Создание Django API используя Django Rest Framework часть 1

API (Application Programming Interface — интерфейс прикладного программирования) — это программное обеспечение, которое позволяет двум приложениям общаться друг с другом. В серии статей будет показано как создать API, которое позволит создавать, читать, редактировать и удалять статьи, в нашем тестовом блоге. Это будет проект имитирующий блог у которого должно быть серверное API. Мы рассмотрим различные способы создания API на базе библиотеки Django Rest Framework (DFR). Серия будет состоять из 3 частей. В первой статье мы рассмотрим использование простого класса **APIView** (часть 1), во второй использования более расширененного класса **GenericAPIView** ([часть 2](https://webdevblog.ru/sozdanie-django-api-ispolzuya-djangorestframework-chast-2/)) и, наконец в третьей использования пожалуй самого продвинутого класса **ViewSets** ([часть 3](https://webdevblog.ru/sozdanie-django-api-ispolzuya-djangorestframework-chast-3/)). Так как мы всего лишь сравниваем применение этих трех классов то обзоры будут короткие и мы не будет подробно рассматривать все возможности выше упомянутых классов.

Итак, приступим. Начнем с установки Django и Django Rest Framework. В терминале создайте каталог и дайте ему любое описательное имя; Затем перейдите в созданный каталог и установите Django, набрав:

pip install django

Так как мы будем использовать Django Rest Framework (DRF) для создания API, установите и его, так же, как мы делали это выше.

pip install djangorestframework

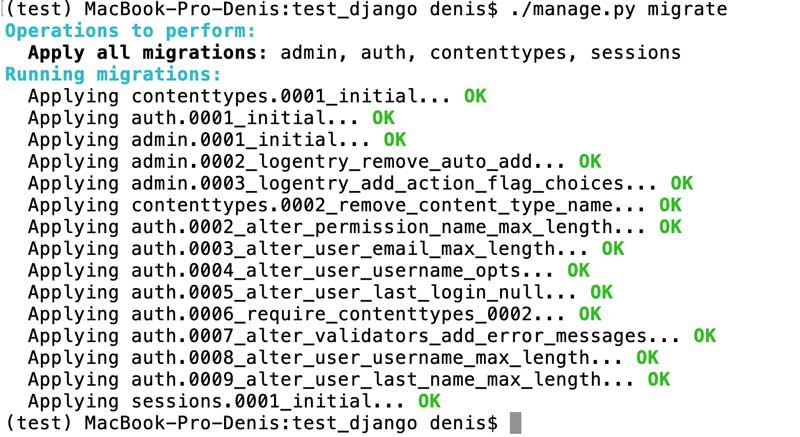
Теперь все готово что бы создать наш первый DRF API.

Для этого нам нужно создать проект Django для работы. Для этого мы запустим следующую команду:

django-admin startproject test\_django

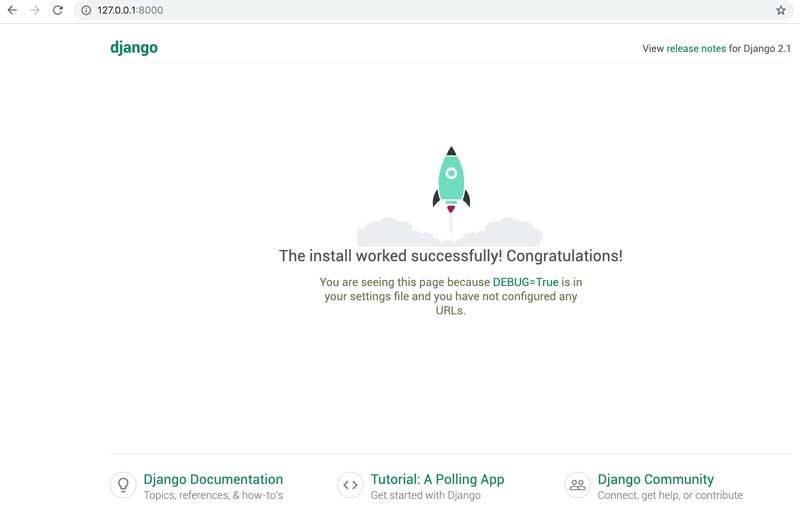
Затем запустить миграцию базы данных. Таким образом будет создана база данных по умолчанию (на sqlite), со всеми требуемыми таблицами. Запустите команду:

python manage.py migrate



Теперь можно запустить сервер, чтобы убедиться, что все в порядке:

python manage.py runserver



Сейчас у нас есть проект Django, в который мы можем интегрировать Django rest framework. Откройте файл **settings.py** и добавьте в атрибут установленных приложениях INSTALLED\_APPS = […] строку rest\_framework в кавычках.

