Методическое пособие:

Знакомство с фреймворком Django REST framework

Автор: Грушевский Юрий Викторович

Контакты: jurapro@yandex.ru

Оглавление

[**1. Введение**](#_860qtefy4d9v) **3**

[Для кого предназначено данное методическое пособие](#_fllkl791hghn) 3

[Необходимое программное обеспечение](#_5iht9ouf8czo) 3

[Необходимые начальные знания](#_8pofxy8nubd) 3

[**2. Сериализация**](#_qk931d4mzws7) **4**

[Установка и первоначальная настройка](#_dlfng9b3hxpt) 4

[Создание модели для работы](#_77ir54it2rwv) 6

[Создание класса сериализатора](#_p4jd4bxd1ck9) 7

[Использование ModelSerializer](#_d5uyuhj3zpms) 8

[Разработка простых представлений](#_4judp0qc41w7) 8

[**3. Объекты Requests и Responses**](#_gk4euqj0oo0u) **13**

[Объект Request](#_lhr129nmo7j2) 13

[Объетк Response](#_q5wcp9izy6og) 13

[Коды статуса](#_6p2lxembehh7) 13

[Обертки представлений для API](#_eptccjk2swoo) 13

[Рефакторинг кода представлений](#_rzvxybnjqthd) 13

[**4. Представления на основе классов**](#_s59js7hpxyuz) **18**

[Рефакторинг представлений](#_i3tfkqiyxo98) 18

[Использование миксинов](#_qa7sn9jlq0j0) 19

[Использование общего класса](#_j2tohmy40i0v) 21

[**5. Аутентификация и авторизация**](#_hhzfed8zsr6b) **22**

[Обновление модели](#_h13quwiqtnvw) 22

[Добавление представлений для пользователя](#_17h0wlbx0009) 23

[Ассоциация сниппетов с пользователями](#_mhh9zckm1oa8) 25

[Обновление сериализатора](#_sydonm6rmgkg) 25

[Добавление необходимых разрешений](#_v62bigize802) 26

[Добавление возможности аутентификации](#_jo2q7gs6t82b) 26

[Добавление ограничений на уровне объекта](#_axgpoww9goao) 28

[Аутентификация с помощью API](#_54kfql63lfrj) 29

[**6. Связанные объекты**](#_6x7ncz13f8zb) **31**

[Создание единой точки входа](#_8z2jo6ydz8au) 31

[Создание точки входа для html-представления сниппетов](#_9cohcrkoo0th) 31

[Гиперссылки в API](#_wmcd8sheola) 32

[Добавление имен для маршрутов](#_58covnhchjbh) 33

[**7. Классы ViewSets и Routes**](#_iein6n264t3o) **35**

[Рефакторинг представлений](#_v5jcgdgksin) 35

[Явная привязка ViewSets к URL-адресам](#_sou4ugvaenrm) 36

[Использование Routers](#_h2w6dp7ndq44) 37

[**8. Использованные источники**](#_1w6zud1s8lrj) **39**

# 

# 1. Введение

## Для кого предназначено данное методическое пособие

Данное пособие не ставит перед собой цель показать основные возможности фреймворка Django REST framework, его можно позиционировать скорее как методический материал, с которым можно ознакомится, чтобы составить общее впечатление об этом фреймворке. Он по возможности максимально избавлен от сложных решений в угоду простоте и скорости разработки. Подойдет для самостоятельного изучения и использования как основы для организации учебных практик в учебных заведениях.

## Необходимое программное обеспечение

В качестве интерпретатора был использован Python 3.8, который можно загрузить на официальном сайте - <https://www.python.org/downloads/>.

В роли среды разработки была выбрана программа JetBrains PyCharm, так как это один из самых популярных IDE для данного фреймворка на текущий момент. Скачать ее можно с официального сайта - <https://www.jetbrains.com/pycharm/download/>. Доступно две редакции: Professional и Community. Вторая бесплатная и ее возможностей вполне достаточно для выполнения примеров из данного пособия.

## Необходимые начальные знания

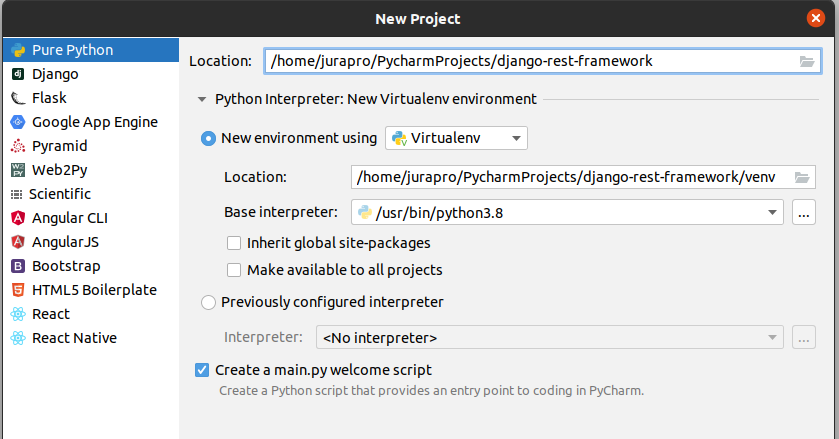
Предполагается, что читатель знаком с основами синтаксиса языка Python, основами объектно-ориентированного программирования, а также основой работы в командной строке (терминале). Кроме того, большим подспорьем будет знакомство с PyCharm и протоколом HTTP.

# 2. Сериализация

## Установка и первоначальная настройка

В этом руководстве будет рассмотрена разработка простого веб-API с использованием данного фреймворка. Функционал приложения сводится к созданию, хранению и обмену небольшими фрагментами кода между пользователями. Попутно будут рассмотрены основные компоненты и приемы разработки.

Прежде чем делать что-либо, мы создадим новый проект в PyCharm, добавив новую виртуальную среду, используя venv. Это обеспечит надежную изоляцию конфигурации нашего пакета от любых других проектов, над которыми мы работаем (Рис 2.1).



**Рис. 2.1.** Создание нового проекта и активация виртуальной среды

**Примечание:** Подробнее про venv можно прочитать здесь - <https://docs.python.org/3/library/venv.html>.

Используя терминал PyCharm установим необходимые пакеты:

*pip install django*

*pip install djangorestframework*

*pip install pygments*

**Примечание:** django - пакет базового фреймворка, djangorestframework - пакет изучаемого фреймворка, pygments - пакет для организации подсветки кода в хранимых фрагментах.

Создаем проект django и переходим внутрь созданной директории:

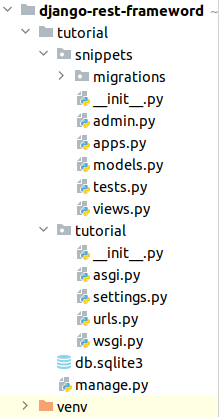
*django-admin startproject tutorial*

*cd tutorial*

Добавляем новое приложение snippets:

*python manage.py startapp snippets*

Если все сделано верно, то структура всего проекта будет как на Рис. 2.2.



**Рис. 2.2.** Первоначальная структура проекта

Осталось добавить приложение snippets и сам rest\_framework в раздел INSTALLED\_APPS настроек проекта (Листинг 2.1).

