Django Class Notes

Clarusway

Django Rest Framework - Serializers

Nice to have VSCode Extentions:

- Djaneiro - Django Snippets

Needs

- Python, add the path environment variable
- pip
- virtualenv

Summary

- What is API?
- Create project and app
- Serializers
    - Serializer class
    - ModelSerializer class
- Custom Validations
- Additional Features
- Relational fields
    - StringRelatedField
    - PrimaryKeyRelatedField
- Serializer Fields
    - read_only

    - write_only
- Nested Serializer

What is API?

An application programming interface is a connection between computers or between computer programs.

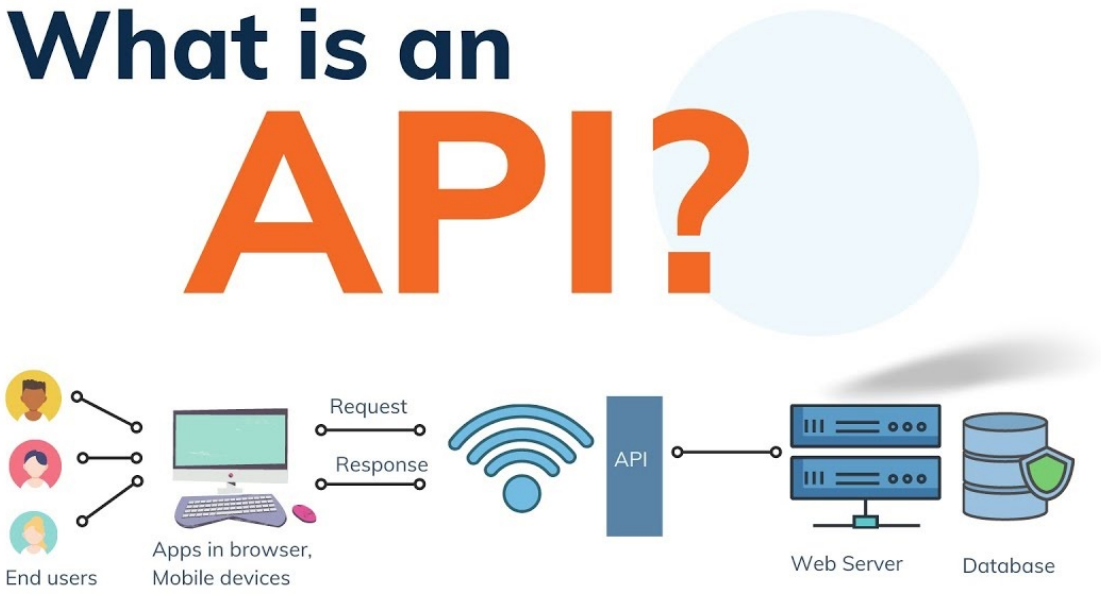
It is a type of software interface, offering a service to other pieces of software.

A document or standard that describes how to build such a connection or interface is called an API specification.

The primary goal of API is to standardize data exchange between web services. Depending on the type of API, the choice of protocol changes. On the other hand, REST API is an architectural style for building web services that interact via an HTTP protocol.

API is **the messenger that delivers your request to the provider that you're requesting it from and then delivers the response back to you**.

What is REST API?



An API, or *application programming interface*, is a set of rules that define how applications or devices can connect to and communicate with each other. A REST API is an API that conforms to the design principles of the REST, or *representational state transfer* architectural style. For this reason, REST APIs are sometimes referred to RESTful APIs*.*

First defined in 2000 by computer scientist Dr. Roy Fielding in his doctoral dissertation, REST provides a relatively high level of flexibility and freedom for developers. This flexibility is just one reason why REST APIs have emerged as a common method for connecting components and applications in a microservices architecture.

At the most basic level, an API is a mechanism that enables an application or service to access a resource within another application or service. The application or service doing the accessing is called the client, and the application or service containing the resource is called the server.