INSTALLED\_APPS = [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

'rest\_framework'

]

Далее приступим к созданию приложения для нашего проекта. Мы создадим приложение **article**, в котором мы разместим все, что связано со статьями.

python manage.py startapp article

Потом необходимо включить созданное приложение в список установленных приложений:

INSTALLED\_APPS = [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

'rest\_framework',

'article'

]

Затем перейдите к файлу **models.py**. В нем, мы создадим наши модели.



Начнем с создания модели авторов:

**from** django.db **import** models

**class** Author(models.Model):

name = models.CharField(max\_length=255)

email = models.EmailField()

Затем ниже создадим модель статей:

**class** Article(models.Model):

title = models.CharField(max\_length=120)

description = models.TextField()

body = models.TextField()

author = models.ForeignKey('Author', related\_name='articles', on\_delete=models.CASCADE)

Далее нужно создать таблицы в базе данных на основе описанные моделей. Для этого нужно провести миграцию. С начало нужно создать файлы миграции а потом ее запустить. Для этого в командной строки запустите две команды:

python manage.py makemigrations

python manage.py migrate



Далее, создадим пользователя с правами администратора:

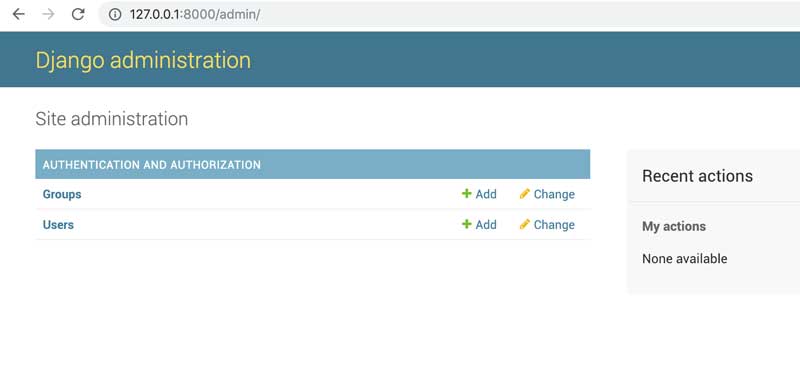
python manage.py createsuperuser

команда запросить данные нового пользователя. Нужно будет ввести имя пользователя, его email и пароль с подтверждением.



Теперь можно снова запустить сервер и зайти на страницу админки <http://127.0.0.1:8000/admin/login/>

Далее нужно ввести учетные данные администратора и мы попадем в приложение Django administration.



Далее, нужно зарегистрировать наши модели, чтобы они появлялись на этой странице.  
Для этого мы откройте файл **article/admin.py**  и внесите следующие изменения:

**from** django.contrib **import** admin

**from** *.models* **import** Article, Author

admin.site.register(Article)

admin.site.register(Author)

Теперь наша админка должна выглядеть как то так:



Пришло время создать наше API.  
Мы начнем с метода через которого можно просмотреть все статьи. Для этого откройте файл **article/views.py** и вставьте следующий код

**from** rest\_framework.response **import** Response

**from** *rest\_framework.views* **import** APIView

**from** *.models* **import** Article

**class** ArticleView(APIView):

**def** get(self, request):

articles = Article.objects.all()

**return** Response({"articles": articles})

Далее нам нужно создать URL-адрес, с которого пользователь сможет получить доступ к этому методу. Для этого создайте файл **article/urls.py** . В этом файле вставьте следующий код.

