Классы APIView

Фреймворк REST предоставляет APIView класс, который является подклассом класса Django View.

APIView Классы отличаются от обычных View классов следующими способами:

1. Запросы, передаваемые методам обработчика, будут экземплярами фреймворка REST , а не экземплярами Request Django, т.е. HttpRequest
2. Методы обработчика могут возвращать REST framework Response, а не Django HttpResponse. Представление будет управлять согласованием содержимого и установкой правильного средства визуализации в ответе.
3. Любые APIException исключения будут перехвачены и преобразованы в соответствующие ответы.
4. Входящие запросы будут аутентифицированы, и перед отправкой запроса методу обработчика будут выполняться соответствующие проверки разрешений и/или ограничений.

Использование APIView класса почти такое же, как использование обычного View класса, как обычно, входящий запрос направляется соответствующему методу обработчика, такому как .get()или .post(). Кроме того, в классе может быть установлен ряд атрибутов, управляющих различными аспектами политики API.

Рассмотрим пример:
Создадим API для хранения информации об известных людях. Небольшая электронная энциклопедия. Определим класс MenAPIView на основе базового APIView. Это класс, на основе которого создаются все другие классы представлений в DRF и он содержит лишь некоторый базовый функционал. Полный список классов представлений можно посмотреть на странице официальной документации https://www.django-rest-framework.org/api-guide/generic-views/

- CreateAPIView
- ListAPIView
- RetrieveAPIView
- DestroyAPIView
- UpdateAPIView
- ListCreateAPIView
- RetrieveUpdateAPIView
- RetrieveDestroyAPIView
- RetrieveUpdateDestroyAPIView

Для начала импортируем класс APIView вместе с классом Response для формирования ответа в виде JSON

```
from rest_framework.response import Response
from rest_framework.views import APIView
```

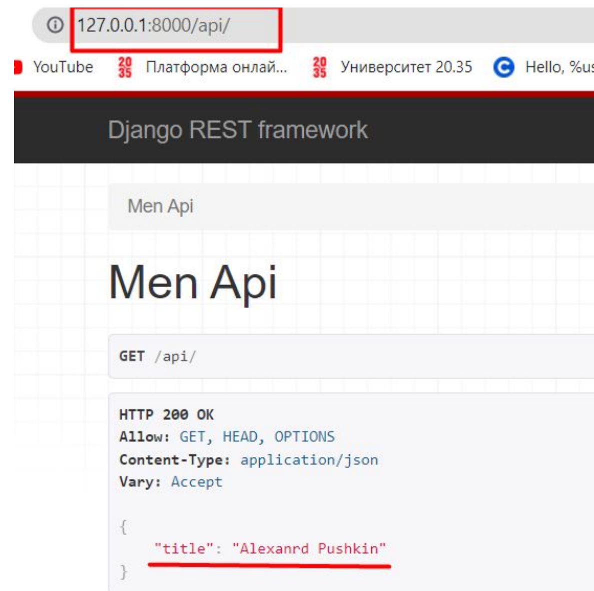
Определим MenAPIView. Будем создавать представления без использования сериализатора. Для этого в ответ передадим данные в формате JSON.

```
class MenAPIView(APIView):
    def get(self, request):
        return Response({'title': 'Alexanrd Pushkin'})
```

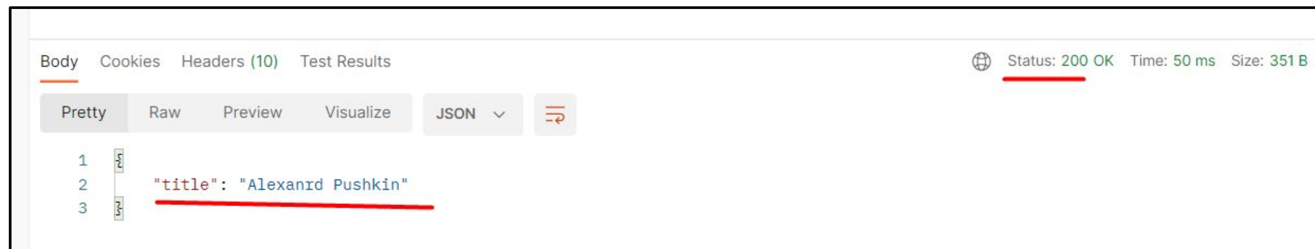
Прописав соответствующий маршрут мы получим следующую информацию по URL

```
from django.urls import path
from . import views

urlpatterns = [
    path('', views.MenAPIView.as_view()),
]
```