Some APIs, such as SOAP or XML-RPC, impose a strict framework on developers. But REST APIs can be developed using virtually any programming language and support a variety of data formats. The only requirement is that they align to the following six REST design principles - also known as architectural constraints:

1. **Uniform interface**. All API requests for the same resource should look the same, no matter where the request comes from. The REST API should ensure that the same piece of data, such as the name or email address of a user, belongs to only one uniform resource identifier (URI). Resources shouldn't be too large but should contain every piece of information that the client might need.
2. **Client-server decoupling**. In REST API design, client and server applications must be completely independent of each other. The only information the client application should know is the URI of the requested resource; it can't interact with the server application in any other ways. Similarly, a server application shouldn't modify the client application other than passing it to the requested data via HTTP.
3. **Statelessness**. REST APIs are stateless, meaning that each request needs to include all the information necessary for processing it. In other words, REST APIs do not require any server-side sessions. Server applications aren't allowed to store any data related to a client request.
4. **Cacheability**. When possible, resources should be cacheable on the client or server side. Server responses also need to contain information about whether caching is allowed for the delivered resource. The goal is to improve performance on the client side, while increasing scalability on the server side.
5. **Layered system architecture**. In REST APIs, the calls and responses go through different layers. As a rule of thumb, don't assume that the client and server applications connect directly to each other. There may be a number of different intermediaries in the communication loop. REST APIs need to be designed so that neither the client nor the server can tell whether it communicates with the end application or an intermediary.
6. **Code on demand (optional)**. REST APIs usually send static resources, but in certain cases, responses can also contain executable code (such as Java applets). In these cases, the code should only run on-demand.

Create project and app:

- Create a working directory, name it as you wish, cd to new directory
- Create virtual environment as a best practice:

```
python3 -m venv env # for Windows or
python -m venv env # for Windows
virtualenv env # for Mac/Linux or;
virtualenv env -p python3 # for Mac/Linux
```

- Activate scripts:

```
.\env\Scripts\activate  # for Windows
source env/bin/activate  # for MAC/Linux
```

- See the (env) sign before your command prompt.
- Install django:

```
pip install django
```

- See installed packages:

```
pip freeze

# you will see:
asgiref==3.3.4
Django==3.2.4
pytz==2021.1
sqlparse==0.4.1

# If you see lots of things here, that means there is a problem with your virtual
env activation.
# Activate scripts again
```

- Create requirements.txt same level with working directory, send your installed packages to this file, requirements file must be up to date:

```
pip freeze > requirements.txt
```

- Create project:

```
django-admin startproject main .
```

- Various files has been created!
- Check your project if it's installed correctly:

```
python3 manage.py runserver  # or,
python manage.py runserver  # or,
py -m manage.py runserver
```

- Start app

```
python manage.py startapp student_api
```

gitignore

add a gitignore file at same level as env folder, and check that it includes .env and /env lines

Python Decouple

pip install python-decouple

create a new file and name as .env at same level as env folder

copy your SECRET_KEY from settings.py into this .env file. Don't forget to remove quotation marks from SECRET_KEY

```
SECRET_KEY = django-insecure-)=b-%-w+0_^slb(exmy*mfiaj&wz6_fb4m&s=az-zs!#1^ui7j
```

go to settings.py, make amendments below

```
from decouple import config

SECRET_KEY = config('SECRET_KEY')
```

go to terminal

```
py manage.py migrate
py manage.py runserver
```

click the link with CTRL key pressed in the terminal and see django rocket.

go to terminal

```
py manage.py startapp student_api
```

go to settings.py and add 'student_api' app to installed apps

go to student_api.models.py

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

```
class Student(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
    number = models.IntegerField(blank=True, null=True)
    # blank=True for admin dashboard
    # null=True for db

    def __str__(self):
        return f"{self.last_name} {self.first_name}"
```

