList.as_view(),
         name=views.GameCategoryList.name),
    path('game-categories/<int:pk>/', views.GameCategoryDetail.as_view(),
         name=views.GameCategoryDetail.name),
    path('players/', views.PlayerList.as_view(),
         name=views.PlayerList.name),
    path('players/<int:pk>/', views.PlayerDetail.as_view(),
         name=views.PlayerDetail.name),
    path('scores/', views.ScoreList.as_view(),
         name=views.ScoreList.name),
    path('scores/<int:pk>/', views.ScoreDetail.as_view(),
         name=views.ScoreDetail.name),
    path('', views.ApiRoot.as_view(),
         name=views.ApiRoot.name),

]
```

Exercício

Após implementar os exemplos dos slides, responda as seguintes questões:

1. Proponha validações do modelo na entidade Score, usando prioritariamente validações nos serializers:
   - Player e game não selecionado;
   - Score negativo ou não preenchido;
   - Data do score no futuro.
2. É possível simplificar mais ainda as views usando o DRF, visto que o código ficou repetitivo? Proponha uma alternativa.

WEB APIs

Class-based views e Hiperlinked API

Ely – elydasilvamiranda@gmail.com