**from** django.urls **import** path

**from** *.views* **import** ArticleView

app\_name = "articles"

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

urlpatterns = [

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

]

Далее нам нужно включить эти URL-адреса в основной файл URL-адресов **test\_django/urls.py**

Чтобы включить наши URL из приложения articles в основную конфигурацию URL, мы будем использовать метод  include  Django. Откройте файл **test\_django/urls.py**, отредактируйте следующим образом:

from django.contrib import admin

from django.urls import path, include

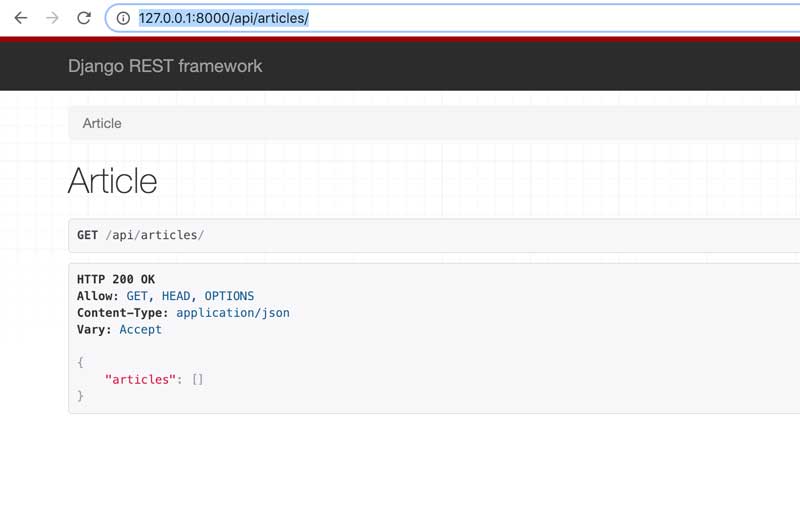
urlpatterns = [

path('admin/', admin.site.urls),

path('api/', include('article.urls')),

]

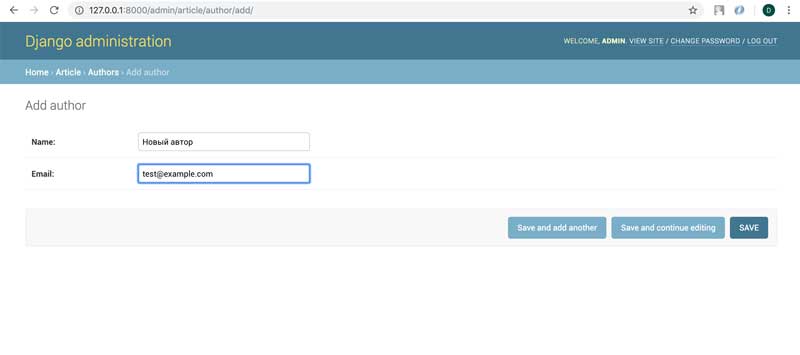
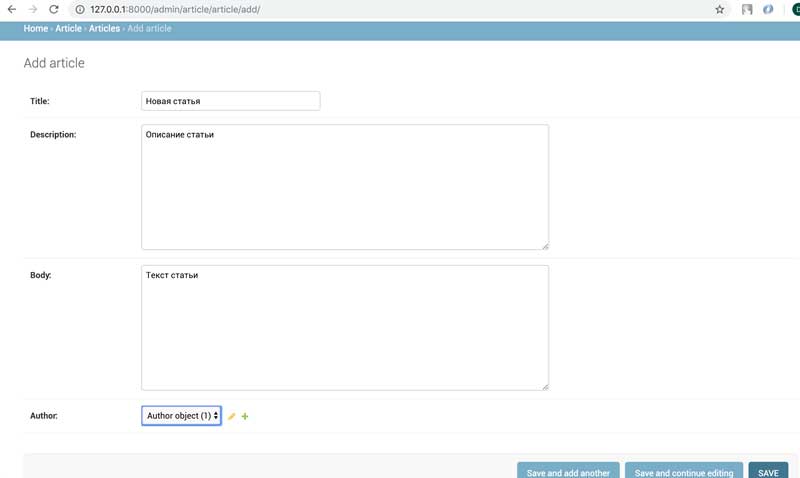
Теперь каждый раз, когда мы будем заходить по адресу http://127.0.0.1:8000/api/articles/ , мы получим доступ ко всем URL-адресам, которые мы определяем внутри **articles/urls.py** .