Запросив Get запрос данного URL в Постмен также получим



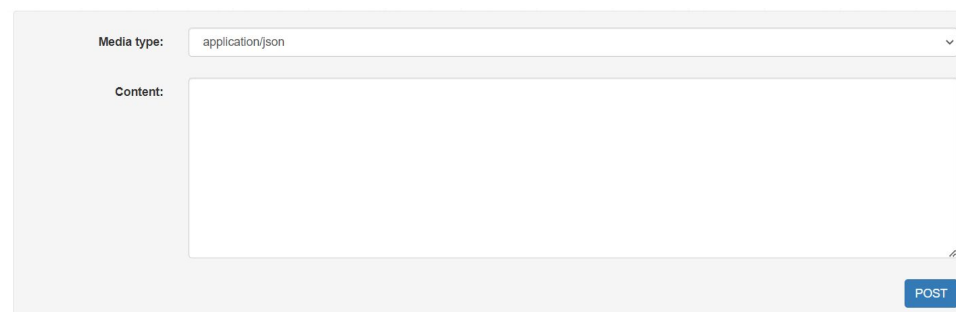
При попытке организации Post запроса нам сообщат об ошибке. Этот ответ был автоматически сгенерирован базовым классом APIView, который берет на себя обработку типовых ошибок при запросах

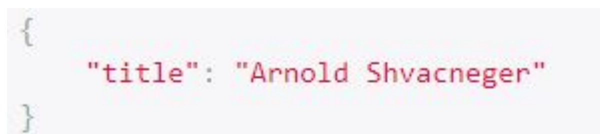
Добавим в класс метод post

```python
class MenAPIView(APIView):
    def get(self, request):
        return Response({'title': 'Alexanrd Pushkin'})

    def post(self, request):
        return Response({'title': 'Arnold Shvacneger'})
```

У нас появится форма для внесения данных и при нажатии на кнопку POST мы получим сообщение о новых данных





Это тестовые примеры и реальный API-запрос, как правило, ожидает получения данных из таблиц БД или, какого-либо другого хранилища. В рамках класса MenAPIView это можно сделать, следующим образом (для GET-запроса):

```python
def get(self, request):
    lst = Men.objects.all().values()
    return Response({'posts': list(lst)})
```

Для этого необходимо предварительно создать модель Men и Category

```python
class Men(models.Model):
    title = models.CharField(max_length=255)
    content = models.TextField(blank=True)
    time_create = models.DateTimeField(auto_now_add=True)
    is_published = models.BooleanField(default=True)
    cat = models.ForeignKey('Category', on_delete=models.PROTECT, null=True)
    class Meta:
        verbose_name = "Man"
        verbose_name_plural = "Men"

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

```python
class Category(models.Model):
    name = models.CharField(max_length=100, db_index=True)

    class Meta:
        verbose_name = "Category"
        verbose_name_plural = "Categories"

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

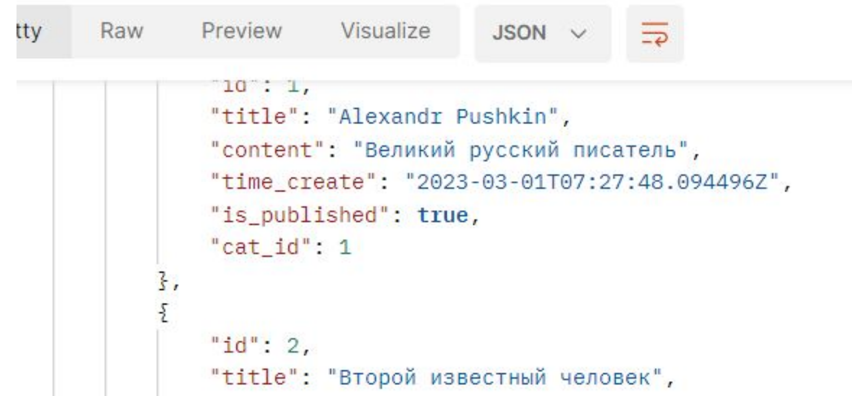
Подключить их в админке и заполнить данными

```python
from .models import Men, Category

admin.site.register(Men)
admin.site.register(Category)
```