**Листинг 2.1** Файл tutorial/tutorial/settings.py

INSTALLED\_APPS = [

**...**

**'rest\_framework'**,

**'snippets.apps.SnippetsConfig'**,

]

## Создание модели для работы

Создадим для нашего приложения модель Snippets, которая будет использоваться для хранения фрагментов кода пользователями (Листинг 2.2).

**Листинг 2.2** Файл tutorial/snippets/models.py

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

**from** pygments.lexers **import** get\_all\_lexers

**from** pygments.styles **import** get\_all\_styles

LEXERS = [item **for** item **in** get\_all\_lexers() **if** item[1]]

LANGUAGE\_CHOICES = sorted([(item[1][0], item[0]) **for** item **in** LEXERS])

STYLE\_CHOICES = sorted([(item, item) **for** item **in** get\_all\_styles()])

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

created = models.DateTimeField(auto\_now\_add=**True**)

title = models.CharField(max\_length=100, blank=**True**, default=**''**)

code = models.TextField()

linenos = models.BooleanField(default=**False**)

language = models.CharField(choices=LANGUAGE\_CHOICES, default=**'python'**, max\_length=100)

style = models.CharField(choices=STYLE\_CHOICES, default=**'friendly'**, max\_length=100)

**class** Meta:

ordering = [**'created'**]

Данная модель это класс на основе которого с помощью миграции будет создана таблица в базе данных. Он представляет собой описание полей таблицы с необходимыми ограничениями.

Создадим и выполним миграцию необходимую для создания таблицы:

*python manage.py makemigrations snippets*

*python manage.py migrate*

## Создание класса сериализатора

Первое, что нам нужно для начала работы с нашим веб-API, - это предоставить способ сериализации и десериализации экземпляров сниппета в такие представления, как json. Мы можем сделать это, объявив класс сериализатора, который работает очень похоже на формы Django. Создадим файл serializers.py внутри директории snippets (Листинг 2.3).

**Листинг 2.3** Файл tutorial/snippets/serializers.py

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

**from** .models **import** Snippet, LANGUAGE\_CHOICES, STYLE\_CHOICES

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

id = serializers.IntegerField(read\_only=**True**)

title = serializers.CharField(required=**False**, allow\_blank=**True**, max\_length=100)

code = serializers.CharField(style={**'base\_template'**: **'textarea.html'**})

linenos = serializers.BooleanField(required=**False**)

language = serializers.ChoiceField(choices=LANGUAGE\_CHOICES, default=**'python'**)

style = serializers.ChoiceField(choices=STYLE\_CHOICES, default=**'friendly'**)

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

*"""*

*Create and return a new `Snippet` instance, given the validated data.*

*"""*

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

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

*"""*

*Update and return an existing `Snippet` instance, given the validated data.*

*"""*

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

instance.code = validated\_data.get(**'code'**, instance.code)

instance.linenos = validated\_data.get(**'linenos'**, instance.linenos)

instance.language = validated\_data.get(**'language'**, instance.language)

instance.style = validated\_data.get(**'style'**, instance.style)

instance.save()

**return** instance

Одна часть класса определяет какие поля будут сериализовать/ десериализовать. Методы create() и update() позволяют определить как будут обрабатываться данные при создании или обновлении сниппета.

**Примечание:** Можно было бы сократить большую часть кода, если наследоваться от класса ModelSerializer.

## Использование ModelSerializer

Созданный выше класс во многом дублирует модель Snippet. Точно так же, как Django предоставляет классы Form и ModelForm, rest\_framework включает как классы Serializer, так и классы ModelSerializer.

Заменим код SnippetSerializer на представленный в листинге 2.4.

**Листинг 2.4** Файл tutorial/snippets/serializers.py

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

**from** .models **import** Snippet

**class** SnippetSerializer(serializers.ModelSerializer):

**class** Meta:

model = Snippet

fields = [**'id'**, **'title'**, **'code'**, **'linenos'**, **'language'**, **'style'**]

По факту данный код делает то же самое, что и в листинге 2.3. Чтобы убедится в этом можно открыть в терминале Django shell (python manage.py shell) и выполнить следующий код:

*from snippets.serializers import SnippetSerializer*

*serializer = SnippetSerializer()*

*print(repr(serializer))*

**Примечание:** Для выхода из Django shell используйте сочетание клавиш ctrl+d.

## Разработка простых представлений

Создадим два представления для отображения списка сниппетов и одного сниппета (Листинг 2.5).

**Листинг 2.5** Файл tutorial/snippets/views.py

**from** django.http **import** HttpResponse, JsonResponse

**from** django.views.decorators.csrf **import** csrf\_exempt

**from** rest\_framework.parsers **import** JSONParser

**from** .models **import** Snippet

**from** .serializers **import** SnippetSerializer

@csrf\_exempt

**def** snippet\_list(request):

*"""*

*List all code snippets, or create a new snippet.*

*"""*

**if** request.method == **'GET'**:

snippets = Snippet.objects.all()

serializer = SnippetSerializer(snippets, many=**True**)

**return** JsonResponse(serializer.data, safe=**False**)

**elif** request.method == **'POST'**:

data = JSONParser().parse(request)

serializer = SnippetSerializer(data=data)

**if** serializer.is\_valid():

serializer.save()

**return** JsonResponse(serializer.data, status=201)

**return** JsonResponse(serializer.errors, status=400)

@csrf\_exempt

**def** snippet\_detail(request, pk):

*"""*

*Retrieve, update or delete a code snippet.*

*"""*

**try**:

snippet = Snippet.objects.get(pk=pk)

**except** Snippet.DoesNotExist:

**return** HttpResponse(status=404)

**if** request.method == **'GET'**:

serializer = SnippetSerializer(snippet)

**return** JsonResponse(serializer.data)

**elif** request.method == **'PUT'**:

data = JSONParser().parse(request)

serializer = SnippetSerializer(snippet, data=data)

**if** serializer.is\_valid():

serializer.save()

**return** JsonResponse(serializer.data)

**return** JsonResponse(serializer.errors, status=400)

**elif** request.method == **'DELETE'**:

snippet.delete()

**return** HttpResponse(status=204)

Функция snippet\_list() будет отвечать за отображение списка сниппетов (при обращении GET-запросом) и создание нового сниппета (POST-запрос). Соответственно функция snippet\_detail() отвечает за показ, обновление и удаление существующей записи.

**Примечание:** Декоратор @csrf\_exempt отключает проверку для функции токен CSRF. Это не самое элегантное решение, однако для простой демонстрации используем его.

Осталось добавить необходимые маршруты. Делегируем все маршруты начинающиеся на api приложению snippet (Листинг 2.6).

**Листинг 2.6** Файл tutorial/tutorial/urls.py

**from** django.urls **import** path, include

urlpatterns = [

path(**'api/'**, include(**'snippets.urls'**)),

]

В директории snippets создадим файл urls.py и добавим два необходимых маршрута (Листинг 2.7).

