Django Rest Framework Tutorial

Setting up a new environment

Before we do anything else we'll create a new virtual environment, using virtualenv. This will make sure our package configuration is kept nicely isolated from any other projects we're working on.

```
virtualenv env
source env/bin/activate
```

Now that we're inside a virtualenv environment, we can install our package requirements.

```
pip install django
pip install djangorestframework
pip install pygments  # We'll be using this for the code highlighting
```

Getting started

Okay, we're ready to get coding. To get started, let's create a new project to work with.

```
cd ~
```

```
django-admin.py startproject tutorial
```

```
cd tutorial
```

Once that's done we can create an app that we'll use to create a simple Web API.

```
python manage.py startapp snippets
```

We'll need to add our new `snippets` app and the `rest_framework` app to `INSTALLED_APPS`. Let's edit the`tutorial/settings.py` file:

```
INSTALLED_APPS = (
    ...
    'rest_framework',
    'snippets',
)
```

We also need to wire up the root urlconf, in the `tutorial/urls.py` file, to include our snippet app's URLs.

```
urlpatterns = [
    url(r'^', include('snippets.urls')),
]
```

Creating a model to work with

For the purposes of this tutorial we're going to start by creating a simple `Snippet` model that is used to store code snippets. Go ahead and edit the `snippets/models.py` file.

```python
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',)
```

We'll also need to create an initial migration for our snippet model, and sync the database for the first time.

```
python manage.py makemigrations snippets
```

```
python manage.py migrate
```

```python
from rest_framework import serializers
from snippets.models import Snippet, LANGUAGE_CHOICES, STYLE_CHOICES


class SnippetSerializer(serializers.Serializer):
    pk = 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
```

- Defines the fields that get serialized/deserialized.
- The `create()` and `update()` methods define how fully fledged instances are created or modified when calling `serializer.save()`
- A serializer class is very similar to a Django `Form` class, and includes similar validation flags on the various fields, such as`required`, `max_length` and `default`.

The field flags can also control how the serializer should be displayed in certain circumstances, such as when rendering to HTML. The `{'base_template': 'textarea.html'}` flag above is equivalent to using `widget=widgets.Textarea` on a Django`Form` class. This is particularly useful for controlling how the browsable API should be displayed, as we'll see later in the tutorial.

We can actually also save ourselves some time by using the `ModelSerializer` class, as we'll see later, but for now we'll keep our serializer definition explicit.

Working with Serializers

Before we go any further we'll familiarize ourselves with using our new Serializer class. Let's drop into the Django shell.

```
python manage.py shell
```

Okay, once we've got a few imports out of the way, let's create a couple of code snippets to work with.

```
from snippets.models import Snippet
from snippets.serializers import SnippetSerializer
from rest_framework.renderers import JSONRenderer
from rest_framework.parsers import JSONParser

snippet = Snippet(code='foo = "bar"\n')
snippet.save()

snippet = Snippet(code='print "hello, world"\n')
snippet.save()
```

We've now got a few snippet instances to play with. Let's take a look at serializing one of those instances.

```
serializer = SnippetSerializer(snippet)
serializer.data
# {'pk': 2, 'title': u'', 'code': u'print "hello, world"\n', 'linenos': False, 'language': u'python', 'style': u'friendly'}
```

At this point we've translated the model instance into Python native datatypes. To finalize the serialization process we render the data into `json`.

```
content = JSONRenderer().render(serializer.data)
content
# '{"pk": 2, "title": "", "code": "print \\"hello, world\\"\\n", "linenos": false, "language": "python", "style": "friendly"}'
```

Deserialization is similar. First we parse a stream into Python native datatypes...

```
from django.utils.six import BytesIO

stream = BytesIO(content)
data = JSONParser().parse(stream)
```

...then we restore those native datatypes into to a fully populated object instance.

```
serializer = SnippetSerializer(data=data)
serializer.is_valid()
# True
serializer.validated_data
# OrderedDict([('title', ''), ('code', 'print "hello, world"\n'), ('linenos', False), ('language', 'python'),
('style', 'friendly')])
serializer.save()
# <Snippet: Snippet object>
```

Notice how similar the API is to working with forms. The similarity should become even more apparent when we start writing views that use our serializer.

