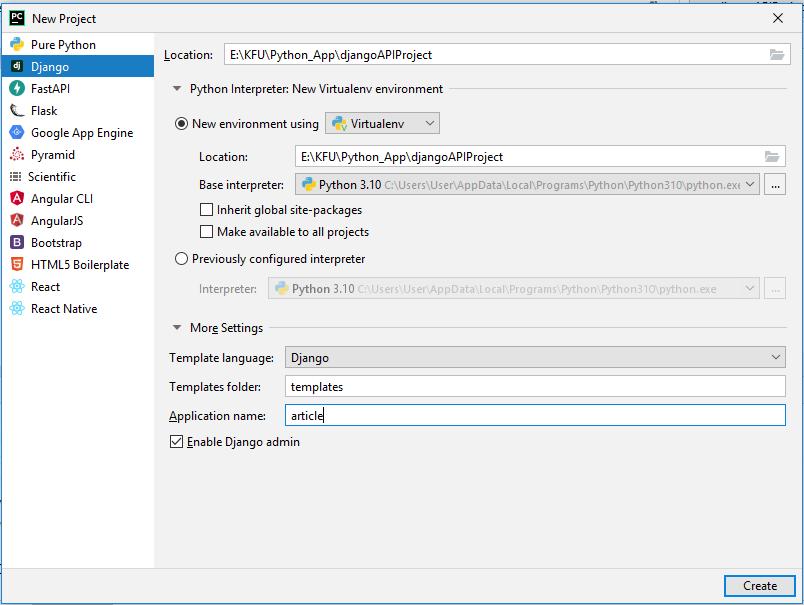
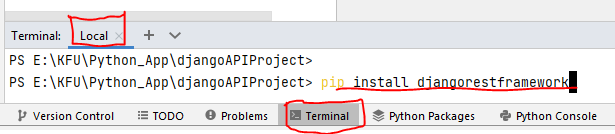
**Lecture 8. Создание Django API используя Django Rest Framework**

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. Откроем PyCharm и создадим новый Django проект с именем djangoAPIProject.



Так как мы будем использовать Django Rest Framework (DRF) для создания API, установите и его. Для этого откроем терминал программы и добавим в нем следуюший код и нажимаем на кнопку Enter.



|  |
| --- |
| pip install djangorestframework |

Теперь скажите Django, что мы установили REST Framework в mysite/settings.py:

|  |
| --- |
| INSTALLED\_APPS = [  …………………   **'rest\_framework'**, ] |

В итоге получим:

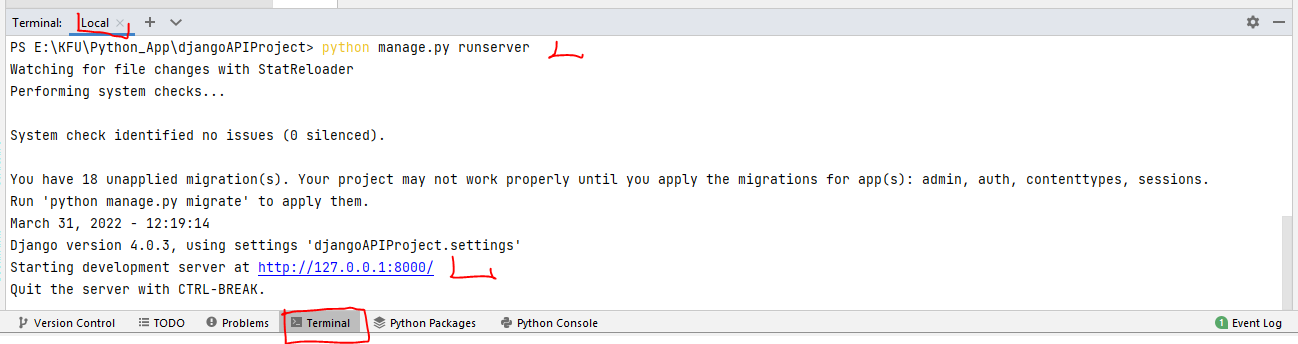
|  |
| --- |
| INSTALLED\_APPS = [  **'django.contrib.admin'**,  **'django.contrib.auth'**,  **'django.contrib.contenttypes'**,  **'django.contrib.sessions'**,  **'django.contrib.messages'**,  **'django.contrib.staticfiles'**,  **'article.apps.ArticleConfig'**,  **'rest\_framework'**, ] |

Теперь все готово что бы создать наш первый DRF API.

Если есть необходимость создать новый проект Django, запустите следующую команду:

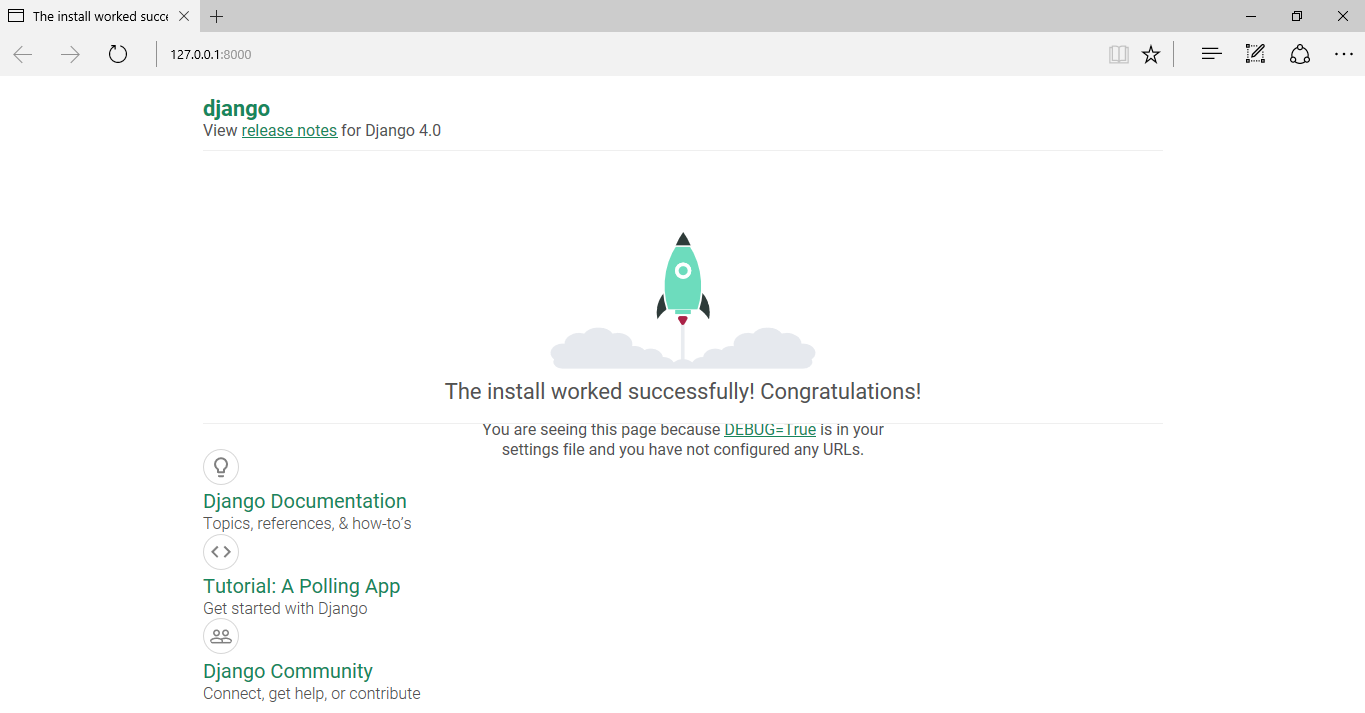
|  |
| --- |
| django-admin startproject *djangoAPIProject* |

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



|  |
| --- |
| python manage.py runserver |

Чтобы открыть проект в браузере нажимаема на ссылку http://127.0.0.1:8000/



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

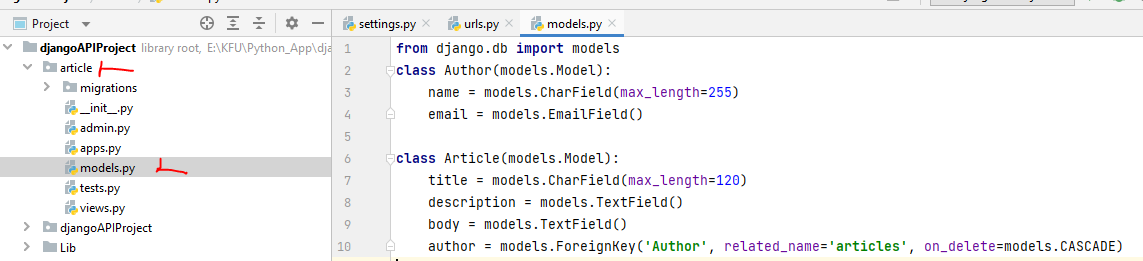
Начнем с создания модели авторов:

|  |
| --- |
| **from** django.db **import** models **class** Author(models.Model):  name = models.CharField(max\_length=255)  email = models.EmailField() |

Затем ниже создадим модель статей:

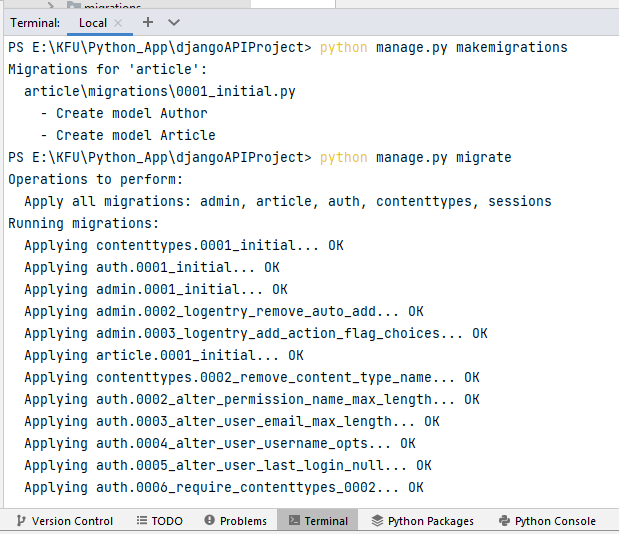
|  |
| --- |
| **class** Article(models.Model):  title = models.CharField(max\_length=120)  description = models.TextField()  body = models.TextField()  author = models.ForeignKey(**'Author'**, related\_name=**'articles'**, on\_delete=models.CASCADE) |

В итоге получим:



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 |

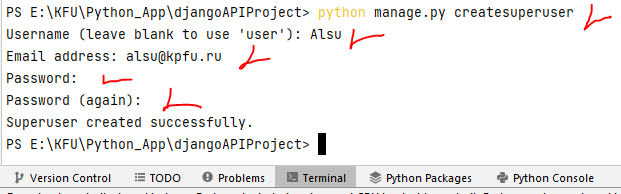


Таким образом будет создана база данных по умолчанию (на sqlite), со всеми требуемыми таблицами. Запустите команду:

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.



**Примечание**. *При вводе пароля пароль здесь не отображается, а сохраняется в памяти.*

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.