**Листинг 2.7** Файл tutorial/snippets/urls.py

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

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

urlpatterns = [

path(**'snippets/'**, views.snippet\_list),

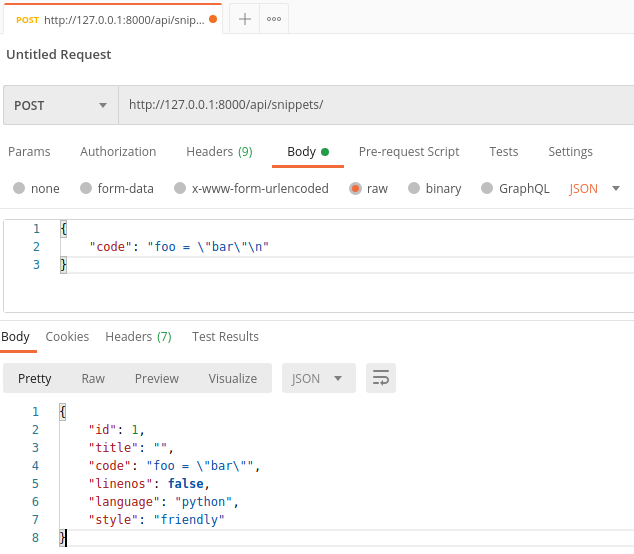
path(**'snippets/<int:pk>/'**, views.snippet\_detail),

]

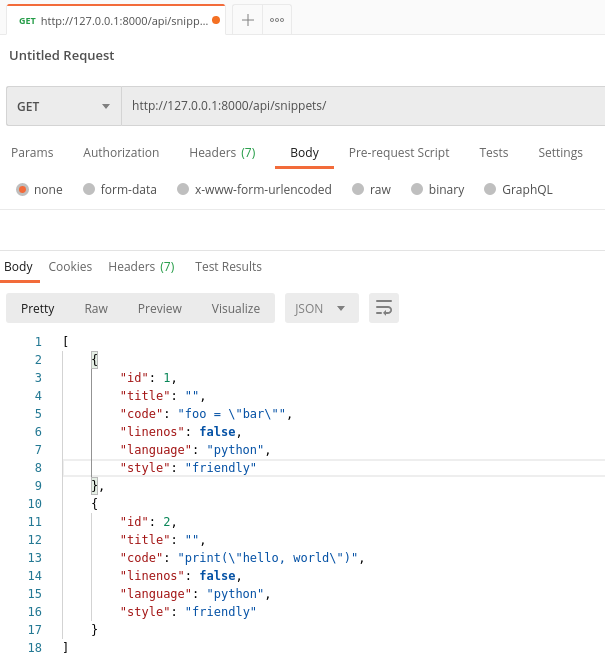
Для проверки работоспособности запустим тестовый сервер Django:

*python manage.py runserver*

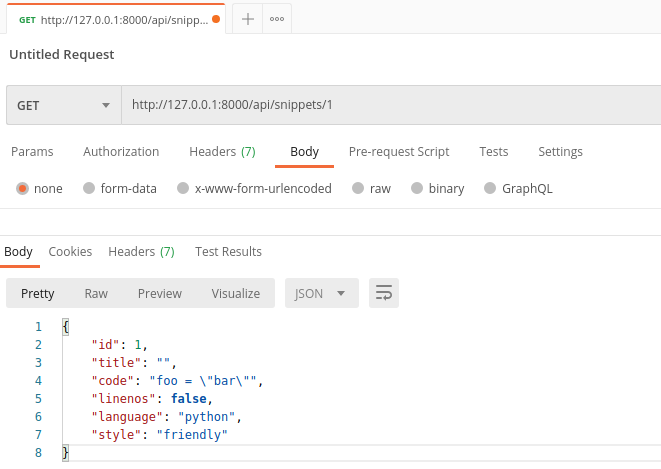
Используя программу Postman проверим работу этих маршрутов (Рис. 2.3 - 2.5).



**Рис. 2.3.** Создание нового сниппета



**Рис. 2.4.** Отображение списка сниппетов



**Рис. 2.5.** Отображение одного сниппета

# 

# 3. Объекты Requests и Responses

## Объект Request

REST framework содержит класс Request который расширяет стандартный класс HttpRequest. Основная функциональность объекта Request - это атрибут request.data, который похож на request.POST, но более удобен для работы с веб-API:

*request.POST #Обрабатывает только данные формы. Работает только для метода POST.*

*request.data #Обрабатывает произвольные данные. Работает с методами POST, PUT и PATCH.*

## Объетк Response

В фреймворке также представлен объект Response, являющийся типом TemplateResponse, который принимает неотрендеренное содержимое и использует контекст для определения правильного типа содержимого для возврата клиенту:

*return Response(data) #Выполняет рендеринг в тип данных по запросу клиента.*

## Коды статуса

Использование числовых кодов состояния HTTP в ваших представлениях не всегда очевидно. REST framework предоставляет более явные идентификаторы для каждого кода состояния, например HTTP\_400\_BAD\_REQUEST. Лучше использовать их повсюду, чем использовать числовые идентификаторы.

## Обертки представлений для API

REST framework предоставляет два декоратора для использования в представлениях:

1. @api\_view декоратор для использования в функциях;
2. Класс APIView для использования в классах.

## Рефакторинг кода представлений

Изменим функцию snippet\_list() в соответствии с материалом данной главы (Листинг 3.1).

**Листинг 3.1** Файл tutorial/snippets/views.py

**from** rest\_framework **import** status

**from** rest\_framework.decorators **import** api\_view

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

**from** .models **import** Snippet

**from** .serializers **import** SnippetSerializer

@api\_view([**'GET'**, **'POST'**])

**def** snippet\_list(request):

*"""*

*List all code snippets, or create a new snippet.*

*"""*

**if** request.method == **'GET'**:

snippets = Snippet.objects.all()

serializer = SnippetSerializer(snippets, many=**True**)

**return** Response(serializer.data)

**elif** request.method == **'POST'**:

serializer = SnippetSerializer(data=request.data)

**if** serializer.is\_valid():

serializer.save()

**return** Response(serializer.data, status=status.HTTP\_201\_CREATED)

**return** Response(serializer.errors, status=status.HTTP\_400\_BAD\_REQUEST)

…

Код был значительно улучшен. Он стал немного более кратким, и теперь выглядит очень похожим на тот, если бы мы работали с обычными формами. Мы также используем именованные коды состояния, что делает значения ответов более очевидными.

Модернизируем также функцию snippet\_detail() (Листинг 3.2).

**Листинг 3.2** Файл tutorial/snippets/views.py

…

@api\_view([**'GET'**, **'PUT'**, **'DELETE'**])

**def** snippet\_detail(request, pk):

*"""*

*Retrieve, update or delete a code snippet.*

*"""*

**try**:

snippet = Snippet.objects.get(pk=pk)

**except** Snippet.DoesNotExist:

**return** Response(status=status.HTTP\_404\_NOT\_FOUND)

**if** request.method == **'GET'**:

serializer = SnippetSerializer(snippet)

**return** Response(serializer.data)

**elif** request.method == **'PUT'**:

serializer = SnippetSerializer(snippet, data=request.data)

**if** serializer.is\_valid():

serializer.save()

**return** Response(serializer.data)

**return** Response(serializer.errors, status=status.HTTP\_400\_BAD\_REQUEST)

**elif** request.method == **'DELETE'**:

snippet.delete()

**return** Response(status=status.HTTP\_204\_NO\_CONTENT)

Обратите внимание, что мы больше не привязываем наши запросы или ответы к определенному типу контента. request.data может обрабатывать входящие запросы json, но также может обрабатывать другие форматы. Точно так же мы возвращаем объекты ответа с данными, но позволяем фреймворку REST отображать ответ в правильный для нас тип контента.