Выбираем все записи из таблицы Men, преобразуем их к списку и возвращаем в виде JSON-строки. Если теперь выполнить GET-запрос через Postman или на странице API, то получим всю информацию из БД.


```
{
    "posts": [
        {
            "id": 1,
            "title": "Alexandr Pushkin",
            "content": "Великий русский писатель",
            "time_create": "2023-03-01T07:27:48.094496Z",
            "is_published": true,
            "cat_id": 1
        },
        {
            "id": 2,
            "title": "Второй известный человек",
            "content": "Ну очень известный",
            "time_create": "2023-03-01T07:36:35.559104Z",
            "is_published": true,
            "cat_id": 2
        }
    ]
}
```

Изменим POST-запрос, чтобы он добавлял информацию в нашу таблицу

```python
def post(self, request):
    post_new = Men.objects.create(
        title=request.data['title'],
        content=request.data['content'],
        cat_id=request.data['cat_id']
    )
    return Response({'post': model_to_dict(post_new)})
```

Здесь возвращается JSON-строка с содержимым добавленной записи. Для этого мы должны объект post_new преобразовать в словарь, например, с помощью функции model_to_dict фреймворка Django

```python
from django.forms import model_to_dict
```

Данный способ организации Get и Post запросов позволяет обходиться без сериализатора, используя только класс представления. Но когда API становится более функциональным подключение сериализаторов необходимо.

При реализации API сайта обмен данными выполняется посредством определенного формата. Чаще всего используют JSON кодирование, реже XML. При необходимости можно описать свой формат обмена данными. В 99% случаях все же применяется JSON. Роль сериализатора выполнять конвертирование произвольных объектов языка Python в формат JSON (в том числе модели фреймворка Django и наборы QuerySet). И, обратно, из JSON – в соответствующие объекты Python. (Полагая, что используется JSON в API-запросах).

Определим сериализатор с тем же набором атрибутов, что и класс модели

```
class MenSerializer(serializers.Serializer):
    title = serializers.CharField(max_length=255)
    content = serializers.CharField()
    time_create = serializers.DateTimeField(read_only=True)
    time_update = serializers.DateTimeField(read_only=True)
    is_published = serializers.BooleanField(default=True)
    cat_id = serializers.IntegerField()
```

Теперь здесь присутствует атрибут cat_id как поле с целочисленным значением, так как на вход сериализатора будет передаваться объект, где внешний ключ cat уже определен и сформирован как локальный атрибут cat_id, содержащий целое значение.

Применим этот сериализатор в файле views.py

```python
from .serializers import MenSerializer
class MenAPIView(APIView):
    def get(self, request):
        w = Men.objects.all()
        return Response({'posts': MenSerializer(w, many=True).data})

    def post(self, request):
        post_new = Men.objects.create(
            title=request.data['title'],
            content=request.data['content'],
            cat_id=request.data['cat_id']
        )

        return Response({'post': MenSerializer(post_new).data})
```

В методе get() теперь просто читаются все записи, представленные объектом QuerySet, и передаем эту коллекцию в сериализатор, дополнительно указываем параметр many=True, так как на выходе нужно сформировать список из записей таблицы (по умолчанию формируется одна запись). Далее, класс Response «знает» как обрабатывать объект сериализованных данных, превращая их в байтовую JSON-строку.

Выполним теперь POST-запрос для того же адреса. Теперь возвратились данные, содержащие одну новую добавленную запись. Причем, здесь также видим значения времени, которое было автоматически подставлено нашим сериализатором.

Однако, если в POST-запросе передать неверные данные, то возникнет ошибка на этапе добавления записи в БД. Мы можем обработать этот момент с помощью нашего сериализатора, следующим образом:

```
def post(self, request):
    serializer = MenSerializer(data=request.data)
    serializer.is_valid(raise_exception=True)
    post_new = Men.objects.create(
        title=request.data['title'],
        content=request.data['content'],
        cat_id=request.data['cat_id']
    )

    return Response({'post': MenSerializer(post_new).data})
```

Здесь мы формируем объект сериализации на основе принятых от клиента данных, а затем, вызываем метод is_valid() и указываем параметр raise_exception=True для генерации исключений (ошибок), которые будут передаваться в виде JSON-строки клиенту. При валидации прописываем аргумент raise_exception=True.