go to student_api.admin.py

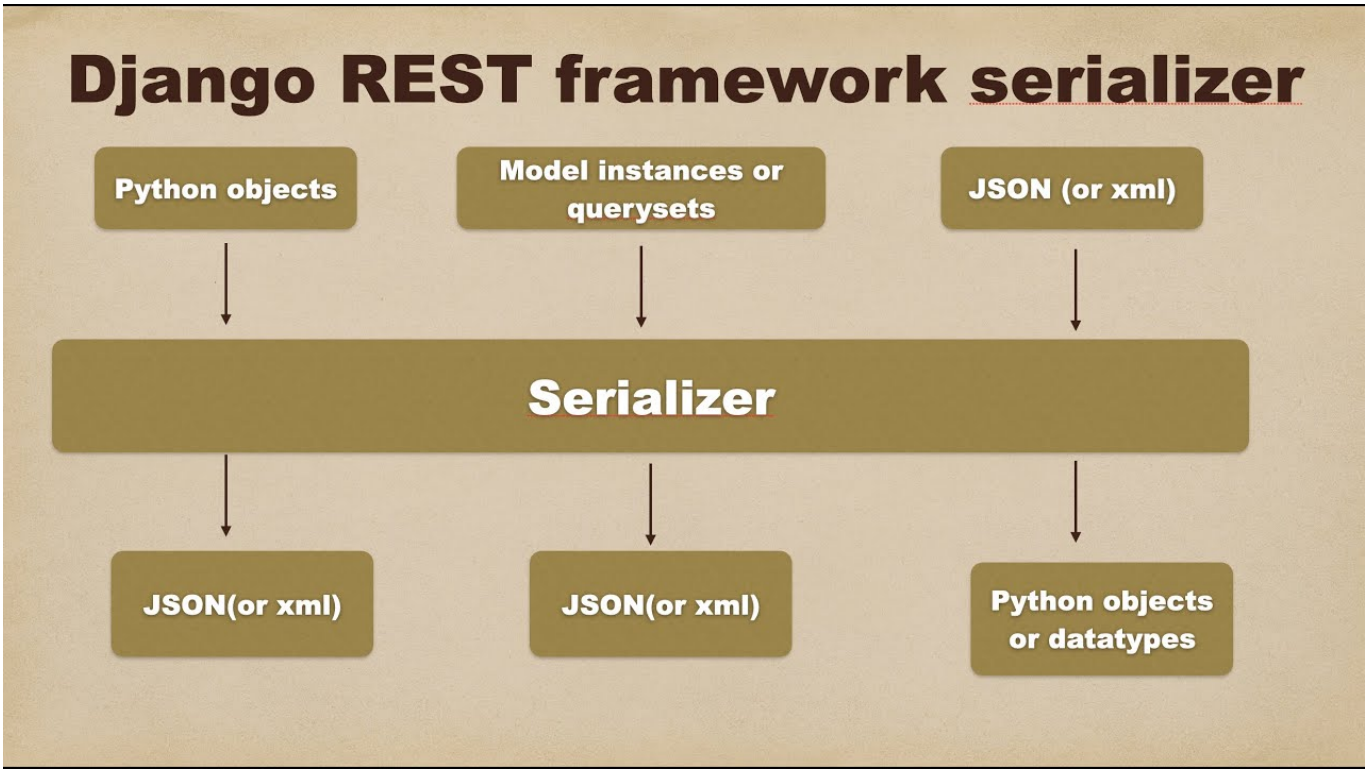
```
from django.contrib import admin
from .models import Student
# Register your models here.

admin.site.register(Student)
```

go to terminal

```
pip install djangorestframework
```

go to settings.py and add 'rest_framework' to installed apps

Serializers



Serializers allow complex data such as querysets and model instances to be converted to native Python

datatypes that can then be easily rendered into JSON, XML or other content types. Serializers also provide deserialization, allowing parsed data to be converted back into complex types, after first validating the incoming data.

Or, serializers in Django REST Framework are responsible for converting objects into data types understandable by javascript and front-end frameworks. Serializers also provide deserialization, allowing parsed data to be converted back into complex types, after first validating the incoming data.