Чтобы воспользоваться тем фактом, что наши ответы больше не привязаны к одному типу контента, добавим поддержку суффиксов формата в наши маршруты API. Использование суффиксов формата дает нам URL-адреса, которые явно относятся к заданному формату, и означает, что наш API сможет обрабатывать такие URL-адреса, как http://example.com/api/items/4.json.

Для начала добавим параметр формата для каждой функции:

**def** snippet\_list(request, format=**None**):

и

**def** snippet\_detail(request, pk, format=**None**):

Далее обновим файл с маршрутами приложения (Листинг 3.3).

**Листинг 3.3** Файл tutorial/snippets/urls.py

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

**from** rest\_framework.urlpatterns **import** format\_suffix\_patterns

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

urlpatterns = [

path(**'snippets/'**, views.snippet\_list),

path(**'snippets/<int:pk>/'**, views.snippet\_detail),

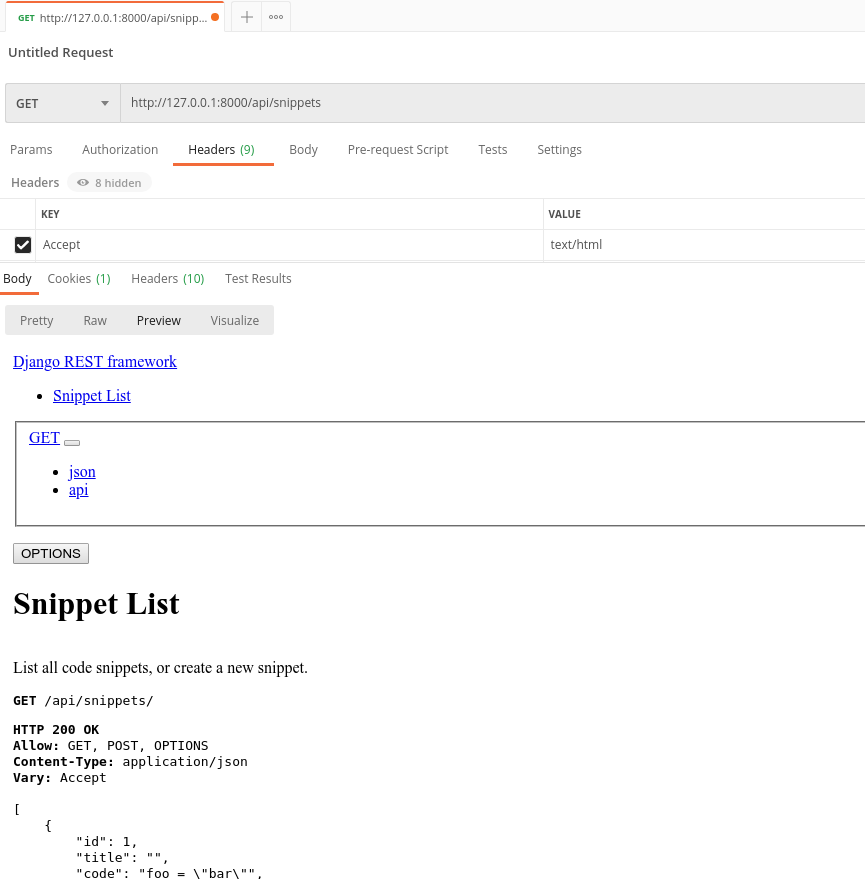
]

urlpatterns = format\_suffix\_patterns(urlpatterns)

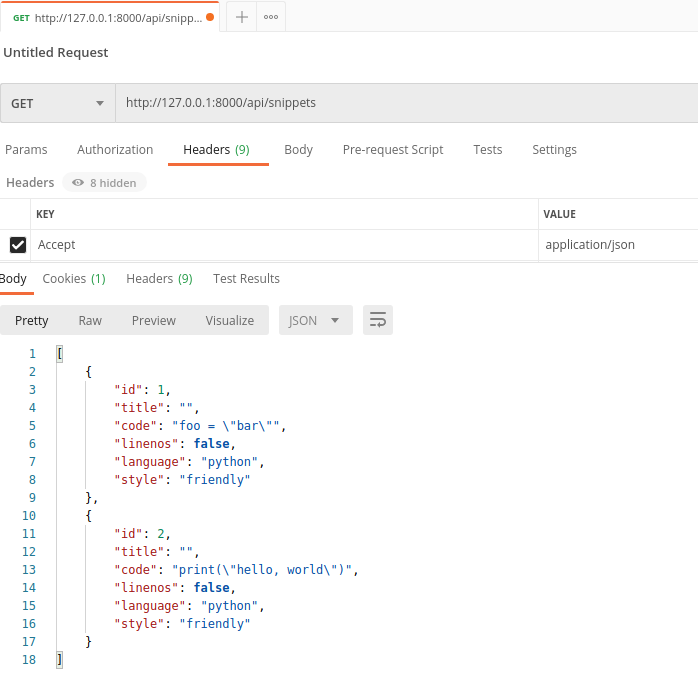
Чтобы проверить разный формат ответа, отправим запросы с различным заголовком Accept:

*http http://127.0.0.1:8000/snippets/ Accept:application/json # Вернет JSON*

*http http://127.0.0.1:8000/snippets/ Accept:text/html # Вернет HTML*



**Рис. 3.1.** Возврат данных в формате html



**Рис. 3.2.** Возврат данных в формате json

# 

# 4. Представления на основе классов

## Рефакторинг представлений

REST framework позволяет возможность писать представления на основе классов. С их использованием код становится более структурированным и переиспользуемым.

Заменим функции snippet\_list() и snippet\_detail() на соответствующие классы (Листинг 4.1).

**Листинг 4.1** Файл tutorial/snippets/views.py

**from** django.http **import** Http404

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

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

**from** rest\_framework **import** status

**from** .models **import** Snippet

**from** .serializers **import** SnippetSerializer

**class** SnippetList(APIView):

*"""*

*List all snippets, or create a new snippet.*

*"""*

**def** get(self, request, format=**None**):

snippets = Snippet.objects.all()

serializer = SnippetSerializer(snippets, many=**True**)

**return** Response(serializer.data)

**def** post(self, request, format=**None**):

serializer = SnippetSerializer(data=request.data)

**if** serializer.is\_valid():

serializer.save()

**return** Response(serializer.data, status=status.HTTP\_201\_CREATED)

**return** Response(serializer.errors, status=status.HTTP\_400\_BAD\_REQUEST)

**class** SnippetDetail(APIView):

*"""*

*Retrieve, update or delete a snippet instance.*

*"""*

**def** get\_object(self, pk):

**try**:

**return** Snippet.objects.get(pk=pk)

**except** Snippet.DoesNotExist:

**raise** Http404

**def** get(self, request, pk, format=**None**):

snippet = self.get\_object(pk)

serializer = SnippetSerializer(snippet)

**return** Response(serializer.data)

**def** put(self, request, pk, format=**None**):

snippet = self.get\_object(pk)

serializer = SnippetSerializer(snippet, data=request.data)

**if** serializer.is\_valid():

serializer.save()

**return** Response(serializer.data)

**return** Response(serializer.errors, status=status.HTTP\_400\_BAD\_REQUEST)

**def** delete(self, request, pk, format=**None**):

snippet = self.get\_object(pk)

snippet.delete()

**return** Response(status=status.HTTP\_204\_NO\_CONTENT)

По факту код не слишком изменился. Однако, запросы с различными методами стали гораздо лучше разделены.

Модернизируем маршруты приложения (Листинг 4.2).