Методы save(), create() и update() класса Serializer

Сериализатор MenSerializer отвечает сейчас только за конвертацию данных в JSON-формат и обратно. Однако, хорошей практикой считается, когда в сериализаторе определяется алгоритм сохранения или изменения данных в БД. Сейчас добавление новой записи происходит в методе post() представления WomenAPIView. Но лучше этот функционал перенести в сериализатор. Для этого каждый класс сериализаторов имеет два специальных метода:

create(self, validated_data) – для добавления (создания) записи (данных);
update(self, instance, validated_data) – для изменения данных (записи).
Для начала определим метод create() для переноса кода из метода post() класса представления в сери… метод create(), который должен возвращать экземпляр объе…

```
def create(self, validated_data):
    return Men.objects.create(**validated_data)
```

Это объект модели класса Men. Причем коллекция validated_data – это словарь с проверенными данными, полученными в результате POST-запроса после выполнения метода serializer.is_valid().

Теперь нужно вызвать метод create(). Для этого в представлении MenAPIView в методе post() удаляем строчки, связанные с созданием новой записи и вместо них вызываем метод save():

```
def post(self, request):
    serializer = MenSerializer(data=request.data)
    serializer.is_valid(raise_exception=True)
    serializer.save()

    return Response({'post': serializer.data})
```

Расширим функционал API и добавим возможность изменять уже существующую запись в БД. Для этого в сериализаторе нужно прописать метод update()

```
def update(self, instance, validated_data):
    instance.title = validated_data.get("title", instance.title)
    instance.content = validated_data.get("content", instance.content)
    instance.time_update = validated_data.get("time_update", instance.time_update)
    instance.is_published = validated_data.get("is_published", instance.is_published)
    instance.cat_id = validated_data.get("cat_id", instance.cat_id)
    instance.save()
    return instance
```

В этот метод передается ссылка instance на объект, который следует изменить и словарь validated_data с проверенными данными. Далее, используя ORM Django меняем локальные атрибуты объекта instance и вызываем метод save() для изменения данных в таблице БД. В конце возвращаем объект instance.

Пропишем метод put() для PUT-запроса в классе представления

```
def put(self, request, *args, **kwargs):
    pk = kwargs.get("pk", None)
    if not pk:
        return Response({"error": "Method PUT not allowed"})

    try:
        instance = Men.objects.get(pk=pk)
    except:
        return Response({"error": "Object does not exists"})

    serializer = MenSerializer(data=request.data, instance=instance)
    serializer.is_valid(raise_exception=True)
    serializer.save()

    return Response({"post": serializer.data})
```

Метод put() ожидает входной параметр pk – идентификатор записи, по которому она выбирается из БД, а затем, изменяется сериализатором MenSerializer(). Так как в сериализаторе MenSerializer дополнительно указан параметр instance, то при вызове метода serializer.save() будет автоматически вызываться метод update() сериализатора.

Пропишем еще один маршрут для представления, где передается параметр pk

```
urlpatterns = [
    path('', views.MenAPIView.as_view()),
    path('<int:pk>/', views.MenAPIView.as_view()),
]
```

И добавим метод Delete, где передается параметр pk. Для его реализации нам не нужно преобразовывать данные, поэтому сериализатор не используется.

```
def delete(self, request, *args, **kwargs):
    pk = kwargs.get("pk", None)
    if not pk:
        return Response({"error": "Method DELETE not allowed"})

    w = Men.objects.get(pk=pk)
    w.delete()

    return Response({"post": "delete post " + str(pk)})
```

Задание

Создать API Для хранения данных о товарах. В БД 3 таблицы: Товар и Категория, Производитель. В Таблице товар следующие поля: Наименование, Размер, Производитель, Категория и Цена. В таблице Категория: Наименование. В таблице Производитель: Название фирмы, Страна производитель. Организовать CRUD с помощью класса APIView (только).