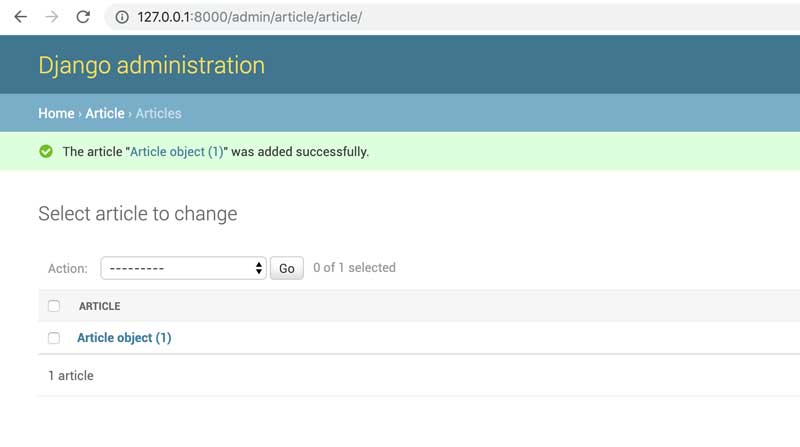
Теперь у нас метод API через который мы можем просматривать все статьи в базе данных.

В настоящее время в нашей базе данных нет статей, поэтому мы получаем пустой список. Одной из причин, по которой мы добавили наши модели  Article и  Author на страницу администратора, была возможность добавлять  авторов  и статьи  непосредственно в нашу базу данных через приложение администратора.

Теперь давайте создадим новые объекты в таблице **Author** и  **Article** в приложение администратора.

После добавления одного и более автора и статьи, вы может заметить что их отображение не совсем так как хотелось бы



**Article object(1)** не особо информативное отображение имени объекта. Давай те изменим это, добавим отображение имени статьи. Для этого вернемся в наш **article/models.py** и добавим метод \_\_str\_\_ Этот метод даст нам удобочитаемое отображение объектов  Article.

**class** Article(models.Model):

title = models.CharField(max\_length=120)

description = models.TextField()

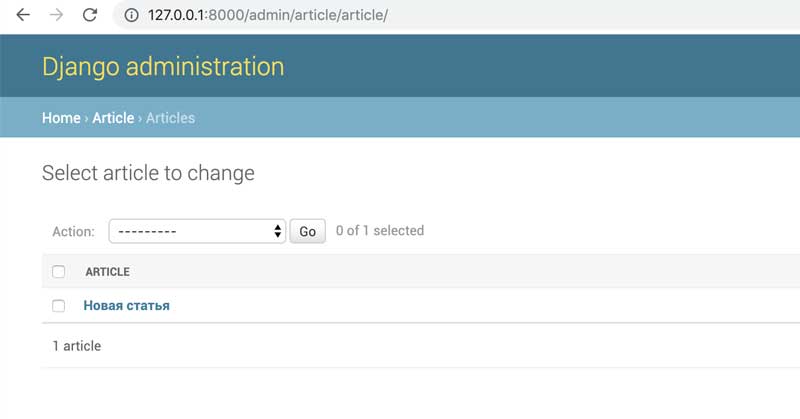
body = models.TextField()

author = models.ForeignKey('Author', related\_name='articles', on\_delete=models.CASCADE)

**def** \_\_str\_\_(self):

**return** self.title

После добавления этого метода в класс Article, наша админка будет выглядеть как то так:



Добавьте такой же метод в класс Author.

**class** Author(models.Model):

name = models.CharField(max\_length=255)

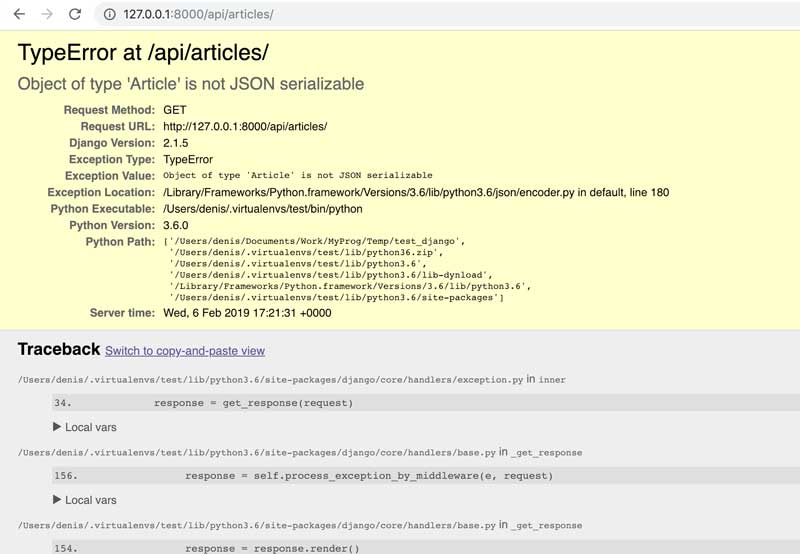
email = models.EmailField()

