



Școala  
informală  
de IT

Google

# Python Development

Django REST Framework



Școala  
informală  
de IT

Google

# Cuprins

1. REST API
2. Django REST Framework
3. Serializers & ViewSets
4. ModelSerializers & ModelViewSets
5. Serializarea relațiilor
6. Validarea datelor
7. Autentificare - JWT
8. DRF & JWT



Scoala  
informală  
de IT

Google

Django REST Framework

# REST API

Django REST Framework



Scoala  
informală  
de IT



# REST API

## API - Application Programming Interface

- Este o interfață care definește interacțiunea dintre mai mulți intermediari software.
- Definește:
  - tipul de request-uri care pot fi făcute
  - modul în care trebuie făcut request-ul
  - formatul datelor care trebuie folosit în cadrul request-ului
  - etc.
- Datorită creșterii din punct de vedere al popularității REST-ului și serviciilor web, acestora li se mai asociază termenul de API.
- Scopul API-ului este să simplifice programarea prin abstractizarea implementării și expunerea numai a acelor obiecte și acțiuni necesare programatorului.
- *Exemplu* - Un API pentru gestionarea unor fișiere poate oferi programatorului o funcție care copiază un fișier dintr-o parte în alta fără a necesita înțelegerea sistemului de copiere ce intervene la nivelul sistemului de fișiere.



Scoala  
informala  
de IT



# REST API

## REST - REpresentational State Transfer

- Este un stil de arhitectură pentru sistemele distribuite.
- A fost prezentat pentru prima dată de Roy Fielding în lucrarea sa de dizertație din anul 2000.
- Are la bază 6 principii care trebuie respectate pentru a putea spune că o interfață este **RESTful**:
  1. **Client-server**. Prin separarea responsabilităților ce țin de interfața de utilizare (UI) de cele necesare stocării și prelucrării datelor se îmbunătățește portabilitatea interfețelor de utilizare pe mai multe platforme și se îmbunătățește scalabilitatea prin simplificarea componentelor server-ului.
  2. **Stateless**. Fiecare request făcut de client trebuie să conțină toate informațiile necesare îndeplinirii acestuia. Deci nu se poate folosi de nici un context stocat pe server, ceea ce înseamnă că aplicația client este cea care trebuie să se ocupe de stocarea sesiunilor.
  3. **Cacheable**. Datele primite în urma unui request pot fi marcate implicit sau explicit ca fiind cache-uibile. Dacă sunt marcate ca fiind cache-uibile, aplicația client are dreptul de a reutiliza aceste date ca răspuns al unui request echivalent ulterior.

# REST API

4. **Uniform interface.** Prin aplicare principiului generalității din ingineria software, arhitectura sistemului este simplificată și vizibilitatea interacțiunilor este îmbunătățită. Pentru a obține o interfață uniformă, mai multe constrângeri arhitecturale sunt necesare pentru a defini comportamentul componentelor. REST este definit de 4 constrângeri:
  - Identificarea resurselor - URI
  - Gestionarea resurselor - metode HTTP
  - Mesaje descriptive - MIME types
  - Hypermedia ca motorul gestionării stării aplicației - folosirea hyperlink-urilor.
5. **Layered system.** Stilul bazat pe straturi permite unei arhitecturi să fie compusă din straturi ierarhice. Astfel fiecare componentă este constrânsă să nu “vadă” cu cine interacționează mai departe de primul strat.
6. **Code on demand (optional).** REST permite extinderea funcționalității clientului prin download-ul și executarea codului sub formă de applets sau script-uri.

# REST API

- Resursa reprezintă metoda cheie de abstractizare a informației într-o arhitectură REST.
- Orice informație care poate fi numită poate fi o resursă: un document, o imagine, o colecție de alte resurse etc.
- Pentru identificarea unei resurse, REST folosește un identificator (resource identifier) unic. Acesta este un URL al resursei, cunoscut și sub numele de **endpoint**. De altfel, URL este acronimul de la **Uniform Resource Locator**.
- Fiecare mod de a interacționa cu o resursă este cunoscut sub numele de metodă. În majoritatea cazurilor metodele de baza sunt:
  - GET - interogarea unei resurse.
  - POST - crearea unei resurse noi.
  - PUT - modificarea unei resurse existente.
  - DELETE - ștergerea unei resurse.



