Django API

API = Application Programming Interface
An API is a set of rules, methods, and data formats that different software applications use to interact with each other.

For Example: Open Weather API

- Sign up for an authentication key
- Make a request
- API returns JSON data in a response
- We can use the data without access to how Open Weather has created it. All that complexity is abstracted into a single endpoint

Setup

Create fresh Django project + Rest Framework

Follow steps in the Django Install pdf

Create API

Models

In models.py:

```python
class Student(models.Model):
  name = models.TextField()
  age = models.PositiveSmallIntegerField()
  courses = models.ManyToManyField('Course')

  def __str__(self):
    course_list = ''
    for c in self.courses.all():
      course_list += c.name + ', '
    return f'Student: {self.name}, Age: {self.age}, Courses: {course_list}'


class Instructor(models.Model):
  name = models.TextField()

  def __str__(self):
    return f'Instructor: {self.name}'


class Course(models.Model):
  name = models.TextField()
  instructor = models.ForeignKey(Instructor, on_delete=models.SET_NULL,
null=True)

  def __str__(self):
    return f'Course: {self.name}'

class Grade(models.Model):
  score = models.PositiveSmallIntegerField()
  course = models.ForeignKey(Course, on_delete=models.CASCADE)
  student = models.ForeignKey(Student, on_delete=models.CASCADE)
```

Serializers

Serializers in Django Rest Framework convert data from our database models into Python data types that can then be converted into JSON. They can also deserialize - converting the other direction.

In your app create a new file: serializers.py
In serializers.py:
Imports

```python
from rest_framework import serializers
from .models import *
```

Create a serializer class for each model

```python
class StudentSerializer(serializers.ModelSerializer):
  class Meta:
    model = Student
    fields = ['id', 'name', 'age', 'courses']
```

Views / ViewSets

A ViewSet is a single class that manages the API endpoints for CRUD. This is where business logic can happen.

In views.py:
imports

```python
from rest_framework import viewsets

from .models import *
from .serializers import *
```

Create a viewset for each model

```python
class StudentViewSet(viewsets.ModelViewSet):
```

```
    queryset = Student.objects.all()
    serializer_class = StudentSerializer
```

URL Routes

Defines the URL endpoints that the Django server will listen to.
Endpoints are specific http URLs that represents functions provided by the API

In urls.py:
Imports

```python
from django.urls import path, include
from rest_framework import routers

from your_app.views import *
```

Register the route and add to urlpatterns

```python
router = routers.DefaultRouter()
router.register(r'students', StudentViewSet)

urlpatterns = [
    path('', include(router.urls)),
]
```

This automatically creates the following routes:

- GET /students/ - list all students
- GET /students/{id}/ - retrieve a specific student
- POST /students/   - create a new student
- PUT /students/{id}/ - update a specific student
- DELETE /students/{id}/ - delete a specific student

Practice

Create serializers, viewSets, and routes for the Course, Instructor, and Grade models.

Start Django Server

1. Start server: $ `python manage.py runserver`
2. Go to development server url (link shown in terminal)
3. CRUD

Example of Serialization

From Object => JSON

In serializers.py

```python
course = Course.objects.get(id=1)

print('\n**********************\n')
print(f'before serializer, course is a python object: {course}')

serializer = CourseSerializer(course)

print(f'after serializer, course is JSON: {serializer.data}')
print('\n**********************\n')
```

Overriding CRUD: Read

Let's add a field, 'letter_grade' that shows the letter instead of their number score.

With a Serializer:

In models.py

```python
class Grade(models.Model):
  score = models.PositiveSmallIntegerField()
  course = models.ForeignKey(Course, on_delete=models.CASCADE)
  student = models.ForeignKey(Student, on_delete=models.CASCADE)
```

In serializers.py

Python

```python
class GradeSerializer(serializers.ModelSerializer):
  letter_grade = serializers.SerializerMethodField()

  class Meta:
    model = Grade
    fields = ['id', 'score', 'course', 'student', 'letter_grade']

  def get_letter_grade(self, obj):
    if obj.score >= 90:
        return 'A'
    elif obj.score >= 80:
        return 'B'
    elif obj.score >= 70:
        return 'C'
    elif obj.score >= 60:
        return 'D'
    else:
        return 'F'
```

With a ViewSet:

In views.py

Python

```python
def get_letter_grade(obj):
  if obj.score >= 90:
      return 'A'
  elif obj.score >= 80:
      return 'B'
  elif obj.score >= 70:
      return 'C'
  elif obj.score >= 60:
      return 'D'
  else:
      return 'F'

class GradeViewSet(viewsets.ModelViewSet):
  queryset = Grade.objects.all()
  serializer_class = GradeSerializer

  def retrieve(self, request, pk=None):
    grade = Grade.objects.get(pk=pk)
```

```
    serializer = GradeSerializer(grade)
    data = serializer.data
    data['letter_grade'] = get_letter_grade(grade)
    return Response(data)
```

Overriding CRUD: Create

Let's add 'Professor' in front of Instructor names when creating them.
In Views.py:

```python
class InstructorViewSet(viewsets.ModelViewSet):
  queryset = Instructor.objects.all()
  serializer_class = InstructorSerializer

  def create(self, request):
     mutable_data_copy = request.data.copy()
     mutable_data_copy['name'] = f'Professor {mutable_data_copy['name']}'

     serializer = InstructorSerializer(data=mutable_data_copy)
     serializer.is_valid(raise_exception=True)
     serializer.save()
     return Response(serializer.data)
```

Overriding CRUD: Update

If a student gets a grade above a 90, change their name to 'Brilliant {student.name}'
In views.py:
Update GradeViewSet:

```python
  def update(self, request, pk=None):
    grade = Grade.objects.get(pk=pk)
    grade_serializer = GradeSerializer(data=request.data)
    grade_serializer.is_valid(raise_exception=True)
    grade_serializer.save()
    student = Student.objects.get(id = grade.student.id)
    if int(request.data['score']) > 90:
```

```
      if not student.name.startswith('Brilliant '):
        student.name = f'Brilliant {student.name}'
        student.save()
    else:
      if student.name.startswith('Brilliant '):
         student.name = student.name.replace('Brilliant ', '')
         student.save()
    return Response(grade_serializer.data)
```

Overriding CRUD: Delete

Block deleting a Course if it has any Grades associated with it.
In views.py:
Import:

```python
from rest_framework.exceptions import ValidationError
```

Update CourseViewSet:

```python
  def destroy(self, request, pk=None):
     course = self.get_object()
     if Grade.objects.filter(course=course).exists():
        raise ValidationError({'detail': 'Cannot delete course because it has
associated grades.'})
     self.perform_destroy(course)
     return Response()
```