**def** \_\_str\_\_(self):

**return** self.title

Теперь у нас должно быть несколько статей  в таблице **Articles**, проверим работает ли доступ API к статьям. Перейдите по адресу http://127.0.0.1:8000/api/articles

Опс!!!



TypeError. Серьезно? После всего того что мы сделали?  
Ошибка возникла в строке return Response({"articles": articles}) где мы пытаемся сериализовать (то есть сконвертировать из объектов в формат JSON) список объектов articles . Но так как мы еще не указали класс для сериализации объектов статей, мы получили ошибку.

Чтобы это исправить, познакомимся с понятием  **[Serializers](https://www.django-rest-framework.org/api-guide/serializers/" \l "serializers" \t "_blank)**.

Serializers (Сериализаторы) позволяют преобразовывать сложные данные, такие как наборы запросов [querysets](https://docs.djangoproject.com/en/2.1/ref/models/querysets/" \t "_blank) и объекты моделей, в типы данных Python, которые затем можно легко преобразовать в JSON, XML или другие content types.

Теперь, давайте создадим сериалайзер, который преобразует наши статьи в список Python, который мы можем вернуть в API запросе.  
Создайте новый файл в папке статей и назовите его что-то вроде  **article/serializers.py**. В этот файл добавьте следующий код:

**from** rest\_framework **import** serializers

**class** ArticleSerializer(serializers.Serializer):

title = serializers.CharField(max\_length=120)

description = serializers.CharField()

body = serializers.CharField()

Как вы можете видеть, мы пока еще не сериализуем автора (author). Мы это сделаем позже.  
Следующий шаг — нужно добавить этот сериализатор в наши представления (views) и сделать так чтобы представление сериализовало статьи. Следующий код показывает, как это делается.

**from** rest\_framework.response **import** Response

**from** *rest\_framework.views* **import** APIView

**from** *.models* **import** Article

**from** *.serializers* **import** ArticleSerializer

**class** ArticleView(APIView):

**def** get(self, request):

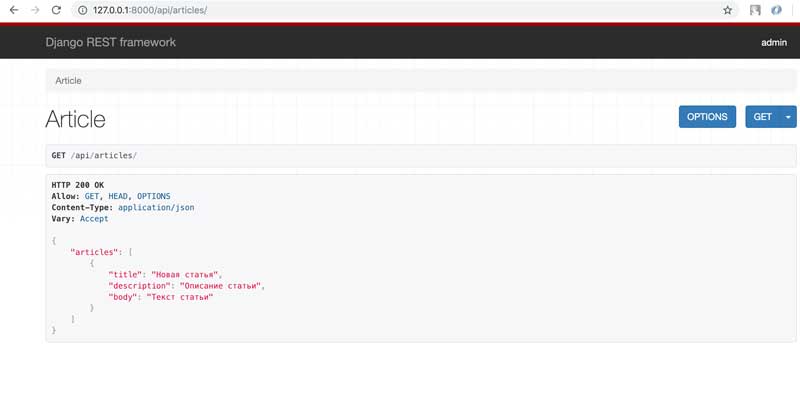
articles = Article.objects.all()

# the many param informs the serializer that it will be serializing more than a single article.

serializer = ArticleSerializer(articles, many=**True**)

**return** Response({"articles": serializer.data})

После этого снова запустите сервер и перейдите к http://127.0.0.1:8000/api/articles/ . Должно получится что то типа такого:



Далее добавим метод API для создание статьи. Можно было бы создать еще один класс с методом post и зарегистрировать его в URL-адресах так же, как мы делали для метода get. Однако вместо создания нового класса, воспользуемся тем что  APIView  позволяет нам указать несколько HTTP-методов для одного класса. Поэтому добавим метод post  внутри нашего  ArticleView .