**Листинг 4.2** Файл tutorial/snippets/urls.py

...

urlpatterns = [

path(**'snippets/'**, views.SnippetList.as\_view()),

path(**'snippets/<int:pk>/'**, views.SnippetDetail.as\_view()),

]

...

## Использование миксинов

Одним из больших преимуществ использования представлений на основе классов является то, что оно позволяет нам легко составлять повторно используемые фрагменты поведения.

Операции create/retrieve/update/delete, которые мы использовали до сих пор, будут очень похожи для любых созданных нами представлений API на основе модели. Эти части общего поведения реализованы в классах миксинов REST framework.

Модернизируем классы SnippetList и SnippetDetail с использованием миксинов (Листинг 4.3).

**Листинг 4.3** Файл tutorial/snippets/views.py

**from** rest\_framework **import** mixins

**from** rest\_framework **import** generics

**from** .models **import** Snippet

**from** .serializers **import** SnippetSerializer

**class** SnippetList(mixins.ListModelMixin,

mixins.CreateModelMixin,

generics.GenericAPIView):

queryset = Snippet.objects.all()

serializer\_class = SnippetSerializer

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

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

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

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

**class** SnippetDetail(mixins.RetrieveModelMixin,

mixins.UpdateModelMixin,

mixins.DestroyModelMixin,

generics.GenericAPIView):

queryset = Snippet.objects.all()

serializer\_class = SnippetSerializer

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

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

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

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

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

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

Мы разработали наше представление SnippetList с помощью GenericAPIView и добавляем ListModelMixin и CreateModelMixin. Базовый класс обеспечивает базовую функциональность, а классы миксинов предоставляют действия .list() и .create().

Для представления SnippetDetail все очень похоже. Мы снова используем класс GenericAPIView для обеспечения основных функций и добавляем миксины для обеспечения действий .retrieve(), .update() и .destroy().

## Использование общего класса

Используя классы миксинов, мы переписали представления, чтобы использовать немного меньше кода, чем раньше, но мы можно пойти еще дальше. Фреймворк предоставляет набор классов с миксинами, которые мы можем использовать, чтобы еще больше урезать наш модуль views.py (Листинг 4.4).

**Листинг 4.4** Файл tutorial/snippets/views.py

**from** rest\_framework **import** generics

**from** .models **import** Snippet

**from** .serializers **import** SnippetSerializer

**class** SnippetList(generics.ListCreateAPIView):

queryset = Snippet.objects.all()

serializer\_class = SnippetSerializer

**class** SnippetDetail(generics.RetrieveUpdateDestroyAPIView):

queryset = Snippet.objects.all()

serializer\_class = SnippetSerializer

Теперь код выглядит максимально лаконично, в соответствии с духом Django.

# 5. Аутентификация и авторизация

В настоящее время в нашем API нет ограничений на то, кто может редактировать или удалять фрагменты кода. Нам необходимо организовать следующую логику:

1. Фрагменты кода всегда связаны с создателем;
2. Только аутентифицированные пользователи могут создавать фрагменты;
3. Только создатель фрагмента может обновить или удалить его;
4. Запросы, не прошедшие проверку подлинности, должны иметь полный доступ только для чтения;

## Обновление модели

Внесем несколько изменений в наш класс модели Snippet. Сначала добавим пару полей. Одно из этих полей будет использоваться для представления пользователя, создавшего фрагмент кода. Другое поле будет использоваться для хранения HTML-представления фрагмента кода (Листинг 5.1).

**Листинг 5.1** Файл tutorial/snippets/models.py

...

owner = models.ForeignKey(**'auth.User'**, related\_name=**'snippets'**, on\_delete=models.CASCADE)

highlighted = models.TextField()

...

Также нужно сделать так, чтобы при сохранении модели заполнялось поле highlighted с помощью библиотеки выделения кода pygments (Листинг 5.2).

**Листинг 5.2** Файл tutorial/snippets/models.py

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

**from** pygments **import** highlight

**from** pygments.formatters.html **import** HtmlFormatter

**from** pygments.lexers **import** get\_all\_lexers, get\_lexer\_by\_name

**from** pygments.styles **import** get\_all\_styles

LEXERS = [item **for** item **in** get\_all\_lexers() **if** item[1]]

LANGUAGE\_CHOICES = sorted([(item[1][0], item[0]) **for** item **in** LEXERS])

STYLE\_CHOICES = sorted([(item, item) **for** item **in** get\_all\_styles()])

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

created = models.DateTimeField(auto\_now\_add=**True**)

title = models.CharField(max\_length=100, blank=**True**, default=**''**)

code = models.TextField()

linenos = models.BooleanField(default=**False**)

language = models.CharField(choices=LANGUAGE\_CHOICES, default=**'python'**, max\_length=100)

style = models.CharField(choices=STYLE\_CHOICES, default=**'friendly'**, max\_length=100)

owner = models.ForeignKey(**'auth.User'**, related\_name=**'snippets'**, on\_delete=models.CASCADE)

highlighted = models.TextField()

**def** save(self, \*args, \*\*kwargs):

*"""*

*Use the `pygments` library to create a highlighted HTML*

*representation of the code snippet.*

*"""*

lexer = get\_lexer\_by\_name(self.language)

linenos = **'table' if** self.linenos **else False**

options = {**'title'**: self.title} **if** self.title **else** {}

formatter = HtmlFormatter(style=self.style, linenos=linenos,

full=**True**, \*\*options)

self.highlighted = highlight(self.code, lexer, formatter)

super(Snippet, self).save(\*args, \*\*kwargs)

**class** Meta:

ordering = [**'created'**]

Теперь нужно обновить базу данных. Обычно для этого используют миграции, но мы просто удалим старую базу и создадим новую:

*rm -f db.sqlite3*

*rm -r snippets/migrations*

*python manage.py makemigrations snippets*

*python manage.py migrate*

Также нужно заново создать суперпользователя:

*python manage.py createsuperuser*

## Добавление представлений для пользователя

В файл serializers.py добавим код для отображения пользователей (Листинг 5.3).

**Листинг 5.3** Файл tutorial/snippets/serializers.py

**from** django.contrib.auth.models **import** User

**...**

**class** UserSerializer(serializers.ModelSerializer):

snippets = serializers.PrimaryKeyRelatedField(many=**True**, queryset=Snippet.objects.all())

**class** Meta:

model = User

fields = [**'id'**, **'username'**, **'snippets'**]

Поле snippets будет содержать все фрагменты кода конкретного пользователя.

В файл views.py добавим классы представлений для пользователей (Листинг 5.4).

**Листинг 5.4** Файл tutorial/snippets/views.py

**from** django.contrib.auth.models **import** User

**from** rest\_framework **import** generics

**from** .models **import** Snippet

**from** .serializers **import** SnippetSerializer, UserSerializer

...

**class** UserList(generics.ListAPIView):

queryset = User.objects.all()

serializer\_class = UserSerializer

**class** UserDetail(generics.RetrieveAPIView):

queryset = User.objects.all()

serializer\_class = UserSerializer

Включим в маршруты необходимые (Листинг 5.5.).

**Листинг 5.5** Файл tutorial/snippets/urls.py

**...**

urlpatterns = [

path(**'snippets/'**, views.SnippetList.as\_view()),

path(**'snippets/<int:pk>/'**, views.SnippetDetail.as\_view()),

path(**'users/'**, views.UserList.as\_view()),

path(**'users/<int:pk>/'**, views.UserDetail.as\_view()),

]

...