We can also serialize querysets instead of model instances. To do so we simply add a `many=True` flag to the serializer arguments.

```
serializer = SnippetSerializer(Snippet.objects.all(), many=True)
serializer.data
# [OrderedDict([('pk', 1), ('title', u''), ('code', u'foo = "bar"\n'), ('linenos', False), ('language', 'python'),
('style', 'friendly')]), OrderedDict([('pk', 2), ('title', u''), ('code', u'print "hello, world"\n'), ('linenos',
False), ('language', 'python'), ('style', 'friendly')]), OrderedDict([('pk', 3), ('title', u''), ('code', u'print
"hello, world"'), ('linenos', False), ('language', 'python'), ('style', 'friendly')])]
```

Using ModelSerializers

Our `SnippetSerializer` class is replicating a lot of information that's also contained in the `Snippet` model. It would be nice if we could keep our code a bit more concise.

In the same way that Django provides both `Form` classes and `ModelForm` classes, REST framework includes both `Serializer`classes, and `ModelSerializer` classes.

Let's look at refactoring our serializer using the `ModelSerializer` class. Open the file `snippets/serializers.py` again, and replace the `SnippetSerializer` class with the following.

```python
from rest_framework import serializers
from snippets.models import Snippet, LANGUAGE_CHOICES, STYLE_CHOICES


class SnippetSerializer(serializers.Serializer):
    pk = 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
```

```python
from rest_framework import serializers
from snippets.models import Snippet

class SnippetSerializer(serializers.ModelSerializer):
    class Meta:
        model = Snippet
        fields = ('id', 'title', 'code', 'linenos', 'language', 'style')
```

Writing regular Django views using our Serializer

Let's see how we can write some API views using our new Serializer class. For the moment we won't use any of REST framework's other features, we'll just write the views as regular Django views.

We'll start off by creating a subclass of HttpResponse that we can use to render any data we return into `json`.

Edit the `snippets/views.py` file, and add the following.

```
from django.http import HttpResponse
from django.views.decorators.csrf import csrf_exempt
from rest_framework.renderers import JSONRenderer
from rest_framework.parsers import JSONParser
from snippets.models import Snippet
from snippets.serializers import SnippetSerializer

class JSONResponse(HttpResponse):
    """
    An HttpResponse that renders its content into JSON.
    """
    def __init__(self, data, **kwargs):
        content = JSONRenderer().render(data)
        kwargs['content_type'] = 'application/json'
        super(JSONResponse, self).__init__(content, **kwargs)
```

```python
@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)

    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)
```

```python
@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)
```

```
from django.conf.urls import url
from snippets import views

urlpatterns = [
    url(r'^snippets/$', views.snippet_list),
    url(r'^snippets/(?P<pk>[0-9]+)/$', views.snippet_detail),
]
```

Testing our first attempt at a Web API

Request objects

REST framework introduces a `Request` object that extends the regular `HttpRequest`, and provides more flexible request parsing. The core functionality of the `Request` object is the `request.data` attribute, which is similar to `request.POST`, but more useful for working with Web APIs.

```
request.POST  # Only handles form data.  Only works for 'POST' method.
request.data  # Handles arbitrary data.  Works for 'POST', 'PUT' and 'PATCH' methods.
```

Response objects

REST framework also introduces a `Response` object, which is a type of `TemplateResponse` that takes unrendered content and uses content negotiation to determine the correct content type to return to the client.

```
return Response(data)  # Renders to content type as requested by the client.
```

Status codes

Using numeric HTTP status codes in your views doesn't always make for obvious reading, and it's easy to not notice if you get an error code wrong. REST framework provides more explicit identifiers for each status code, such as `HTTP_400_BAD_REQUEST` in the `status` module. It's a good idea to use these throughout rather than using numeric identifiers.

Wrapping API views

REST framework provides two wrappers you can use to write API views.

1. The `@api_view` decorator for working with function based views.
2. The `APIView` class for working with class based views.

These wrappers provide a few bits of functionality such as making sure you receive `Request` instances in your view, and adding context to `Response` objects so that content negotiation can be performed.

The wrappers also provide behaviour such as returning `405 Method Not Allowed` responses when appropriate, and handling any `ParseError` exception that occurs when accessing `request.data` with malformed input.

Pulling it all together

Okay, let's go ahead and start using these new components to write a few views.