**class** ArticleView(APIView):

**def** get(self, request):

articles = Article.objects.all()

serializer = ArticleSerializer(articles, many=**True**)

**return** Response({"articles": serializer.data})

**def** post(self, request):

article = request.data.get('article')

# Create an article from the above data

serializer = ArticleSerializer(data=article)

**if** serializer.is\_valid(raise\_exception=**True**):

article\_saved = serializer.save()

**return** Response({"success": "Article '{}' created successfully".format(article\_saved.title)})

Как видите, мы используем созданный ранее сериализатор, чтобы создать новый объект статьи из данных которые мы получаем от пользователя. Как мы уже говорили ранее, мы проигнорировали поле автора в нашем сериализаторе, и поэтому оно не возвращается в полученном ответе. Чтобы мы могли использовать наш сериализатор для создания статей, нам нужно добавить поле  author\_id  в сериализатор, а затем нам потребуется реализовать метод create в сериализаторе, который сообщит сериализатору, что делать, когда вызывается метод  save сериализатора.

Обновим наш ArticleSerializer, чтобы он выглядел следующим образом.

**from** rest\_framework **import** serializers

**from** *.models* **import** Article

**class** ArticleSerializer(serializers.Serializer):

title = serializers.CharField(max\_length=120)

description = serializers.CharField()

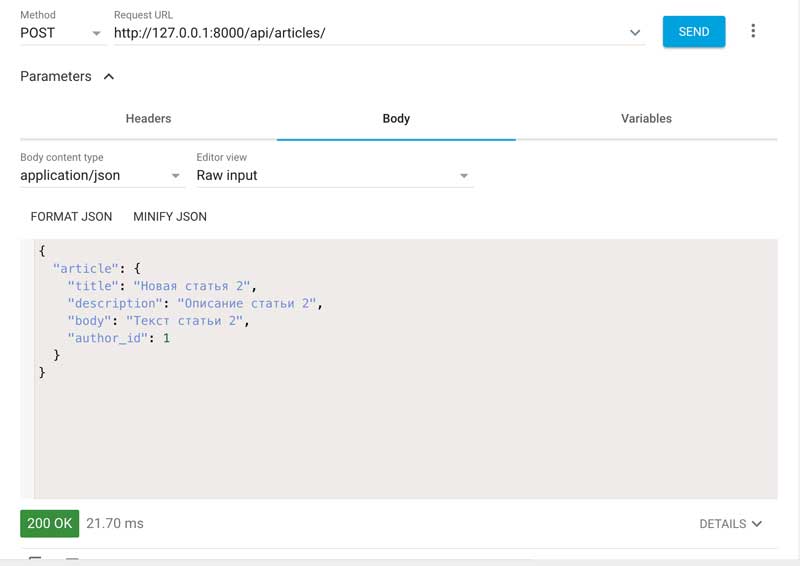
body = serializers.CharField()

author\_id = serializers.IntegerField()

**def** create(self, validated\_data):

**return** Article.objects.create(\*\*validated\_data)

Учитывая, что мы создали автора с панели администратора, теперь вы можете использовать **postman** или любой другой REST клиент для создания статьи (например какой нибудь REST плагин в хроме). Пример такого запроса показан ниже.



Благодаря этому каждый теперь сможет создавать статьи используя API. Теперь создадим возможность редактирования статьи.

Для этого мы обновим API  articles , чтобы пользователи могли обновить статью, отправив запрос **PUT**. Сначала добавим новый path  в файл **article/urls.py**.

**from** django.urls **import** path

**from** *.views* **import** ArticleView

app\_name = "articles"

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

urlpatterns = [

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

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

]

Далее, добавим метод  update  в наш сериализатор, который сделает обновление за нас. Теперь наш код сериалайзера должен выглядеть следующим образом

**from** rest\_framework **import** serializers

**from** *.models* **import** Article

**class** ArticleSerializer(serializers.Serializer):

title = serializers.CharField(max\_length=120)

description = serializers.CharField()

body = serializers.CharField()

author\_id = serializers.IntegerField()

**def** create(self, validated\_data):

**return** Article.objects.create(\*\*validated\_data)

**def** update(self, instance, validated\_data):

instance.title = validated\_data.get('title', instance.title)

instance.description = validated\_data.get('description', instance.description)

instance.body = validated\_data.get('body', instance.body)

instance.author\_id = validated\_data.get('author\_id', instance.author\_id)

instance.save()

**return** instance

Что происходит в методе  update ? В том случае если мы что то передаем в экземпляр статьи, который мы хотим обновить, мы переназначаем это значение, в противном случае мы сохраняем старое значение атрибута.