Declaring Serializers with serializers.Serializer

create serializers.py under student_api

If we want to be able to return complete object instances based on the validated data we need to implement one or both of the .create() and .update() methods.

```
from rest_framework import serializers
from .models import Student

class StudentSerializerWithSerializer(serializers.Serializer):
    first_name = serializers.CharField(max_length=30)
    last_name = serializers.CharField(max_length=30)
    number = serializers.IntegerField(required=False)

    # If your object instances correspond to Django models you'll also want to
ensure that these methods save the object to the database.
    def create(self, validated_data):
        return Student.objects.create(**validated_data)

    def update(self, instance, validated_data):
        instance.first_name = validated_data.get('first_name',
instance.first_name)
        instance.last_name = validated_data.get('last_name', instance.last_name)
        instance.number = validated_data.get('number', instance.number)
        instance.save()
        return instance
```

go to student_api.views.py

```
from django.shortcuts import render, HttpResponse, get_object_or_404

from .models import Student

from .serializers import StudentSerializer
from rest_framework.decorators import api_view
from rest_framework.response import Response
from rest_framework import status


def home(request):
    return HttpResponse('<h1>API Page</h1>')
```

```
@api_view(['GET', 'POST'])
def student_api(request):
    if request.method == 'GET':
        students = Student.objects.all()
        serializer = StudentSerializer(students, many=True)
        return Response(serializer.data)
    elif request.method == 'POST':
        serializer = StudentSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
            data = {
                "message": f"Student {serializer.validated_data.get('first_name')}
saved successfully!"}
            return Response(data, status=status.HTTP_201_CREATED)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)

@api_view(['GET', 'PUT', 'DELETE', 'PATCH'])
def student_api_get_update_delete(request, pk):
    student = get_object_or_404(Student, pk=pk)
    if request.method == 'GET':
        serializer = StudentSerializer(student)
        return Response(serializer.data, status=status.HTTP_200_OK)
    elif request.method == 'PUT':
        serializer = StudentSerializer(student, data=request.data)
        if serializer.is_valid():
            serializer.save()
            data = {
                "message": f"Student {student.last_name} updated successfully"
            }
            return Response(data)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
    elif request.method == 'PATCH':
        serializer = StudentSerializer(student, data=request.data)
        if serializer.is_valid():
            serializer.save()
            data = {
                "message": f"Student {student.last_name} updated successfully"
            }
            return Response(data)
    elif request.method == 'DELETE':
        student.delete()
        data = {
            "message": f"Student {student.last_name} deleted successfully"
        }
        return Response(data)
```

go to main.urls.py

```
from django.contrib import admin
from django.urls import path, include
```

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('student_api.urls')),
]
```

go to student_api.urls.py

```
from django.urls import path
from .views import home,student_api, student_api_get_update_delete

urlpatterns = [
    path('', home),
    path('student/', student_api),
    path('student/<int:pk>/', student_api_get_update_delete, name = "detail")
]
```

go to terminal

```
pip freeze > requirements.txt
py manage.py createsuperuser
py manage.py runserver
```

ModelSerializer

go to serializers.py and make below amendments, comment out previous serializer.

```
from rest_framework import serializers
from .models import Student

class StudentSerializer(serializers.ModelSerializer):
    class Meta:
        model = Student
        fields = ["id", "first_name", "last_name", "number"]
        # fields = '__all__'
        # exclude = ['number']
```

Custom Validations

Those validations are related to serializers only, not admin dashboard changes.

- Field level validation:

```
def validate_title(self, value):
        """
        Check that the blog post is about Django.
```

```
        """
        if 'django' not in value.lower():
            raise serializers.ValidationError("Blog post is not about Django")
        return value
```

- Object level validation:

```
# naming must be validate_ and the field name to validate
def validate_number(self, value):
    """
    Check that student numbers below 1000.
    """
    if value > 1000:
        raise serializers.ValidationError("Student numberneed to be below 1000")
    return value
```

Additional Features

- Customer may want some info which is not in our model. Such as, time since student registration.

Normally we need to add a new field to db:

```
register_date = models.DateTimeField(auto_now_add=True)
```

makemigrations migrate

- We can create a logic inside models and get the time since student registered. But, also we can get that info using SerializerMethodField:

```
from django.utils.timezone import now

days_since_joined = serializers.SerializerMethodField()

def get_days_since_joined(self, obj):
    return (now() - obj.register_date).days
```

Relational fields

There are nested relationships in serializers.

go to student_api.models.py and make below amendments

