[Tutorial 1: Serialization](http://www.django-rest-framework.org/tutorial/1-serialization/#tutorial-1-serialization)

[Introduction](http://www.django-rest-framework.org/tutorial/1-serialization/#introduction)

This tutorial will cover creating a simple pastebin code highlighting Web API. Along the way it will introduce the various components that make up REST framework, and give you a comprehensive understanding of how everything fits together.

The tutorial is fairly in-depth, so you should probably get a cookie and a cup of your favorite brew before getting started. If you just want a quick overview, you should head over to the [quickstart](http://www.django-rest-framework.org/tutorial/quickstart/) documentation instead.

**Note**: The code for this tutorial is available in the [tomchristie/rest-framework-tutorial](https://github.com/encode/rest-framework-tutorial) repository on GitHub. The completed implementation is also online as a sandbox version for testing, [available here](https://restframework.herokuapp.com/).

[Setting up a new environment](http://www.django-rest-framework.org/tutorial/1-serialization/#setting-up-a-new-environment)

Before we do anything else we'll create a new virtual environment, using [virtualenv](http://www.virtualenv.org/en/latest/index.html). 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

**Note:** To exit the virtualenv environment at any time, just type deactivate. For more information see the [virtualenv documentation](http://www.virtualenv.org/en/latest/index.html).

[Getting started](http://www.django-rest-framework.org/tutorial/1-serialization/#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.pyfile:

INSTALLED\_APPS = (

...

'rest\_framework',

'snippets.apps.SnippetsConfig',

)

Okay, we're ready to roll.

[Creating a model to work with](http://www.django-rest-framework.org/tutorial/1-serialization/#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. Note: Good programming practices include comments. Although you will find them in our repository version of this tutorial code, we have omitted them here to focus on the code itself.

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

[Creating a Serializer class](http://www.django-rest-framework.org/tutorial/1-serialization/#creating-a-serializer-class)

The first thing we need to get started on our Web API is to provide a way of serializing and deserializing the snippet instances into representations such as json. We can do this by declaring serializers that work very similar to Django's forms. Create a file in the snippets directory named serializers.py and add the following.

from rest\_framework import serializers

from snippets.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

The first part of the serializer class 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](http://www.django-rest-framework.org/tutorial/1-serialization/#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

# {'id': 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

# '{"id": 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 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([('id', 1), ('title', u''), ('code', u'foo = "bar"\n'), ('linenos', False), ('language', 'python'), ('style', 'friendly')]), OrderedDict([('id', 2), ('title', u''), ('code', u'print "hello, world"\n'), ('linenos', False), ('language', 'python'), ('style', 'friendly')]), OrderedDict([('id', 3), ('title', u''), ('code', u'print "hello, world"'), ('linenos', False), ('language', 'python'), ('style', 'friendly')])]

[Using ModelSerializers](http://www.django-rest-framework.org/tutorial/1-serialization/#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 Serializerclasses, 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.

class SnippetSerializer(serializers.ModelSerializer):

class Meta:

model = Snippet

fields = ('id', 'title', 'code', 'linenos', 'language', 'style')

One nice property that serializers have is that you can inspect all the fields in a serializer instance, by printing its representation. Open the Django shell with python manage.py shell, then try the following:

from snippets.serializers import SnippetSerializer

serializer = SnippetSerializer()

print(repr(serializer))

# SnippetSerializer():

# id = IntegerField(label='ID', read\_only=True)

# title = CharField(allow\_blank=True, max\_length=100, required=False)

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

# linenos = BooleanField(required=False)

# language = ChoiceField(choices=[('Clipper', 'FoxPro'), ('Cucumber', 'Gherkin'), ('RobotFramework', 'RobotFramework'), ('abap', 'ABAP'), ('ada', 'Ada')...

# style = ChoiceField(choices=[('autumn', 'autumn'), ('borland', 'borland'), ('bw', 'bw'), ('colorful', 'colorful')...

It's important to remember that ModelSerializer classes don't do anything particularly magical, they are simply a shortcut for creating serializer classes:

* An automatically determined set of fields.
* Simple default implementations for the create() and update() methods.

[Writing regular Django views using our Serializer](http://www.django-rest-framework.org/tutorial/1-serialization/#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.

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

from django.http import HttpResponse, JsonResponse

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

The root of our API is going to be a view that supports listing all the existing snippets, or creating a new snippet.

@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)

Note that because we want to be able to POST to this view from clients that won't have a CSRF token we need to mark the view as csrf\_exempt. This isn't something that you'd normally want to do, and REST framework views actually use more sensible behavior than this, but it'll do for our purposes right now.