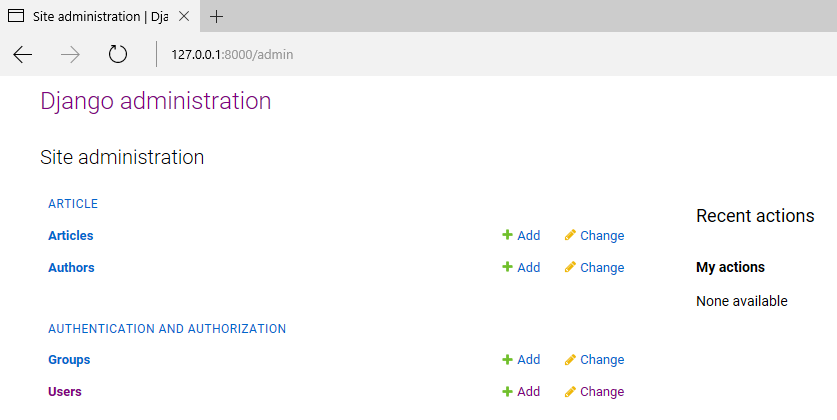


Далее, нужно зарегистрировать наши модели, чтобы они появлялись на этой странице.

Для этого мы откройте файл article/admin.py и внесите следующие изменения:

|  |
| --- |
| **from** django.contrib **import** admin **from** article.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** article.models **import** Article **class** ArticleView(APIView):  **def** get(self, request):  articles = Article.objects.all()  **return** Response({**"article"**: 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.

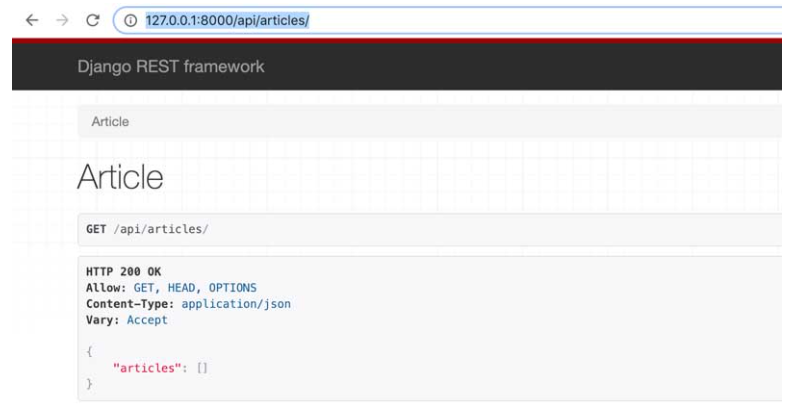
|  |
| --- |
| **from** django.urls **import** path **from** article.views **import** ArticleView app\_name = **"articles"** *# app\_name will help us do a reverse look-up latter.* urlpatterns = [  path(**'articles/'**, ArticleView.as\_view()), ] |

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/> (<http://127.0.0.1:8000/api/articles/> ) , мы получим доступ ко всем URL-адресам, которые мы определяем внутри articles/urls.py.

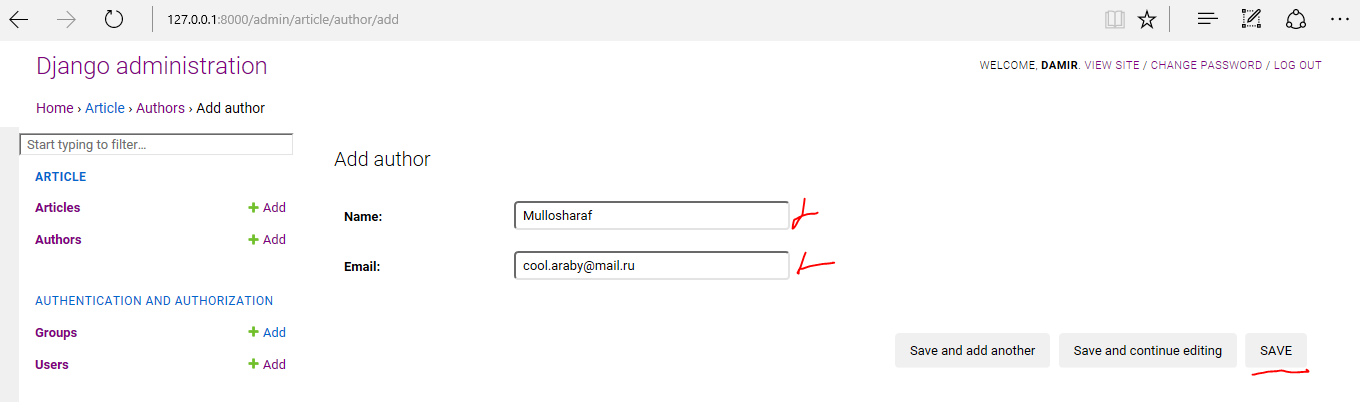


Теперь у нас метод API через который мы можем просматривать все статьи в базе данных.