We don't need our `JSONResponse` class in `views.py` anymore, so go ahead and delete that. Once that's done we can start refactoring our views slightly.

```python
from rest_framework import status
from rest_framework.decorators import api_view
from rest_framework.response import Response
from snippets.models import Snippet
from snippets.serializers import SnippetSerializer


@api_view(['GET', 'POST'])
def snippet_list(request):
    """
    List all 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)
```

```python
@api_view(['GET', 'PUT', 'DELETE'])
def snippet_detail(request, pk):
    """
    Retrieve, update or delete a snippet instance.
    """
    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)
```

DIFF

```python
from django.http import HttpResponse
from django.views.decorators.csrf import csrf_exempt
from rest_framework.renderers import JSONRenderer
from rest_framework.parsers import JSONParser
from snippets.models import Snippet
from snippets.serializers import SnippetSerializer

class JSONResponse(HttpResponse):
    """
    An HttpResponse that renders its content into JSON.
    """
    def __init__(self, data, **kwargs):
        content = JSONRenderer().render(data)
        kwargs['content_type'] = 'application/json'
        super(JSONResponse, self).__init__(content, **kwargs)


@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)

    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)
```

```python
from rest_framework import status
from rest_framework.decorators import api_view
from rest_framework.response import Response
from snippets.models import Snippet
from snippets.serializers import SnippetSerializer


@api_view(['GET', 'POST'])
def snippet_list(request):
    """
    List all 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)
```

```python
@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)
```

```python
@api_view(['GET', 'PUT', 'DELETE'])
def snippet_detail(request, pk):
    """
    Retrieve, update or delete a snippet instance.
    """
    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)
```

Adding optional format suffixes to our URLs

To take advantage of the fact that our responses are no longer hardwired to a single content type let's add support for format suffixes to our API endpoints. Using format suffixes gives us URLs that explicitly refer to a given format, and means our API will be able to handle URLs such as http://example.com/api/items/4/.json.

Start by adding a `format` keyword argument to both of the views, like so.

```
def snippet_list(request, format=None):
```

and

```
def snippet_detail(request, pk, format=None):
```

Now update the `urls.py` file slightly, to append a set of `format_suffix_patterns` in addition to the existing URLs.

```
from django.conf.urls import url
from rest_framework.urlpatterns import format_suffix_patterns
from snippets import views

urlpatterns = [
    url(r'^snippets/$', views.snippet_list),
    url(r'^snippets/(?P<pk>[0-9]+)$', views.snippet_detail),
]

urlpatterns = format_suffix_patterns(urlpatterns)
```

We can control the format of the response that we get back, either by using the `Accept` header:

```
http http://127.0.0.1:8000/snippets/ Accept:application/json  # Request JSON
http http://127.0.0.1:8000/snippets/ Accept:text/html         # Request HTML
```

Or by appending a format suffix:

```
http http://127.0.0.1:8000/snippets.json  # JSON suffix
http http://127.0.0.1:8000/snippets.api   # Browsable API suffix
```

imilarly, we can control the format of the request that we send, using the `Content-Type` header.

```
# POST using form data
http --form POST http://127.0.0.1:8000/snippets/ code="print 123"
```

```
{
  "id": 3,
  "title": "",
  "code": "print 123",
  "linenos": false,
  "language": "python",
  "style": "friendly"
}
```

```
# POST using JSON
http --json POST http://127.0.0.1:8000/snippets/ code="print 456"
```

```
{
    "id": 4,
    "title": "",
    "code": "print 456",
    "linenos": false,
    "language": "python",
    "style": "friendly"
}
```

Now go and open the API in a web browser, by visiting http://127.0.0.1:8000/snippets/.

Browsability

Because the API chooses the content type of the response based on the client request, it will, by default, return an HTML-formatted representation of the resource when that resource is requested by a web browser. This allows for the API to return a fully web-browsable HTML representation.

Having a web-browsable API is a huge usability win, and makes developing and using your API much easier. It also dramatically lowers the barrier-to-entry for other developers wanting to inspect and work with your API.

See the browsable api topic for more information about the browsable API feature and how to customize it.

Tutorial 3: Class Based Views

We can also write our API views using class based views, rather than function based views. As we'll see this is a powerful pattern that allows us to reuse common functionality, and helps us keep our code DRY.