```
from django.db import models

class Path(models.Model):
```

```
    path_name = models.CharField(max_length=50)

    def __str__(self):
        return f"{self.path_name}"

class Student(models.Model):
    path = models.ForeignKey(Path, related_name='students',
on_delete=models.CASCADE)
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
    number = models.IntegerField(blank=True, null=True)

    def __str__(self):
        return f"{self.last_name} {self.first_name}"
```

stop server and delete db tables **Delete db.sqlite3**
Delete 0001_initial.py under student_api/migrations

go to terminal

```
py manage.py makemigrations
py manage.py migrate
py manage.py createsuperuser
py manage.py runserver
```

Add path_api view to views.py

```
@api_view(['GET', 'POST'])
def path_api(request):
    # from rest_framework.decorators import api_view
    # from rest_framework.response import Response
    # from rest_framework import status

    if request.method == 'GET':
        paths = Path.objects.all()
        serializer = PathSerializer(paths, many=True, context={'request':
request})
        return Response(serializer.data)
    elif request.method == 'POST':
        # from pprint import pprint
        # pprint(request)
        serializer = PathSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
            data = {
                "message": f"Path saved successfully!"}
            return Response(data, status=status.HTTP_201_CREATED)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

go to student_api.serializers.py and make below amendments

```
from rest_framework import serializers
from .models import Student, Path

class PathSerializer(serializers.ModelSerializer):
    class Meta:
        model = Path
        fields = ["id", "path_name"]

class StudentSerializer(serializers.ModelSerializer):
    class Meta:
        model = Student
        fields = "__all__"
```

Don't forget to update urls.py

path('path/', path_api),

register on admin.py

StringRelatedField

Now we see the path as an integer. If we wish to see it as string use StringRelatedField.

StringRelatedField may be used to represent the target of the relationship using its **str** method.

go to student_api.serializers.py and make below amendments

```
from rest_framework import serializers
from .models import Student, Path

class PathSerializer(serializers.ModelSerializer):
    students = serializers.StringRelatedField(many=True)
    class Meta:
        model = Path
        fields = "__all__"

class StudentSerializer(serializers.ModelSerializer):
    path = serializers.StringRelatedField()
    class Meta:
        model = Student
        fields = "__all__"
```

StringRelatedField is read-only by default. We can not create a new object using string. Also, we can not use id only for path. Because it is path_id on the students table.

So, we can create a path_id field to assign a student with a path. path_id = serializers.IntegerField()

Serializer Fields

Serializer fields

read_only

Read-only fields are included in the API output, but should not be included in the input during create or update operations. Any 'read_only' fields that are incorrectly included in the serializer input will be ignored.

Set this to True to ensure that the field is used when serializing a representation, but is not used when creating or updating an instance during deserialization.

Defaults to

write_only

Set this to True to ensure that the field may be used when updating or creating an instance, but is not included when serializing the representation.

Defaults to False

- Make path_id write_only

PrimaryKeyRelatedField

PrimaryKeyRelatedField may be used to represent the target of the relationship using its primary key.

go to student_api.serializers.py and make below amendments

```
from rest_framework import serializers
from .models import Student, Path

class PathSerializer(serializers.ModelSerializer):
    students = serializers.PrimaryKeyRelatedField(read_only=True, many=True)
    class Meta:
        model = Path
        fields = "__all__"

class StudentSerializer(serializers.ModelSerializer):
    path = serializers.StringRelatedField()
    class Meta:
        model = Student
        fields = "__all__"
```

Nested Serializer

```
class PathSerializer(serializers.ModelSerializer):
    # students = serializers.StringRelatedField(many=True)
    students = StudentSerializer(many=True)
    class Meta:
```

```
        model = Path
        fields = '__all__'
```

Switch between different uses of nested serializers:

```
class PathSerializer(serializers.ModelSerializer):
    # students = serializers.StringRelatedField(many=True)
    # students = StudentSerializer(many=True)
    students = serializers.PrimaryKeyRelatedField(read_only=True, many=True)
    class Meta:
        model = Path
        fields = '__all__'
```

☺ **Happy Coding!** ✍

Clarusway