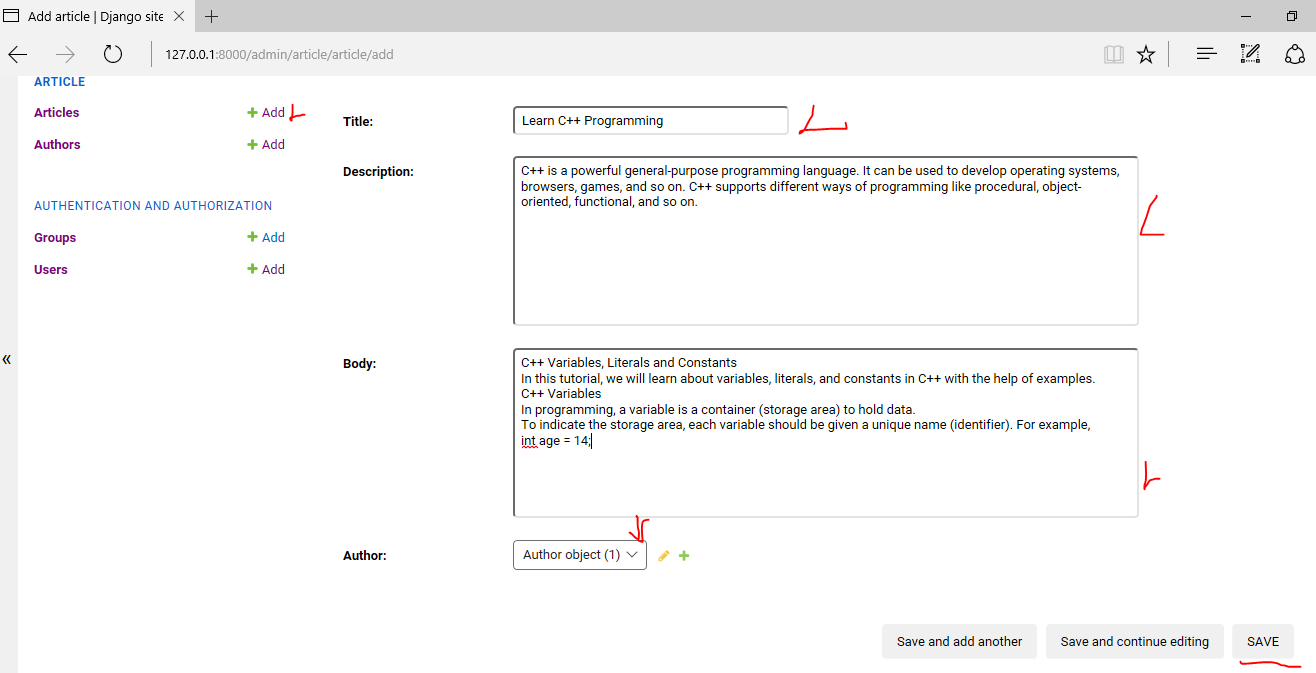
В настоящее время в нашей базе данных нет статей, поэтому мы получаем пустой список. Одной из причин, по которой мы добавили наши модели Article и Author на страницу администратора, была возможность добавлять авторов и статьи непосредственно в нашу базу данных через приложение администратора.

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

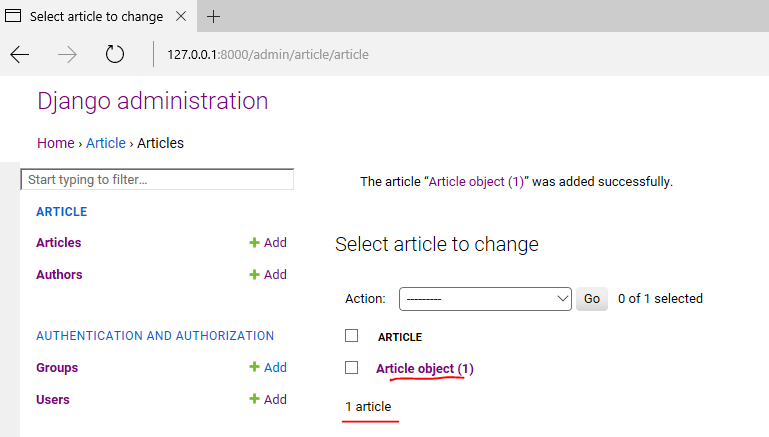
http://127.0.0.1:8000/admin/article/author/add/



http://127.0.0.1:8000/admin/article/article/add/



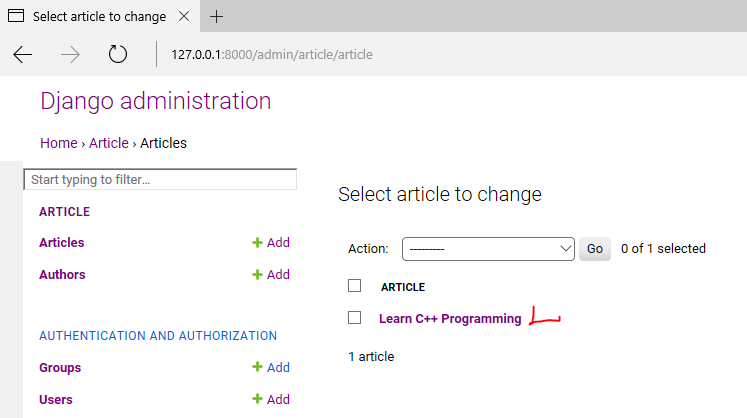
После добавления одного и более автора и статьи, вы может заметить что их отображение не совсем так как хотелось бы



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 |

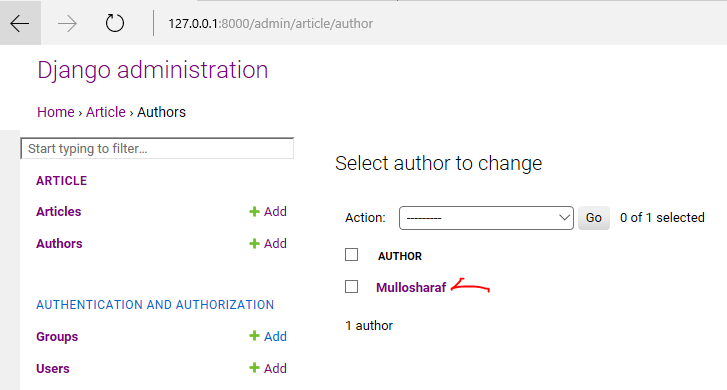
После добавления этого метода в класс Article, наша админка будет выглядеть как то так:



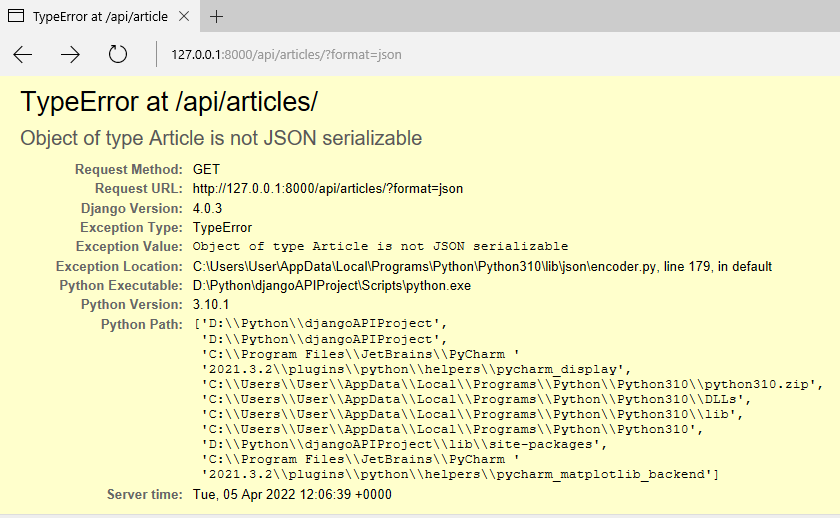
Добавьте такой же метод в класс Author.

|  |
| --- |
| **class** Author(models.Model):  name = models.CharField(max\_length=255)  email = models.EmailField()  **def** \_\_str\_\_(self):  **return** self.title |

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



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> или <http://127.0.0.1:8000/api/articles/?format=json>



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.

Serializers (Сериализаторы) позволяют преобразовывать сложные данные, такие как наборы запросов querysets 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:

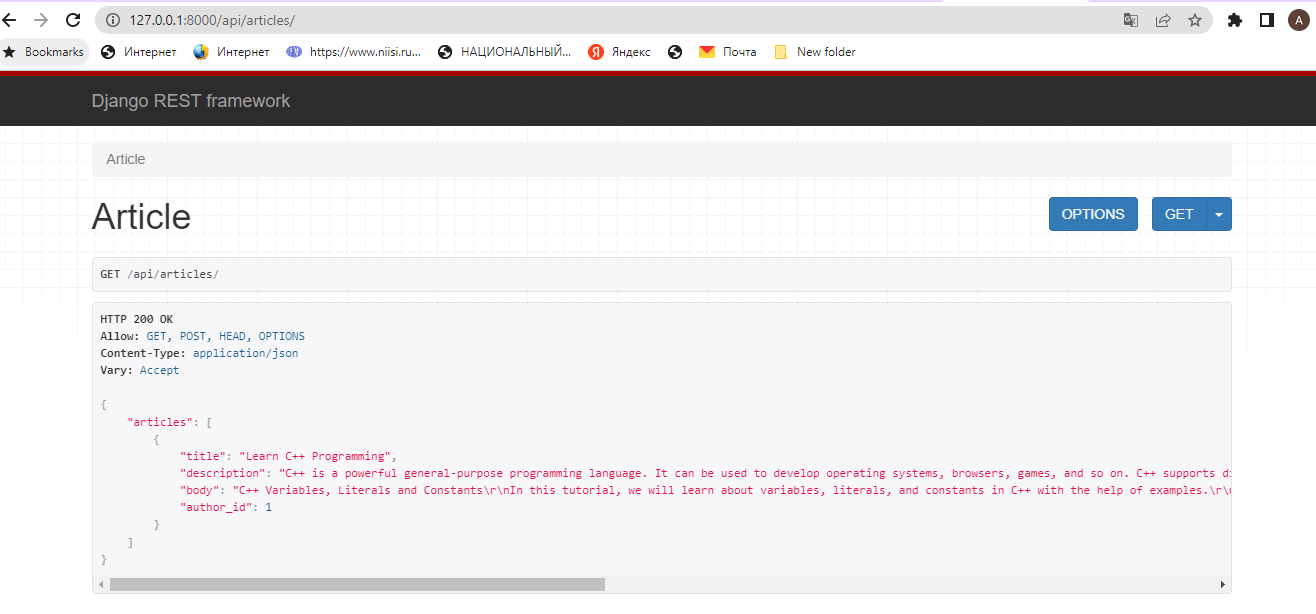
|  |
| --- |
| **from** rest\_framework **import** serializers  **class** ArticleSerializer(serializers.Serializer):  title = serializers.CharField(max\_length=120)  description = serializers.CharField()  body = serializers.CharField() |

Как вы можете видеть, мы пока еще не сериализуем автора (author). Мы это сделаем позже.

Следующий шаг — нужно добавить этот сериализатор в наши представления (article/views.py) 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()   *# параметр many сообщает сериализатору, что он будет сериализовать более одной статьи.* 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/> или http://127.0.0.1:8000/api/articles/?format=json. Должно получится что то типа такого:



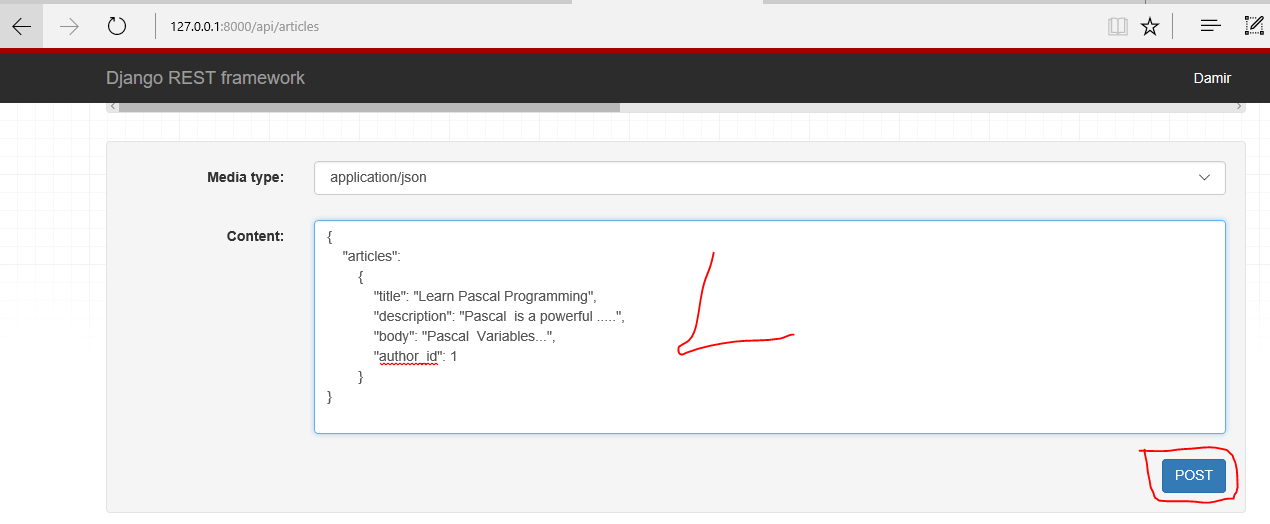
Далее добавим метод API для создание статьи. Можно было бы создать еще один класс с методом post и зарегистрировать его в URL-адресах так же, как мы делали для метода get. Однако вместо создания нового класса, воспользуемся тем что APIView позволяет нам указать несколько HTTP-методов для одного класса.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

|  |
| --- |
| **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()  *# параметр many сообщает сериализатору,*  *#что он будет сериализовать более одной статьи.* serializer = ArticleSerializer(articles, many=**True**)  **return** Response({**"articles"**: serializer.data})    **def** post(self, request):  article = request.data.get(**'articles'**)  *#Создать статью из приведенных выше данных* 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 поможет нам сделать обратный поиск последнего.* 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 (article/ serializers.py) должен выглядеть следующим образом

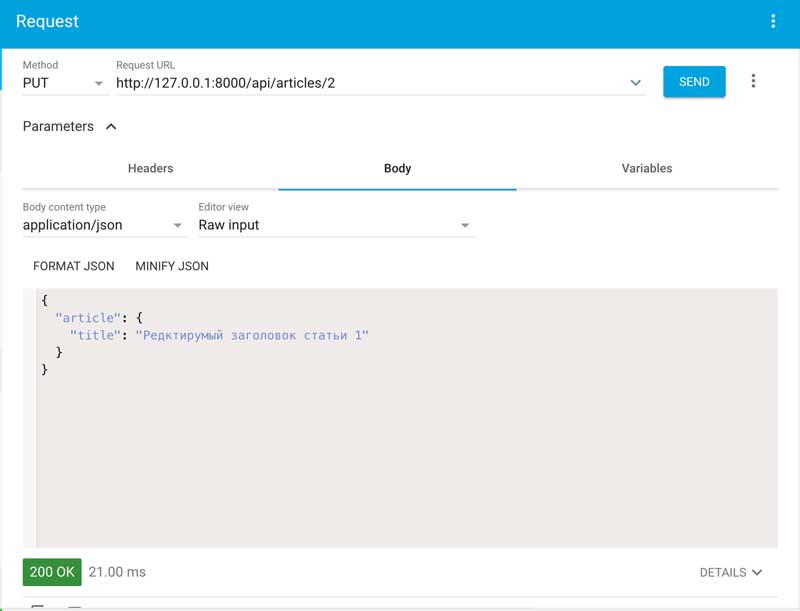
|  |
| --- |
| **from** rest\_framework **import** serializers **from** .models **import** Article **class** ArticleSerializer(serializers.Serializer):  title = serializers.CharField(max\_length=120)  description = serializers.CharField()  body = serializers.CharField()  author\_id = serializers.IntegerField()   **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(**"articles"**)  *# 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(**'articles'**)  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/api/articles/<article id> с данными, которые вы хотите обновить.

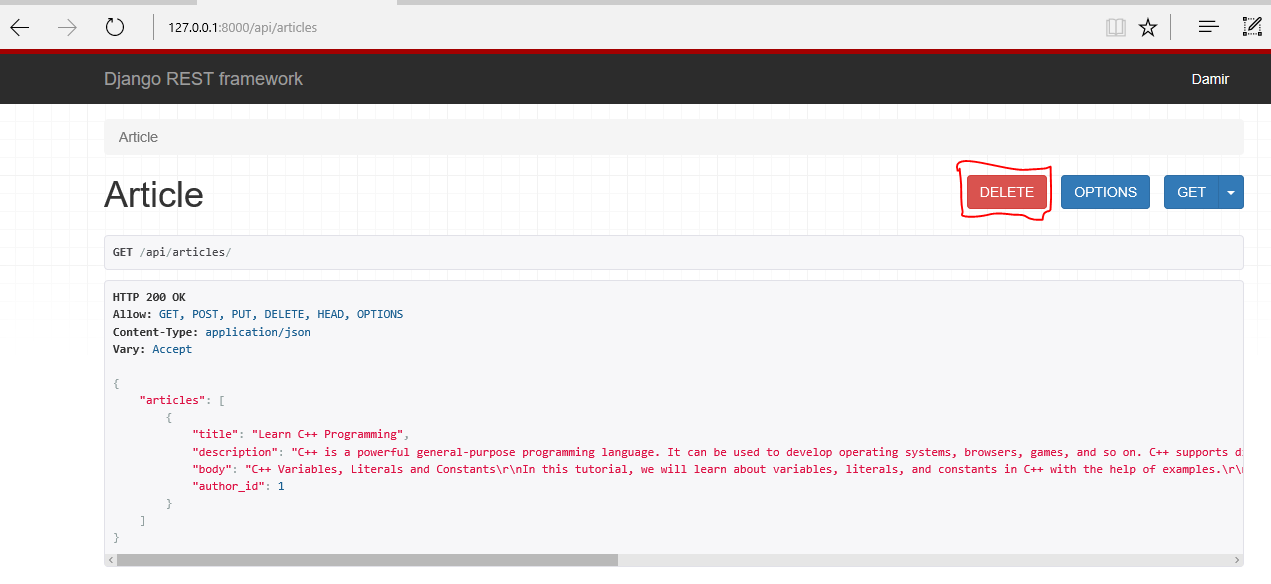


Последнее что нам осталось добавить для создания полноценного CRUD API, это метод удаления. Для этого мы создадим метод delete в APIView , который будет принимать id статьи, в качестве аргумента.

|  |
| --- |
| **def** delete(self, request, pk):  *# Получить объект с этим 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).



