Создание Django API используя Django Rest Framework часть 2

Это вторая часть из серии статей про Django API. Первая часть находится [здесь](https://webdevblog.ru/sozdanie-django-api-ispolzuya-django-rest-framework-apiview/). В этой статье мы заново создадим **Blog API**, но в этот раз используем класс **[GenericAPIView](https://www.django-rest-framework.org/api-guide/generic-views/" \l "genericapiview" \t "_blank)** вместо **[APIView](https://www.django-rest-framework.org/api-guide/views/" \t "_blank).**

Возможно у вас сразу возникнет вопрос чем GenericAPIView отличается от APIView. Класс GenericAPIView расширяет возможности APIView, добавляя в него часто используемые методы list и detail. Рассмотрим пример использования GenericAPIView.

class ArticleView(ListModelMixin, GenericAPIView):

queryset = Article.objects.all()

serializer\_class = ArticleSerializer

Из примера видно что мы унаследовали наш базовый класс от двух классов ListModelMixin и GenericAPIView

Класс GenericAPIView обеспечивает базовую функциональность. ListModelMixin реализует action .list() (класс ListModelMixin является [Mixins](https://www.django-rest-framework.org/api-guide/generic-views/" \l "mixins" \t "_blank) , то есть в нем реализуется actions, которые используются для обеспечения базового поведения представления). Для того что использовать .list() нам нужно связать метод get с action list. Для этого внесем соотвествующие изменения в наш файл **article/views.py**

from rest\_framework.generics import GenericAPIView

from rest\_framework.mixins import ListModelMixin

from .models import Article

from .serializers import ArticleSerializer

**class** ArticleView(ListModelMixin, GenericAPIView):

queryset = Article.objects.all()

serializer\_class = ArticleSerializer

**def** get(self, request, \*args, \*\*kwargs):

**return** self.list(request, \*args, \*\*kwargs)

Этот пример демонстрирует возможность получения списка всех статей, которые есть в настоящее время в нашей базе данных. Для того что бы это заработало, мы определили два обязательных атрибута.

1. **queryset**Это базовый **queryset** (запрос к базе) который используется для получение объектов. В нашем случае получение все статей. Если нам бы потребовалось например запрос с фильтрацией и т.п., мы могли бы переопределить метод get\_queryset и в через него вернуть требуемый запрос.
2. **serializer\_class**Это класс сериализатора, который используется для проверки и десериализации объектов из базы. Мы использовали ArticleSerializer которые мы создали ранее.

Однако наш сериализатор копирует много информации, которая и также содержится в модели Article. Я думаю, было бы неплохо, если бы мы могли сделать наш код немного более коротким.

#### ****ModelSerializer****

Давайте обновим наш сериализатор, как показано ниже:

class ArticleSerializer(serializers.ModelSerializer):

**class** Meta:

model = Article

fields = ('id', 'title', 'description', 'body', 'author\_id')

Используя **ModelSerializer,**мы сразу получаем методыcreate и update . Кроме того, так же набор валидаторов по умолчанию.

Поскольку мы создаем приложение **CRUD**, у нас должна быть возможность создавать статьи. Для этого мы используем другой **мixin** под названием **CreateModelMixin**.

Обновим наш **article/views.py** следующим образом:

from rest\_framework.generics import get\_object\_or\_404

from rest\_framework.generics import GenericAPIView, CreateModelMixin

from rest\_framework.mixins import ListModelMixin

from .models import Article, Author

from .serializers import ArticleSerializer

**class** ArticleView(ListModelMixin, CreateModelMixin, GenericAPIView):

queryset = Article.objects.all()

serializer\_class = ArticleSerializer

**def** get(self, request, \*args, \*\*kwargs):

**return** self.list(request, \*args, \*\*kwargs)

**def** perform\_create(self, serializer):

author = get\_object\_or\_404(Author, id=self.request.data.get('author\_id'))

**return** serializer.save(author=author)

**def** post(self, request, \*args, \*\*kwargs):

**return** self.create(request, \*args, \*\*kwargs)

Поскольку в каждой статье должен быть автор, то в каждом **post** запросе есть параметр **author\_id** и он использует его для получения соответствующего автора из базы.

Теперь можно создать статью, отправив POST запрос по адресу http://127.0.0.1:8000/api/articles/, точно так же как в прошлой раз.  
Уверен, что вы уже оценили размер кода, которыми мы сократили в сравнение с предыдущей статьей.

Что касается того, что мы определили методы **get** и **post** в нашем классе, но не определяли другие методы. Класс **GenericAPIViews** существуют, как раз для этого. **CreateAPIView** наследуется от **CreateModelMixin**, который мы использовали выше, и в нем определили метод post. Поэтому мы можем наследоваться от **CreateAPIView** и забыть о написании собственного метода **post**. То же самое относится и к методу **get**. Мы можем просто наследоваться от **ListAPIView** и забыть о написании метода get.

Обновим нам класс ArticleView следующим образом:

from rest\_framework.generics import get\_object\_or\_404

from rest\_framework.generics import CreateAPIView, ListAPIView

from .models import Article, Author

from .serializers import ArticleSerializer

**class** ArticleView(CreateAPIView, ListAPIView):

queryset = Article.objects.all()

serializer\_class = ArticleSerializer

**def** perform\_create(self, serializer):

author = get\_object\_or\_404(Author, id=self.request.data.get('author\_id'))

**return** serializer.save(author=author)

На этом этапе работа API не должна измениться, мы просто упростили наш код.