## Ассоциация сниппетов с пользователями

Прямо сейчас, если бы мы создали фрагмент кода, не было бы возможности связать пользователя, создавшего фрагмент, с экземпляром фрагмента. Пользователь не отправляется как часть сериализованного представления, а является свойством входящего запроса.

Чтобы после создания нового сниппета он привязался к текущему пользователю переопределим метод .perform\_create() (Листинг 5.6).

**Листинг 5.6** Файл tutorial/snippets/views.py

**...**

**class** SnippetList(generics.ListCreateAPIView):

queryset = Snippet.objects.all()

serializer\_class = SnippetSerializer

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

serializer.save(owner=self.request.user)

...

Теперь методу create() нашего сериализатора будет передано дополнительное поле owner вместе с данными авторизированного пользователя.

## Обновление сериализатора

Теперь, когда сниппеты связаны с пользователем, который их создал, обновим SnippetSerializer (Листинг 5.7).

**Листинг 5.7** Файл tutorial/snippets/serializers.py

**...**

**class** SnippetSerializer(serializers.ModelSerializer):

owner = serializers.ReadOnlyField(source=**'owner.username'**)

**class** Meta:

model = Snippet

fields = [**'id'**, **'title'**, **'code'**, **'linenos'**, **'language'**, **'style'**, **'owner'**]

...

Поле, которое мы добавили, является нетипизированным классом ReadOnlyField. В отличие от типизированных полей, таких как CharField, BooleanField и т. д. ReadOnlyField всегда доступно только для чтения и будет использоваться для сериализованных представлений, но не будет используется для обновления экземпляров модели при десериализации. Мы также могли бы использовать здесь CharField (read\_only = True).

## Добавление необходимых разрешений

Теперь, когда фрагменты кода связаны с пользователями, необходимо чтобы только прошедшие аутентификацию пользователи могли создавать, обновлять и удалять фрагменты кода.

REST framework включает в себя ряд классов разрешений, которые можно использовать для ограничения доступа. В данном случае нам необходим класс IsAuthenticatedOrReadOnly, который гарантирует, что аутентифицированные пользователь получат доступ для чтения и записи, а остальные - доступ только для чтения.

Импортируем необходимый класс:

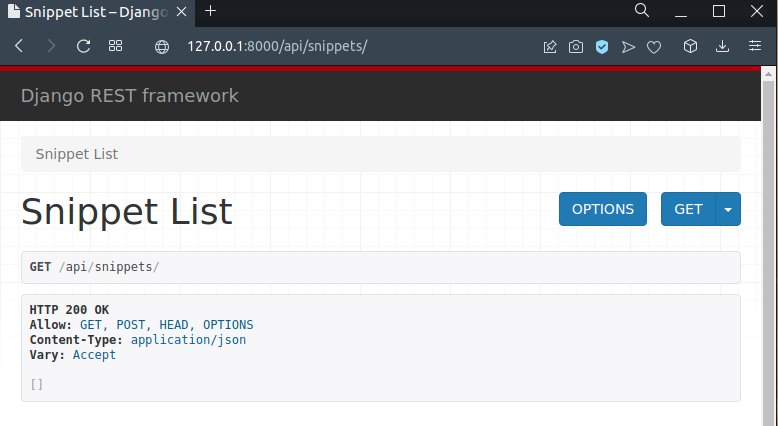
**from** rest\_framework **import** permissions

Добавим свойство permission\_classes в классы SnippetList и SnippetDetail:

permission\_classes = [permissions.IsAuthenticatedOrReadOnly]

## Добавление возможности аутентификации

Если сейчас в браузере открыть страницу с отображением сниппетов, то увидим, что возможности авторизоваться нет (Рис. 5.1).



**Рис. 5.1.** Отображение сниппетов в браузере

Добавим эту возможность, отредактировав маршруты (Листинг 5.8).

**Листинг 5.8** Файл tutorial/snippets/urls.py

**from** django.urls **import** path, include

**from** rest\_framework.urlpatterns **import** format\_suffix\_patterns

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

urlpatterns = [

path(**'snippets/'**, views.SnippetList.as\_view()),

path(**'snippets/<int:pk>/'**, views.SnippetDetail.as\_view()),

path(**'users/'**, views.UserList.as\_view()),

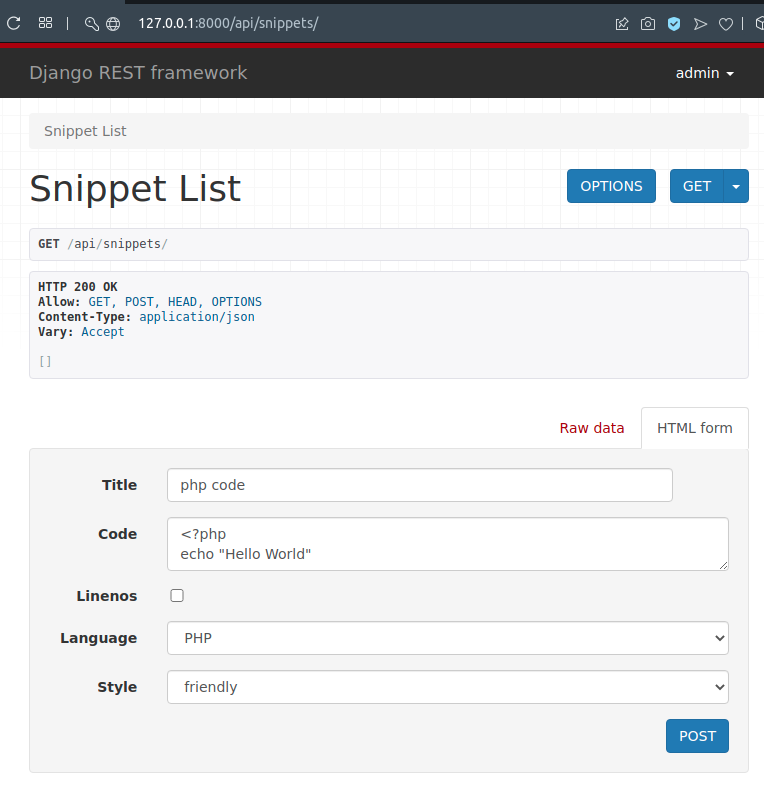
path(**'users/<int:pk>/'**, views.UserDetail.as\_view()),

path(**'auth/'**, include(**'rest\_framework.urls'**)),

]

urlpatterns = format\_suffix\_patterns(urlpatterns)

Теперь в браузере появилась возможность аутентификации, и после входа появилась возможность добавлять новые сниппеты (Рис 5.2).



**Рис. 5.2.** Отображение сниппетов в браузере после аутентификации

## Добавление ограничений на уровне объекта

В разрабатываемом API возможность удалить и обновить фрагмент кода доступна только автору, а чтение для всех. Для организации такой логики создадим кастомное ограничение. Добавим файл permissions.py и разместим там класс IsOwnerOrReadOnly (Листинг 5.9).