# Django REST Framework

Django REST Framework

# Django REST Framework

- Fiind un nume foarte lung vom folosi următoarea prescurtare:

**Django REST Framework - DRF**

- Este un pachet Python care ne oferă suport pentru implementarea unui REST API folosind framework-ul nostru web preferat. Pentru intalare vom folosi:

**pip install djangorestframework**

- Trebuie să adăugăm **rest\_framework** la lista de aplicații instalate.
- Acest pachet vine cu Browsable API care ne permite foarte ușor să gestionăm API-ul nostru din browser.
- Pentru a putea folosi inclusiv rutele care necesită logare va trebui să adăugăm următoarea linie în variabila **urlpatterns** din **urls.py**:

```
path('api-auth/', include('rest_framework.urls'))
```

# Django REST Framework

- Primul pas în implementarea API-ului este definirea URL-urilor și a router-ului.

```
from django.urls import path, include  
from rest_framework import routers  
  
router = routers.DefaultRouter()  
  
urlpatterns = [  
    path('', include(router.urls)),  
    path('api-auth/', include('rest_framework.urls'))  
]
```

- Putem să folosim propriile URL-uri sau putem folosi router-ul din DRF. Routerele disponibile sunt:
  - SimpleRouter
  - DefaultRouter
  - CustomRouter
- Acestea se găsesc în **rest\_framework.routers**
- În momentul acesta putem urmări API-ul nostru în browser cu ajutorul *Browsable API*.



# Serializers & Viewsets

Django REST Framework

# Serializers & Viewsets

- În contextul transmiterii datelor către client avem nevoie de serializarea datelor Python.
- Serializarea presupune transpunerea obiectelor Python în date ce pot fi trimise prin intermediul internetului.
- Majoritatea API-urilor trimit datele în format JSON (**JavaScript Object Notation**).
- DRF Serializers permit conversia datelor complexe cum ar fi QuerySet-urile și instanțe ale modelelor în date Python care pot fi mai departe serializate ușor în JSON, XML sau alte tipuri de conținut.
- DRF Serializers permit și deserializarea datelor, permitând datelor parsate să fie converte în date complexe (acest proces presupune o validare prealabilă a datelor).

# Serializers & Viewsets

```
from django.contrib.auth import get_user_model
from rest_framework import serializers

UserModel = get_user_model()

class UserSerializer(serializers.Serializer):
    id = serializers.IntegerField()
    first_name = serializers.CharField()
    last_name = serializers.CharField()
    email = serializers.EmailField()

    def create(self, validated_data):
        pass

    def update(self, instance, validated_data):
        pass
```

- Cel mai simplu serializer pe care îl putem scrie este pentru modelul **User**.
- Orice subclasă a clasei **serializers.Serializer** trebuie să implementeze metodele **create** și **update**.
- Atributele ce se doresc a fi serializate trebuie definite în mod explicit ca atribute a acestei clase.



# Serializers & Viewsets

- Un **ViewSet** este un view definit ca o clasă care implementează o listă de metode cu ajutorul cărora pot fi gestionate resursele:
  - **list** - oferă toate resursele de un anumit tip
  - **create** - creează o resursă de un anumit tip
  - **retrieve** - oferă o anumită resursă de un anumit tip pe baza unui ID
  - **update** - modifică o resursă de un anumit tip
  - **partial\_update** - modifică parțial o resursă
  - **destroy** - șterge o resursă de un anumit tip



# Serializers & Viewsets

```
from rest_framework.response import Response
from django.shortcuts import get_object_or_404

class UsersViewSet(viewsets.ViewSet):
    def list(self, request):
        queryset = UserModel.objects.all()
        serializer = UserSerializer(queryset, many=True)
        return Response(serializer.data)

    def retrieve(self, request, pk=None):
        user = get_object_or_404(UserModel, pk=pk)
        serializers = UserSerializer(user)
        return Response(serializers.data)
```

- Cel mai simplu view pe care îl putem crea este pentru a obține o listă de utilizatori și un anumit utilizator bazat pe ID-ul acestuia.
- Pentru asta trebuie să moștenim clasa `viewsets.ViewSet` și să implementăm metodele `list` și `retrieve`.