Теперь создадим обработку запроса на обновление статьи. Мы должны определить метод  put  в нашем  ArticleView , этот метод должен принять параметр  pk из URL, найти требуемый экземпляр из базы и запустить сериалайзер на обновление. Внесем соотвествующие изменения в файл **article.views.py**.

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

**from** *rest\_framework.response* **import** Response

**from** *rest\_framework.views* **import** APIView

**from** *.models* **import** Article

**from** *.serializers* **import** ArticleSerializer

**class** ArticleView(APIView):

**def** get(self, request):

articles = Article.objects.all()

serializer = ArticleSerializer(articles, many=**True**)

**return** Response({"articles": serializer.data})

**def** post(self, request):

article = request.data.get("article")

# Create an article from the above data

serializer = ArticleSerializer(data=article)

**if** serializer.is\_valid(raise\_exception=**True**):

article\_saved = serializer.save()

**return** Response({"success": "Article '{}' created successfully".format(article\_saved.title)})

**def** put(self, request, pk):

saved\_article = get\_object\_or\_404(Article.objects.all(), pk=pk)

data = request.data.get('article')

serializer = ArticleSerializer(instance=saved\_article, data=data, partial=**True**)

**if** serializer.is\_valid(raise\_exception=**True**):

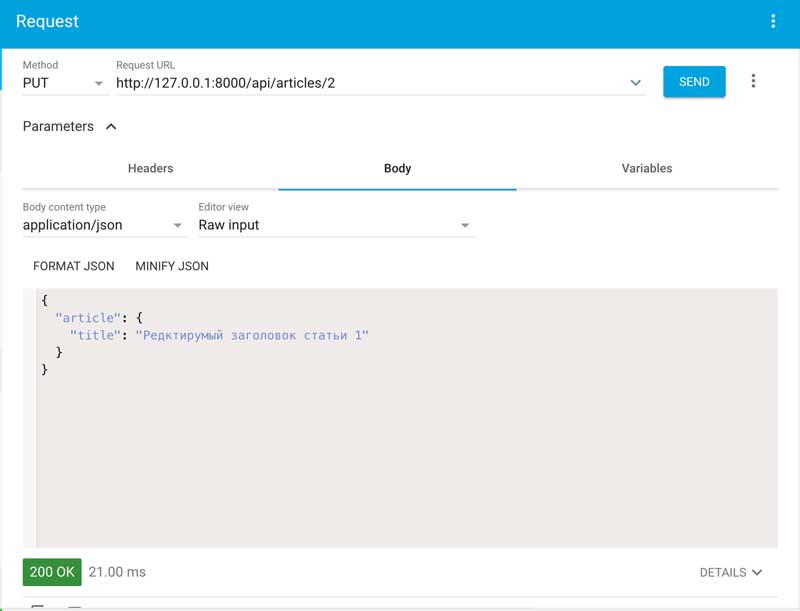
article\_saved = serializer.save()

**return** Response({

"success": "Article '{}' updated successfully".format(article\_saved.title)

})

Мы передаем partial=True в сериализатор, поскольку хотим иметь возможность обновлять только некоторые поля, но не обязательно все сразу.   
Используя **Postman** или любой другой инструмент, теперь можно обновить статью, отправив запрос по адресу http://127.0.0.1:8000/articles/<article id> с данными, которые вы хотите обновить.



Последнее что нам осталось добавить для создания полноценного CRUD API, это метод удаления. Для этого мы создадим метод  delete  в APIView , который будет принимать  id  статьи, в качестве аргумента.

**def** delete(self, request, pk):

# Get object with this pk

article = get\_object\_or\_404(Article.objects.all(), pk=pk)

article.delete()

**return** Response({

"message": "Article with id `{}` has been deleted.".format(pk)

}, status=204)

В методе delete все, что мы делаем — это получаем статью из базы, и если она существует, удаляем ее и затем возвращаем ответ пользователю.

Теперь у нас есть полностью функционирующие API, с помощью которого мы можем выполнить все основные задачи API, т. е. Операции Create Read Update Delete (**CRUD**).