Rewriting our API using class based views

We'll start by rewriting the root view as a class based view. All this involves is a little bit of refactoring of `views.py`.

Class Based Views

Django's class based views are a welcome departure from the old-style views.

— *Reinout van Rees*

REST framework provides an `APIView` class, which subclasses Django's `View` class.

`APIView` classes are different from regular `View` classes in the following ways:

- Requests passed to the handler methods will be REST framework's `Request` instances, not Django's `HttpRequest`instances.
- Handler methods may return REST framework's `Response`, instead of Django's `HttpResponse`. The view will manage content negotiation and setting the correct renderer on the response.
- Any `APIException` exceptions will be caught and mediated into appropriate responses.
- Incoming requests will be authenticated and appropriate permission and/or throttle checks will be run before dispatching the request to the handler method.

Using the `APIView` class is pretty much the same as using a regular `View` class, as usual, the incoming request is dispatched to an appropriate handler method such as `.get()` or `.post()`. Additionally, a number of attributes may be set on the class that control various aspects of the API policy.

```python
from snippets.models import Snippet
from snippets.serializers import SnippetSerializer
from django.http import Http404
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework import status


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)
```

```python
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)
```

DIFF

```python
from rest_framework import status
from rest_framework.decorators import api_view
from rest_framework.response import Response
from snippets.models import Snippet
from snippets.serializers import SnippetSerializer


@api_view(['GET', 'POST'])
def snippet_list(request):
    """
    List all 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)
```

```python
from snippets.models import Snippet
from snippets.serializers import SnippetSerializer
from django.http import Http404
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework import status


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)
```

```python
@api_view(['GET', 'PUT', 'DELETE'])
def snippet_detail(request, pk):
    """
    Retrieve, update or delete a snippet instance.
    """
    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)
```

```python
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)
```

That's looking good. Again, it's still pretty similar to the function based view right now.

We'll also need to refactor our `urls.py` slightly now we're using class based views.

```
from django.conf.urls import url
from rest_framework.urlpatterns import format_suffix_patterns
from snippets import views

urlpatterns = [
    url(r'^snippets/$', views.SnippetList.as_view()),
    url(r'^snippets/(?P<pk>[0-9]+)/$', views.SnippetDetail.as_view()),
]

urlpatterns = format_suffix_patterns(urlpatterns)
```

Using mixins

One of the big wins of using class based views is that it allows us to easily compose reusable bits of behaviour.

The create/retrieve/update/delete operations that we've been using so far are going to be pretty similar for any model-backed API views we create. Those bits of common behaviour are implemented in REST framework's mixin classes.

Let's take a look at how we can compose the views by using the mixin classes. Here's our `views.py` module again.

```python
from snippets.models import Snippet
from snippets.serializers import SnippetSerializer
from rest_framework import mixins
from rest_framework import generics

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)
```

```python
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)
```

DIFF

```python
from snippets.models import Snippet
from snippets.serializers import SnippetSerializer
from django.http import Http404
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework import status


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)
```

```python
from snippets.models import Snippet
from snippets.serializers import SnippetSerializer
from rest_framework import mixins
from rest_framework import generics

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)
```

```python
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)
```

```python
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)
```

Using generic class based views

Using the mixin classes we've rewritten the views to use slightly less code than before, but we can go one step further. REST framework provides a set of already mixed-in generic views that we can use to trim down our `views.py` module even more.

```python
from snippets.models import Snippet
from snippets.serializers import SnippetSerializer
from rest_framework import generics


class SnippetList(generics.ListCreateAPIView):
    queryset = Snippet.objects.all()
    serializer_class = SnippetSerializer


class SnippetDetail(generics.RetrieveUpdateDestroyAPIView):
    queryset = Snippet.objects.all()
    serializer_class = SnippetSerializer
```

DIFF

```python
from snippets.models import Snippet
from snippets.serializers import SnippetSerializer
from rest_framework import mixins
from rest_framework import generics

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)
```

```python
from snippets.models import Snippet
from snippets.serializers import SnippetSerializer
from rest_framework import generics


class SnippetList(generics.ListCreateAPIView):
    queryset = Snippet.objects.all()
    serializer_class = SnippetSerializer


class SnippetDetail(generics.RetrieveUpdateDestroyAPIView):
    queryset = Snippet.objects.all()
    serializer_class = SnippetSerializer
```