**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/#genericapiview)  вместо [**APIView**](https://www.django-rest-framework.org/api-guide/views/)**.**

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/#mixins) , 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 получения 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 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**

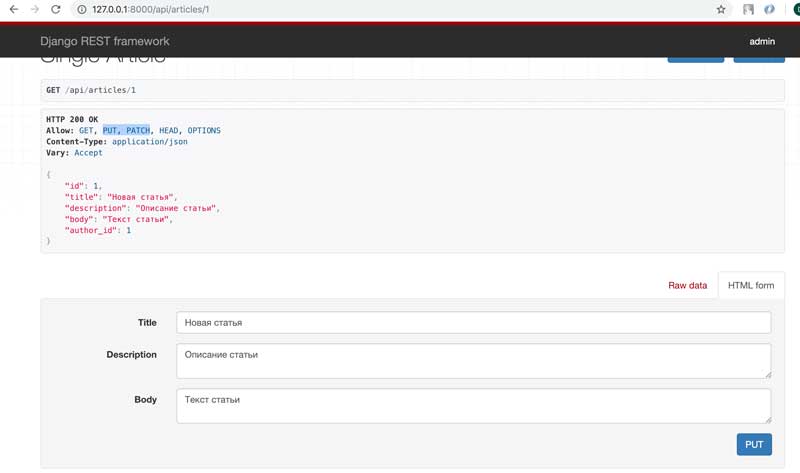
|  |
| --- |
| from django.urls import path  from .views import ArticleView, SingleArticleView  app\_name = "articles"  # app\_name will help us do a reverse look-up latter.  urlpatterns = [  path('articles/', ArticleView.as\_view()),  path('articles/<int:pk>', SingleArticleView.as\_view()),  ] |

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

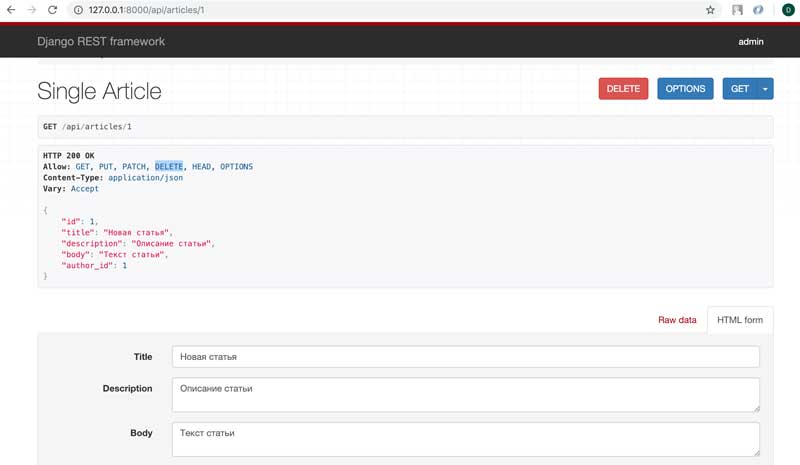


Теперь для таблицы Article у нас есть методы **create, list, retrieve и update**.

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. Узнаем об этом [следующей статье](https://webdevblog.ru/sozdanie-django-api-ispolzuya-djangorestframework-chast-3/) о использование viewsets.

**Создание Django API используя Django Rest Framework часть 3**

Это третья часть из серии статей про Django API. Первая часть находится [здесь](https://webdevblog.ru/sozdanie-django-api-ispolzuya-django-rest-framework-apiview/). Вторая [здесь](https://webdevblog.ru/sozdanie-django-api-ispolzuya-djangorestframework-chast-2/). В этой статье мы рассмотрим использование наборы представлений класс [viewsets](https://www.django-rest-framework.org/api-guide/viewsets/) от Django REST Framework**.** Основное отличие **viewsets** от **GenericAPIView** и **APIView**, в том что это еще один дополнительный абстрагирующий класс, который с одной стороны позволяет уменьшить базовый код а с другой имеет большие возможности по кастомизации и расширению.

И так начнем, с изменений в файле представления. Используем базовый класс **viewsets.ViewSet** унаследуем от него наш класс ArticleView и определим методы получения всех статей (list) и одну выбранную статью (retrieve). Для этого внесем следующие изменения в **article/views.py**:

|  |
| --- |
| from django.shortcuts import get\_object\_or\_404  from rest\_framework import viewsets  from rest\_framework.response import Response  from .models import Article  from .serializers import ArticleSerializer  **class** ArticleView(viewsets.ViewSet):  """  A simple ViewSet that for listing or retrieving users.  """  **def** list(self, request):  queryset = Article.objects.all()  serializer = ArticleSerializer(queryset, many=**True**)  **return** Response(serializer.data)  **def** retrieve(self, request, pk=None):  queryset = Article.objects.all()  user = get\_object\_or\_404(queryset, pk=pk)  serializer = ArticleSerializer(user)  **return** Response(serializer.data) |

Далее нам нужно связать наши методы list и retrieve с URL API. Для этого внесите изменения в **article/urls.py**:

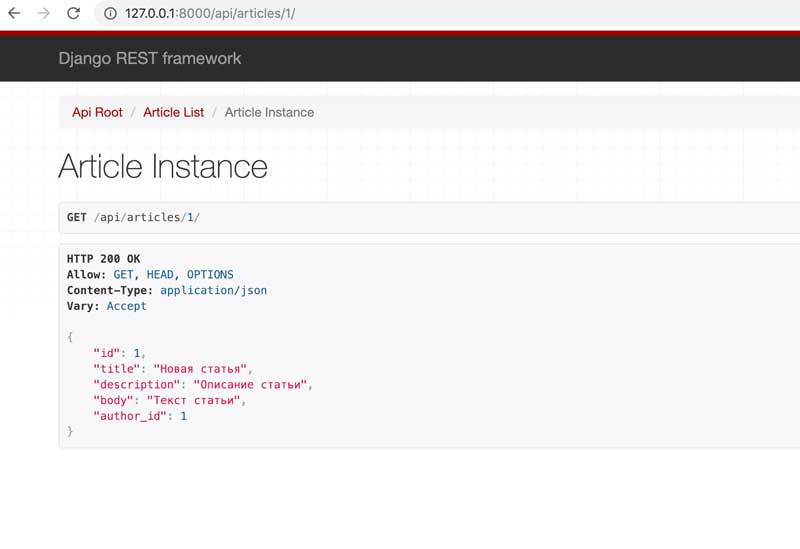
|  |
| --- |
| from django.urls import path  from .views import ArticleView  app\_name = "articles"  # app\_name will help us do a reverse look-up latter.  urlpatterns = [  path('articles/', ArticleView.as\_view({'get': 'list'})),  path('articles/<int:pk>', ArticleView.as\_view({'get': 'retrieve'})),  ] |

Если теперь обратится по URL: http://127.0.0.1:8000/api/articles/ или http://127.0.0.1:8000/api/articles/1/ то должно все работать как раньше.

Теперь воспользуется новым для нас классом роутера DRF **DefaultRouter**. Этот класс позволяет значительно уменьшить код в URL. Зарегистрируем наш класс **ArticleView** в этом классе и позволяем ему теперь автоматически генерировать обработчики url. Удалите весь старый код из **article/urls.py** и внесите следующие изменения в этот файл:

|  |
| --- |
| **from** rest\_framework.routers **import** DefaultRouter  **from** *.views* **import** ArticleView  router = DefaultRouter()  router.register(r'articles', ArticleView, basename='user')  urlpatterns = router.urls |

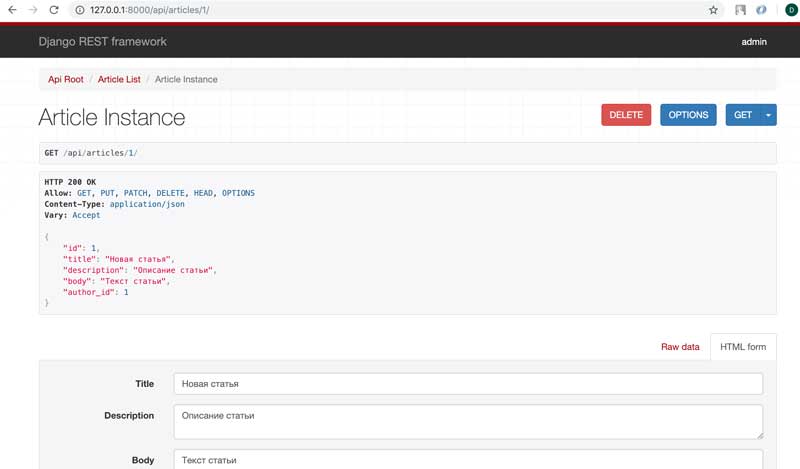
Теперь если обратится по URL: http://127.0.0.1:8000/api/articles/ или http://127.0.0.1:8000/api/articles/1/ то опять все должно работать как раньше. Но обратите внимание мы реализовали только два метода API получения списка статей и получение одной выбранной статье.



Что бы реализовать методы создания, редактирования и удаления мы можем определить эти методы вручную или воспользоваться классом viewsets.ModelViewSet в котором уже все сделано за нас. Внесите следующие изменения в файл **article/views.py**:

|  |
| --- |
| **from** rest\_framework **import** viewsets  **from** *.models* **import** Article  **from** *.serializers* **import** ArticleSerializer  **class** ArticleViewSet(viewsets.ModelViewSet):  serializer\_class = ArticleSerializer  queryset = Article.objects.all()  И соотвествующие изменения в файл **article/urls.py**  from rest\_framework.routers import DefaultRouter  from .views import ArticleViewSet  router = DefaultRouter()  router.register(r'articles', ArticleViewSet, basename='user')  urlpatterns = router.urls |

Теперь мы можем создавать, редактировать и удалять



Используя **viewsets** мы буквально в пару строк получили полностью рабочий CRUD для выбранной таблицы.

На этом я пожалуй хотел завершить эту статью. В этой серии статей мы очень кратко рассмотрели и сравнили возможности классов DRF **APIView**, **GenericAPIView**, **ViewSets** для создания CRUD. На самом деле у этих классов гораздо больше возможностей и особенностей их применения и если вы захотите изучить их глубже обратитесь к официальной [документации DRF](https://www.django-rest-framework.org/).

1. <https://medium.com/the-andela-way/creating-a-django-api-using-django-rest-framework-apiview-b365dca53c1d>
2. <https://medium.com/the-andela-way/creating-a-djangorest-api-using-djangorestframework-part-2-1231fe949795>
3. <https://medium.com/django-rest-for-not-beginners/generic-viewsets-an-introduction-be8712a37e6d>