**Листинг 5.9** Файл tutorial/snippets/permissions.py

**from** rest\_framework **import** permissions

**class** IsOwnerOrReadOnly(permissions.BasePermission):

*"""*

*Custom permission to only allow owners of an object to edit it.*

*"""*

**def** has\_object\_permission(self, request, view, obj):

*# Read permissions are allowed to any request,*

*# so we'll always allow GET, HEAD or OPTIONS requests.*

**if** request.method **in** permissions.SAFE\_METHODS:

**return True**

*# Write permissions are only allowed to the owner of the snippet.*

**return** obj.owner == request.user

Теперь можно добавить это разрешение в класс SnippetDetail:

permission\_classes = [permissions.IsAuthenticatedOrReadOnly, IsOwnerOrReadOnly]

Естественно его сначала необходимо импортировать:

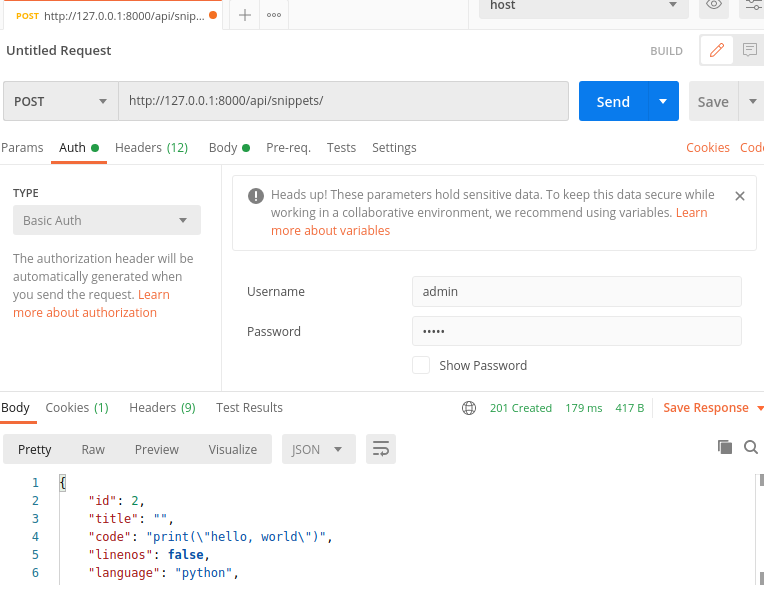
**from** .permissions **import** IsOwnerOrReadOnly

## Аутентификация с помощью API

Поскольку теперь у нас есть набор разрешений для API, нам необходимо аутентифицировать наши запросы к нему, если мы хотим редактировать какие-либо фрагменты. Мы не настроили какие-либо классы аутентификации, поэтому в настоящее время применяются значения по умолчанию, а именно SessionAuthentication и BasicAuthentication.

Когда мы взаимодействуем с API через веб-браузер, мы можем войти в систему, и тогда сеанс браузера обеспечит необходимую аутентификацию для запросов.

Если мы программно взаимодействуем с API, нам необходимо явно предоставлять учетные данные для аутентификации по каждому запросу (Рис 5.3).



**Рис. 5.3.** Базовая аутентификация через Postman

# 

# 6. Связанные объекты

На данный момент связи между объектами в нашем API представлены с помощью первичных ключей. Организуем связь с помощью гиперссылок, что повысит удобство использования и связность приложения.

## Создание единой точки входа

Чтобы создать единую точку входа, мы будем использовать обычное представление на основе функций и декоратор @api\_view, который мы представили ранее (Листинг 6.1).

**Листинг 6.1** Файл tutorial/snippets/views.py

**from** django.contrib.auth.models **import** User

**from** rest\_framework **import** generics, permissions

**from** rest\_framework.decorators **import** api\_view

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

**from** rest\_framework.reverse **import** reverse

**from** .models **import** Snippet

**from** .permissions **import** IsOwnerOrReadOnly

**from** .serializers **import** SnippetSerializer, UserSerializer

@api\_view([**'GET'**])

**def** api\_root(request, format=**None**):

**return** Response({

**'users'**: reverse(**'user-list'**, request=request, format=format),

**'snippets'**: reverse(**'snippet-list'**, request=request, format=format)

})

…

Здесь необходимо заметить две вещи. Во-первых, мы используем функцию reverse(), которая генерирует url-адреса. Во-вторых, для сокращения мы использовали именованные маршруты, которые определим позже в urls.py.

## Создание точки входа для html-представления сниппетов

В отличии от остальных маршрутов, отображение визуализированных сниппетов должно рендерится в html, а не json. Создадим представление для этого (Листинг 6.2).

**Листинг 6.2** Файл tutorial/snippets/views.py

...

**class** SnippetHighlight(generics.GenericAPIView):

queryset = Snippet.objects.all()

renderer\_classes = [renderers.StaticHTMLRenderer]

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

snippet = self.get\_object()

**return** Response(snippet.highlighted)

…

Добавим необходимые маршруты в файл urls.py:

...

urlpatterns = [

path(**'snippets/'**, views.SnippetList.as\_view()),

path(**'snippets/<int:pk>/'**, views.SnippetDetail.as\_view()),

path(**'users/'**, views.UserList.as\_view()),

path(**'users/<int:pk>/'**, views.UserDetail.as\_view()),

path(**'auth/'**, include(**'rest\_framework.urls'**)),

path(**'snippets/<int:pk>/highlight/'**, views.SnippetHighlight.as\_view()),

path(**''**, views.api\_root),

]

…

## Гиперссылки в API

Работа с отношениями между сущностями - один из наиболее сложных аспектов проектирования веб-API. Есть несколько разных способов представления отношений:

* Использование первичных ключей;
* Использование гиперссылок между объектами;
* Использование уникального идентифицирующего служебного поля для связанной сущности;
* Использование строкового представления по умолчанию для связанной сущности;
* Вложение связанной сущности внутри родительского представления;
* Другое нестандартное представление.

Фреймворк поддерживает все эти виды связей. Для примера реализуем вариант с гиперссылками. Для этого необходимо чтобы классы сериализаторов наследовались от класса HyperlinkedModelSerializer. Этот класс подобен ModelSerializer, но несколько различается:

* По умолчанию он не включает поле id;
* Он включает поле URL с использованием HyperlinkedIdentityField;
* Отношения используют HyperlinkedRelatedField вместо PrimaryKeyRelatedField.

Модернизируем наши сериализаторы (Листинг 6.3).

**Листинг 6.3** Файл tutorial/snippets/serializers.py

**from** django.contrib.auth.models **import** User

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

**from** .models **import** Snippet

**class** SnippetSerializer(serializers.HyperlinkedModelSerializer):

owner = serializers.ReadOnlyField(source=**'owner.username'**)

highlight = serializers.HyperlinkedIdentityField(view\_name=**'snippet-highlight'**, format=**'html'**)

**class** Meta:

model = Snippet

fields = [**'url'**, **'id'**, **'highlight'**, **'owner'**,

**'title'**, **'code'**, **'linenos'**, **'language'**, **'style'**]

**class** UserSerializer(serializers.HyperlinkedModelSerializer):

snippets = serializers.HyperlinkedRelatedField(many=**True**, view\_name=**'snippet-detail'**, read\_only=**True**)

**class** Meta:

model = User

fields = [**'url'**, **'id'**, **'username'**, **'snippets'**]

## Добавление имен для маршрутов

Для того, чтобы можно было использовать параметры view\_name в сериализаторах и функцию reverse(), необходимо для определенных маршрутов, добавить названия (Листинг 6.4).