# ModelSerializers & ModelViewsets

Django REST Framework



# ModelSerializers & ModelViewsets

- Uneori avem nevoie să serializăm modelele Django, astfel putem folosi clasa **ModelSerializer** care știe să mapeze atributele modelelor Django.
- Această clasă oferă și validare conformă cu fiecare atribut al modelului.

```
class UserSerializer(serializers.ModelSerializer):  
    class Meta:  
        model = UserModel  
        exclude = ['password']
```

- A se observa maparea cu modelul **UserModel** și definirea atributelor care trebuie folosite sau excluse (după caz).
- În cazul modelului User trebuie exclus câmpul **password** din motive evidente.



# ModelSerializers & ModelViewsets

- Un **ModelViewSet** este un **ViewSet** care știe să mapeze comportamentul modelelor Django.
- Următoarele 2 atribute sunt obligatorii:
  - **queryset** - se ocupă de gestionarea datelor
  - **serializer\_class** - se ocupă de serializarea datelor

```
class UsersViewSet(viewsets.ModelViewSet):  
    queryset = UserModel.objects.all()  
    serializer_class = UserSerializer
```



# Serializarea relațiilor

Django REST Framework

# Serializarea relațiilor

- Fiecare obiect pe care vrem să-l serializăm are nevoie de un **Serializer**.
- Considerăm următorul model **Publisher**:

```
class Publisher(models.Model):  
    owner = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.CASCADE, related_name='publishers')  
    name = models.CharField(unique=True, max_length=255)  
  
    def __str__(self):  
        return self.name
```

- O să scriem un Serializer pentru modelul **Publisher** care va arăta astfel:

```
class PublisherSerializer(serializers.ModelSerializer):  
    class Meta:  
        model = Publisher  
        fields = ['id', 'url', 'owner', 'name']
```

- În exemplul acesta, câmpul **owner** reprezintă un utilizator care este proprietarul publicației.
- View-ul aferent modelului **Publisher** este:

```
class PublisherViewSet(viewsets.ModelViewSet):  
    queryset = Publisher.objects.all()  
    serializer_class = PublisherSerializer
```



# Serializarea relațiilor

- Prin accesarea URL-ului pentru resursa **publishers** vom obține următoarele date:

```
[{"id": 75,  
 "url": "http://localhost:8000/api/publishers/75/",  
 "owner": 142,  
 "name": "Kit-Kat Ø Publisher"}]
```

- Din căte se poate observa, atributul **owner** conține ID-ul utilizatorului, dar nu conține alte detalii despre acesta.
- Pentru a obține aceste informații avem două opțiuni:
  - implicită - prin folosirea atributului **depth**
  - explicită - prin declararea câmpurile care reprezintă o relație a modelului nostru.



# Serializarea relațiilor

- Pentru varianta implicită trebuie să definim atributul **depth** în clasa **Meta** a serializer-ului nostru.

```
class PublisherSerializer(serializers.ModelSerializer):  
    class Meta:  
        model = Publisher  
        fields = ['id', 'url', 'owner', 'name']  
        depth = 1
```

- Valoarea setată atributului reprezintă numărul de nivele ale relațiilor care trebuie parseate înainte de a fi serializate datele.

```
{  
    "id": 76,  
    "url": "http://localhost:8000/api/publishers/76/",  
    "owner": {  
        "id": 143,  
        "password": "pbkdf2_sha256$180000$leHhJwMBzxJ5$EeBNSobpCEqiR+/7yQrAPCIeSu+IWty40UfI+Ey5aVE=",  
        "last_login": null,  
        "is_superuser": false,  
        "first_name": "Nickolas",  
        "last_name": "Strickland",  
        "is_staff": true,  
        "is_active": true,  
        "date_joined": "2020-05-28T18:50:46.837595Z",  
        "email": "publisher_account1@gmail.com",  
        "groups": [  
            8  
        ],  
        "user_permissions": []  
    },  
    "name": "Thor 1 Publisher"  
}
```