Но мы можем еще больше сократить наш код. Для этого, используем специальный класс **GenericView** и скомбинируем создание статьи, и ее публикацию. Этот класс называется **ListCreateAPIView**. Обновим наш код следующим образом.

from rest\_framework.generics import get\_object\_or\_404

from rest\_framework.generics import ListCreateAPIView

from .models import Article, Author

from .serializers import ArticleSerializer

**class** ArticleView(ListCreateAPIView):

queryset = Article.objects.all()

serializer\_class = ArticleSerializer

**def** perform\_create(self, serializer):

author = get\_object\_or\_404(Author, id=self.request.data.get('author\_id'))

**return** serializer.save(author=author)

Далее разрешим пользователям обновлять свои статьи. Для этого, нам нужно предоставить пользователю способ получение статьи. DRF предоставляет нам класс **RetrieveAPIView**. Cоздадим новый класс, который наследуется от **RetrieveAPIView**, как показано ниже.

from rest\_framework.generics import get\_object\_or\_404

from rest\_framework.generics import ListCreateAPIView, RetrieveAPIView

from .models import Article, Author

from .serializers import ArticleSerializer

**class** ArticleView(ListCreateAPIView):

queryset = Article.objects.all()

serializer\_class = ArticleSerializer

**def** perform\_create(self, serializer):

author = get\_object\_or\_404(Author, id=self.request.data.get('author\_id'))

**return** serializer.save(author=author)

**class** SingleArticleView(RetrieveAPIView):

queryset = Article.objects.all()

serializer\_class = ArticleSerializer

Заметьте, как мы просто добавили еще один класс, и это все? Это основное преимущество от использования GenericViews.  
Далее нам нужно обновить наши URL, чтобы воспользоваться новым классом. Изменим **article/urls.py**

from django.urls import path

from .views import ArticleView, SingleArticleView

app\_name = "articles"

# app\_name will help us do a reverse look-up latter.

urlpatterns = [

path('articles/', ArticleView.as\_view()),

path('articles/<int:pk>', SingleArticleView.as\_view()),

]

С помощью класса **RetrieveAPIView** мы можем просматривать только одну статью, используя идентификатор статьи. Чтобы иметь возможность обновить статью, нам нужно будет использовать другой GenericView **RetrieveUpdateAPIView**.

Обновите наш класс SingleArticleView следующим образом:

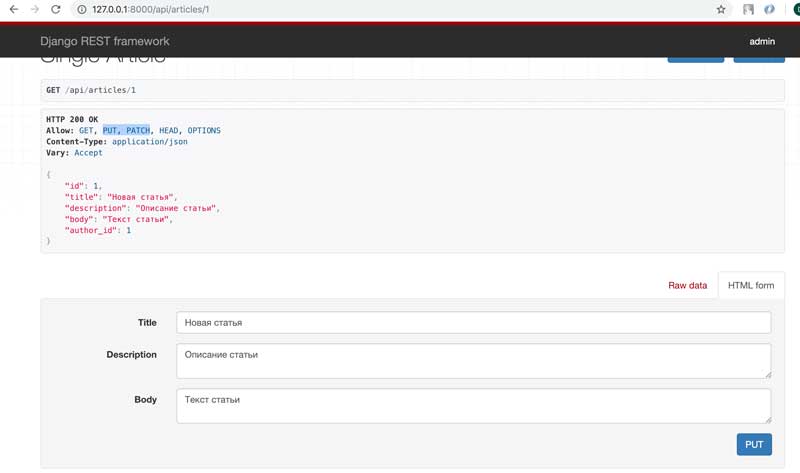
from rest\_framework.generics import RetrieveUpdateAPIView

**class** SingleArticleView(RetrieveUpdateAPIView):

queryset = Article.objects.all()

serializer\_class = ArticleSerializer

А затем зайти по адресу http://127.0.0.1:8000/api/articles/1, обратите внимание на появление методов PUT, PATCH



Теперь для таблицы Article у нас есть методы **create, list, retrieve и update**.

Мы забыли об еще одном методе, я уверен, вы правильно догадались. Метод удаления статей. Для этого мы будем следовать той же тенденции и выберем другой общий класс DRF, который был специально создан для этого. **RetrieveUpdateDestroyAPIView**. Внесем соответсвующие изменения в нас класс SingleArticleView:

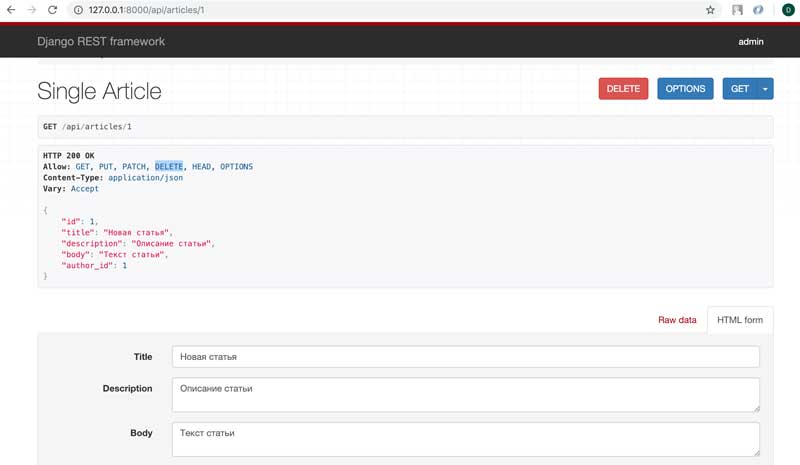
from rest\_framework.generics import RetrieveUpdateDestroyAPIView

**class** SingleArticleView(RetrieveUpdateDestroyAPIView):

queryset = Article.objects.all()

serializer\_class = ArticleSerializer

С этим классом, мы можем получить статью, обновить и удалить.



Используя **GenericAPIViews** мы значительно сократили объем кода, который нам пришлось писать для выполнения операций **CRUD**. Я не уверен, что можно было бы написать еще меньше кода используя viewsets.