**Листинг 6.4** Файл tutorial/snippets/urls.py

**from** django.urls **import** path, include

**from** rest\_framework.urlpatterns **import** format\_suffix\_patterns

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

urlpatterns = [

path(**'snippets/'**, views.SnippetList.as\_view(), name=**'snippet-list'**),

path(**'snippets/<int:pk>/'**, views.SnippetDetail.as\_view(), name=**'snippet-detail'**),

path(**'users/'**, views.UserList.as\_view(), name=**'user-list'**),

path(**'users/<int:pk>/'**, views.UserDetail.as\_view(), name=**'user-detail'**),

path(**'auth/'**, include(**'rest\_framework.urls'**)),

path(**'snippets/<int:pk>/highlight/'**, views.SnippetHighlight.as\_view(), name=**'snippet-highlight'**),

path(**''**, views.api\_root),

]

urlpatterns = format\_suffix\_patterns(urlpatterns)

Добавим пагинацию для выводимых значений. Для этого нужно добавить соответствующий параметр в файл settings.py:

...

REST\_FRAMEWORK = {

**'DEFAULT\_PAGINATION\_CLASS'**: **'rest\_framework.pagination.PageNumberPagination'**,

**'PAGE\_SIZE'**: 10

}

Теперь если запустить сервер и перейти по адресу <http://127.0.0.1:8000/api/> можно убедиться в работоспособности API.

# 7. Классы ViewSets и Routes

REST framework предоставляет класс ViewSets для упрощения работы с маршрутами. Классы ViewSet почти то же самое, что классы View, за исключением того, что они предоставляют такие операции, как извлечение или обновление, а не обработчики методов, такие как get или put.

## Рефакторинг представлений

Модернизируем классы UserList и UserDetail. Вместо них добавим класс на основе ViewSets (Листинг 7.1).

**Листинг 7.1** Файл tutorial/snippets/views.py

...

**class** UserViewSet(viewsets.ReadOnlyModelViewSet):

*"""*

*This viewset automatically provides `list` and `retrieve` actions.*

*"""*

queryset = User.objects.all()

serializer\_class = UserSerializer

...

Здесь мы использовали класс ReadOnlyModelViewSet для автоматического предоставления операций по умолчанию «только для чтения». Мы по-прежнему устанавливаем атрибуты queryset и serializer\_class точно так же, как и при использовании обычных представлений, но нам больше не нужно предоставлять одну и ту же информацию двум отдельным классам.

Далее заменим классы представления SnippetList, SnippetDetail и SnippetHighlight. Мы можем удалить три представления и снова заменить их одним классом (Листинг 7.2).

**Листинг 7.2** Файл tutorial/snippets/views.py

**...**

**class** SnippetViewSet(viewsets.ModelViewSet):

*"""*

*This viewset automatically provides `list`, `create`, `retrieve`,*

*`update` and `destroy` actions.*

*Additionally we also provide an extra `highlight` action.*

*"""*

queryset = Snippet.objects.all()

serializer\_class = SnippetSerializer

permission\_classes = [permissions.IsAuthenticatedOrReadOnly,

IsOwnerOrReadOnly]

@action(detail=**True**, renderer\_classes=[renderers.StaticHTMLRenderer])

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

snippet = self.get\_object()

**return** Response(snippet.highlighted)

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

serializer.save(owner=self.request.user)

…

На этот раз мы использовали класс ModelViewSet, чтобы получить полный набор операций чтения и записи по умолчанию.

Обратите внимание, что мы также использовали декоратор @action для создания настраиваемого действия под названием highlight. Пользовательские действия, использующие декоратор @action, по умолчанию будут отвечать на запросы GET. Мы можем использовать аргумент для указания методов, если нам нужно действие, отвечающее на запросы POST.

URL-адреса для настраиваемых действий по умолчанию зависят от самого имени метода. Если вы хотите изменить способ создания url, вы можете включить url\_path в качестве аргумента ключевого слова декоратора.

## Явная привязка ViewSets к URL-адресам

Методы обработчика привязываются к действиям только тогда, когда мы определяем. Чтобы увидеть, что происходит под капотом, давайте сначала явно создадим набор представлений из наших ViewSets (Листинг 7.3.)

**Листинг 7.3** Файл tutorial/snippets/urls.py

**from** django.urls **import** path, include

**from** rest\_framework **import** renderers

**from** rest\_framework.urlpatterns **import** format\_suffix\_patterns

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

**from** .views **import** SnippetViewSet, UserViewSet

snippet\_list = SnippetViewSet.as\_view({

**'get'**: **'list'**,

**'post'**: **'create'**

})

snippet\_detail = SnippetViewSet.as\_view({

**'get'**: **'retrieve'**,

**'put'**: **'update'**,

**'patch'**: **'partial\_update'**,

**'delete'**: **'destroy'**

})

snippet\_highlight = SnippetViewSet.as\_view({

**'get'**: **'highlight'**

}, renderer\_classes=[renderers.StaticHTMLRenderer])

user\_list = UserViewSet.as\_view({

**'get'**: **'list'**

})

user\_detail = UserViewSet.as\_view({

**'get'**: **'retrieve'**

})

urlpatterns = [

path(**'snippets/'**, snippet\_list, name=**'snippet-list'**),

path(**'snippets/<int:pk>/'**, snippet\_detail, name=**'snippet-detail'**),

path(**'users/'**, user\_list, name=**'user-list'**),

path(**'users/<int:pk>/'**, user\_detail, name=**'user-detail'**),

path(**'auth/'**, include(**'rest\_framework.urls'**)),

path(**'snippets/<int:pk>/highlight/'**, snippet\_highlight, name=**'snippet-highlight'**),

path(**''**, views.api\_root),

]

urlpatterns = format\_suffix\_patterns(urlpatterns)

Обратите внимание, как мы создаем несколько представлений из каждого класса ViewSet, привязывая методы http к требуемому действию для каждого представления.

## Использование Routers

Поскольку мы используем классы ViewSet, а не классы View, нам фактически не нужно самостоятельно добавлять URL-адрес. Соглашения о подключении маршрутов к представлениям и URL-адресам могут обрабатываться автоматически с помощью класса Router. Все, что нам нужно сделать, это зарегистрировать соответствующие наборы представлений на маршрутизаторе, а все остальное сделать он сам (Листинг 7.4).

**Листинг 7.4** Файл tutorial/snippets/urls.py

**from** django.urls **import** path, include

**from** rest\_framework.routers **import** DefaultRouter

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

*# Create a router and register our viewsets with it.*

router = DefaultRouter()

router.register(**r'snippets'**, views.SnippetViewSet)

router.register(**r'users'**, views.UserViewSet)

*# The API URLs are now determined automatically by the router.*

urlpatterns = [

path(**''**, include(router.urls)),

]

Регистрация наборов представлений в маршрутизаторе аналогична предоставлению шаблона URL. Мы включаем два аргумента - префикс URL-адреса для представлений и сам набор представлений.

Используемый нами класс DefaultRouter также автоматически создает для нас корневое представление API, поэтому теперь мы можем удалить метод api\_root из нашего модуля представлений.

# 8. Использованные источники

1. Официальная документация Django REST framework [Электронный ресурс] – Режим доступа: <https://docs.python.org/3/library/venv.html> Дата обращения: 03.08.2021;
2. venv — Creation of virtual environments [Электронный ресурс] – Режим доступа: <https://www.django-rest-framework.org/> Дата обращения: 03.08.2021;