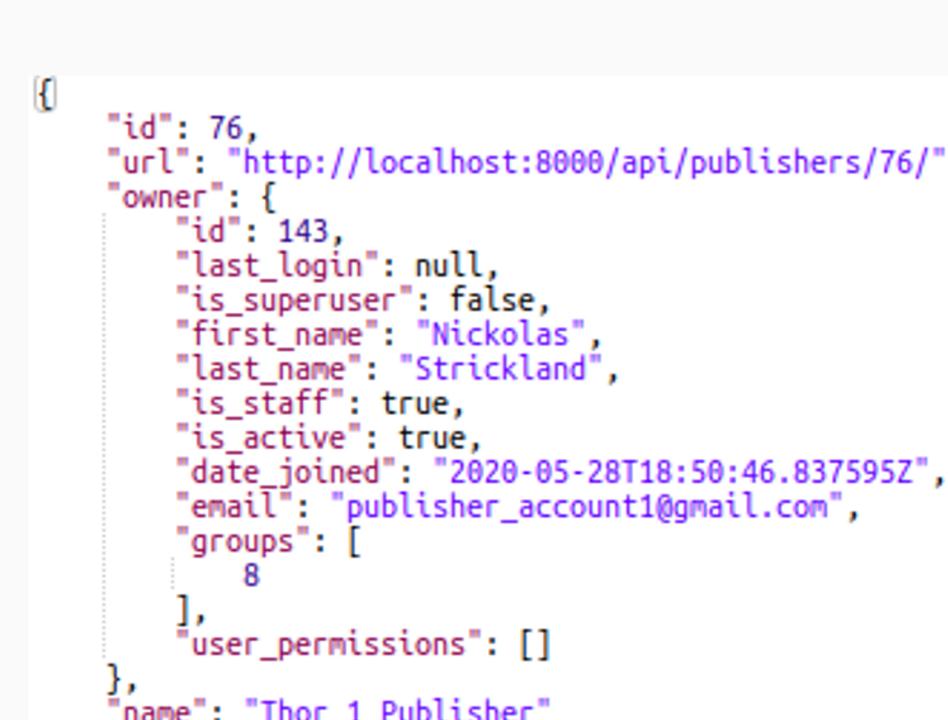
# Serializarea relațiilor

- Varianta explicită reprezintă definirea unui atribut care să aibă același nume ca al relației. Acestui atribut îl se va asocia un serializer care știe să serializeze datele respective.
- În cazul acesta vom avea control asupra modului de serializare a datelor modelului User cu ajutorul UserSerializer-ului definit anterior.

```
class PublisherSerializer(serializers.ModelSerializer):
    owner = UserSerializer()

    class Meta:
        model = Publisher
        fields = ['id', 'url', 'owner', 'name']

    def to_representation(self, instance):
        data = super().to_representation(instance)
        data['owner'] = UserSerializer(instance.owner).data
        return data
```



```
{
    "id": 76,
    "url": "http://localhost:8000/api/publishers/76/",
    "owner": {
        "id": 143,
        "last_login": null,
        "is_superuser": false,
        "first_name": "Nickolas",
        "last_name": "Strickland",
        "is_staff": true,
        "is_active": true,
        "date_joined": "2020-05-28T18:50:46.837595Z",
        "email": "publisher_account1@gmail.com",
        "groups": [
            8
        ],
        "user_permissions": []
    },
    "name": "Thor 1 Publisher"
}
```



# Validarea datelor

Django REST Framework



Scoala  
informală  
de IT

Google

# Validarea datelor

- De fiecare dată când deserializăm datele, acestea trebuie validate.
- Același principiu se aplică și la validarea unui form, înainte de a putea utiliza datele din form.
- Fiecare serializer dispune de metoda **is\_valid()** care are rolul să valideze datele primite. În cazul în care validarea nu trece, erorile rezultate se vor regăsi în atributul **errors**.
- Pentru validarea unui serializer avem două variante:
  - validare la nivel de obiect
  - validare la nivel de câmp
- Dacă există erori neasociate cu unul din câmpuri, acestea se vor regăsi în obiectul **non\_field\_errors**. Denumirea acestuia poate fi schimbată prin setarea DRF **NON\_FIELD\_ERRORS\_KEY**.



# Validarea datelor

```
def validate(self, data):
    errors = {}

    email = data.get('email', None)
    if not email:
        errors['email'] = ['E-mail is mandatory']

    first_name = data.get('first_name', None)
    if not first_name:
        errors['first_name'] = ['First name is mandatory.']

    last_name = data.get('last_name', None)
    if not last_name:
        errors['last_name'] = ['Last name is mandatory.']

    if len(errors) > 0:
        raise serializers.ValidationError(errors)

    return data
```

- Folosind validarea la nivel de obiect ne ajută să gestionăm câmpurile care nu sunt definite explicit.



# Validarea datelor

```
class RegisterSerializer(serializers.Serializer):
    email = serializers.EmailField(required=True)
    first_name = serializers.CharField(max_length=255, required=True)
    last_name = serializers.CharField(max_length=255, required=True)

    def validate_email(self, email):
        is_unique_email = UserModel.objects.filter(email=email).count() == 0

        if not is_unique_email:
            raise serializers.ValidationError('E-mail already taken.')

        return email

    def create(self, validated_data):
        user = UserModel.objects.create_user(
            email=validated_data['email'],
            first_name=validated_data['first_name'],
            last_name=validated_data['last_name']
        )

        return user
```

- Validarea la nivel de câmp este făcută în primul rând prin modul în care sunt definite câmpurile. În cazul din imaginea toate câmpurile sunt obligatorii, altfel datele vor fi invalide.
- Funcțiile de `validate_<field_name>` vor fi apelate abia după acest pas.



# Autentificare - JWT

Django REST Framework

# Autentificare - JWT

- Care este diferența dintre autentificare și autorizare?
- **Autentificarea** este atunci când un utilizator își dovedește identitatea. Prin autentificare dovedești că ești cine pretinzi că ești.
- **Autorizarea** este atunci când un utilizator dovedește că are dreptul să întreprindă o anumită acțiune.



- **JWT** este acronimul de la **JSON Web Token**.
- Este un standard care definește un mod compact și implicit de transmitere securizată a informației sub forma obiectelor JSON.
- Informația poate fi verificată și credibilă pentru că este semnată digital.
- Un JWT poate fi semnat folosind un algoritm HMAC sau o cheie publică/privată (RSA sau ECDSA).

# Autentificare - JWT

- JWT-urile sunt compuse din 3 părți separate de punct ( . ):
  - header
  - payload
  - signature

## JWT - Header

- Contine două componente:
  - tipul token-ului (JWT în cazul de față)
  - algoritmul folosit în semnarea token-ului: HMAC, SHA256, RSA

```
{  
  "alg": "HS256",  
  "typ": "JWT"  
}
```



Școala  
informală  
de IT

Google

Django REST Framework

# Autentificare - JWT

## JWT - Payload

- Contine date despre identitatea unei entități (ex: user) și eventual informații adiționale.

```
{  
  "name": "John Doe",  
  "admin": True  
}
```

## JWT - Signature

- Pentru a crea semnătura trebuie adunate:
  - codificarea header-ului
  - codificarea payload-ului
  - o cheie secretă
- Toate enumerate mai sus trebuie semnate folosind algoritmul specificat în header.

```
HMACSHA256(  
  base64UrlEncode(header) + '.' + base64UrlEncode(payload),  
  secret  
)
```



Scscoala  
informala  
de IT

Google

Django REST Framework

# Autentificare - JWT

- Rezultatul sunt trei şiruri de caractere tip Base64-URL separate de semnul punct ( . ) care pot fi uşor transmise în mediile bazate pe HTML și HTTP.
- Acestea sunt mai compacte decât standardele bazate pe XML (ex: SAML).

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.  
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4  
gRG9lIiwiaXNTb2NpYWwiOnRydWV9.  
4pcPyMD09o1PSyXnrXCjTwXyr4BsezdI1AVTmud2fU4
```

- În ceea ce privește autentificarea, după logarea cu succes a unui utilizator pe baza credențialelor sale, va fi returnat un JWT.
- Când utilizatorul vrea să acceseze o rută/resursă protejată, acesta trebuie să trimită și token-ul aferent.
- De obicei acesta este declarat în header-ul Authorization astfel:

**Authorization: Bearer <token>**



Școala  
informală  
de IT

Google

Django REST Framework

# DRF & JWT

Django REST Framework



Scoala  
informală  
de IT

Google

# DRF & JWT

- Pentru a folosi JWT Authentication în DRF vom folosi pachetul. Pentru instalare folosiți comanda **pip install**.

## djangorestframework\_simplejwt

- După instalarea pachetului vom avea de făcut modificări în fișierele:

- settings.py

```
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': [
        'rest_framework_simplejwt.authentication.JWTAuthentication',
    ],
}
```

- urls.py

```
path('auth/token/', jwt_views.TokenObtainPairView.as_view(), name='token_obtain_pair'),
path('auth/token/refresh/', jwt_views.TokenRefreshView.as_view(), name='token_refresh'),
```



# DRF & JWT

- Cele două rute noi au rolul să:
  - ofere un token de acces. Cu ajutorul lui utilizatorul poate accesa rutele care necesită autentificare. Valabilitate: 5 min.
  - ofere un token de refresh. Cu ajutorul lui utilizatorul poate face refresh la token-ul de acces astfel încât să obțină un nou token de acces valid. Valabilitate: 24h.
- Mai multe detalii puteți găsi în documentație:

<https://django-rest-framework-simplejwt.readthedocs.io/en/latest/>

- Pentru securizarea unui view trebuie

```
class UserViewSet(viewsets.ModelViewSet):  
    queryset = UserModel.objects.all()  
    serializer_class = UserSerializer  
    permission_classes = (IsAuthenticated,)
```

fel:

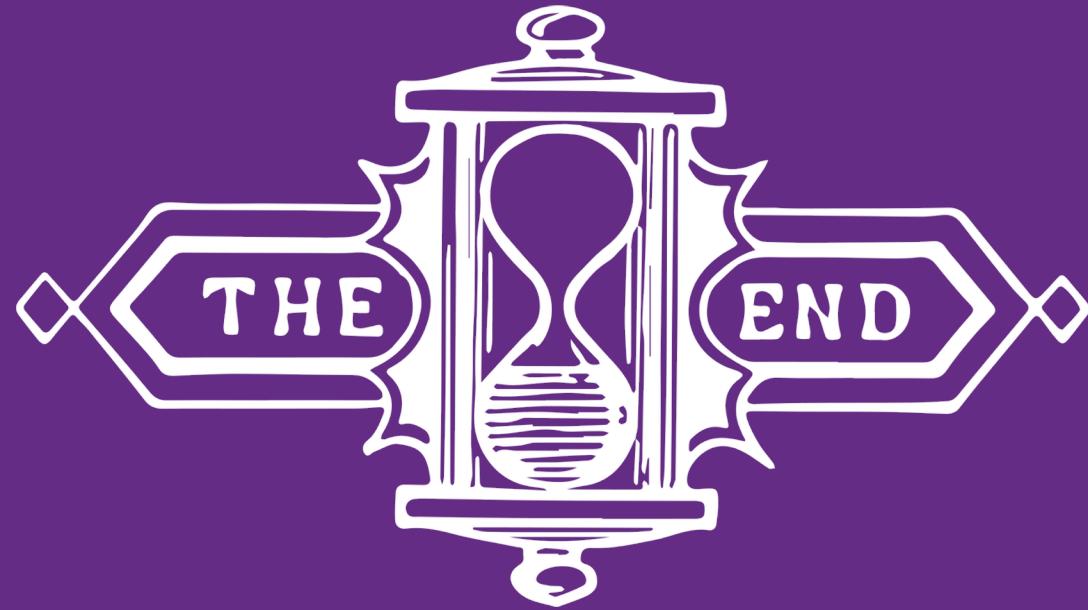


Scală  
informală  
de IT

Google

Acest este definit ca un tuplu care să conțină clasa IsAuthenticated.

Django REST Framework



Vă mulțumesc!



řcoala  
informală  
de IT

Google

Django REST Framework