We'll also need a view which corresponds to an individual snippet, and can be used to retrieve, update or delete the snippet.

@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)

Finally we need to wire these views up. Create the snippets/urls.py file:

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),

]

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

from django.conf.urls import url, include

urlpatterns = [

url(r'^', include('snippets.urls')),

]

It's worth noting that there are a couple of edge cases we're not dealing with properly at the moment. If we send malformed json, or if a request is made with a method that the view doesn't handle, then we'll end up with a 500 "server error" response. Still, this'll do for now.

[Testing our first attempt at a Web API](http://www.django-rest-framework.org/tutorial/1-serialization/#testing-our-first-attempt-at-a-web-api)

Now we can start up a sample server that serves our snippets.

Quit out of the shell...

quit()

...and start up Django's development server.

python manage.py runserver

Validating models...

0 errors found

Django version 1.11, using settings 'tutorial.settings'

Development server is running at http://127.0.0.1:8000/

Quit the server with CONTROL-C.

In another terminal window, we can test the server.

We can test our API using [curl](https://curl.haxx.se/) or [httpie](https://github.com/jakubroztocil/httpie#installation). Httpie is a user friendly http client that's written in Python. Let's install that.

You can install httpie using pip:

pip install httpie

Finally, we can get a list of all of the snippets:

http http://127.0.0.1:8000/snippets/

HTTP/1.1 200 OK

...

[

{

"id": 1,

"title": "",

"code": "foo = \"bar\"\n",

"linenos": false,

"language": "python",

"style": "friendly"

},

{

"id": 2,

"title": "",

"code": "print \"hello, world\"\n",

"linenos": false,

"language": "python",

"style": "friendly"

}

]

Or we can get a particular snippet by referencing its id:

http http://127.0.0.1:8000/snippets/2/

HTTP/1.1 200 OK

...

{

"id": 2,

"title": "",

"code": "print \"hello, world\"\n",

"linenos": false,

"language": "python",

"style": "friendly"

}

Similarly, you can have the same json displayed by visiting these URLs in a web browser.

[Where are we now](http://www.django-rest-framework.org/tutorial/1-serialization/#where-are-we-now)

We're doing okay so far, we've got a serialization API that feels pretty similar to Django's Forms API, and some regular Django views.

Our API views don't do anything particularly special at the moment, beyond serving json responses, and there are some error handling edge cases we'd still like to clean up, but it's a functioning Web API.

We'll see how we can start to improve things i

[Tutorial 2: Requests and Responses](http://www.django-rest-framework.org/tutorial/2-requests-and-responses/#tutorial-2-requests-and-responses)

From this point we're going to really start covering the core of REST framework. Let's introduce a couple of essential building blocks.

[Request objects](http://www.django-rest-framework.org/tutorial/2-requests-and-responses/#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](http://www.django-rest-framework.org/tutorial/2-requests-and-responses/#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](http://www.django-rest-framework.org/tutorial/2-requests-and-responses/#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](http://www.django-rest-framework.org/tutorial/2-requests-and-responses/#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](http://www.django-rest-framework.org/tutorial/2-requests-and-responses/#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 any more, so go ahead and delete that. Once that's done we can start refactoring our views slightly.

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 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)

Our instance view is an improvement over the previous example. It's a little more concise, and the code now feels very similar to if we were working with the Forms API. We're also using named status codes, which makes the response meanings more obvious.

Here is the view for an individual snippet, in the views.py module.

@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)

This should all feel very familiar - it is not a lot different from working with regular Django views.

Notice that we're no longer explicitly tying our requests or responses to a given content type. request.data can handle incoming json requests, but it can also handle other formats. Similarly we're returning response objects with data, but allowing REST framework to render the response into the correct content type for us.

[Adding optional format suffixes to our URLs](http://www.django-rest-framework.org/tutorial/2-requests-and-responses/#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 snippets/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 don't necessarily need to add these extra url patterns in, but it gives us a simple, clean way of referring to a specific format.

[How's it looking?](http://www.django-rest-framework.org/tutorial/2-requests-and-responses/#hows-it-looking)

Go ahead and test the API from the command line, as we did in [tutorial part 1](http://www.django-rest-framework.org/tutorial/1-serialization/). Everything is working pretty similarly, although we've got some nicer error handling if we send invalid requests.

We can get a list of all of the snippets, as before.

http http://127.0.0.1:8000/snippets/

HTTP/1.1 200 OK

...

[

{

"id": 1,

"title": "",

"code": "foo = \"bar\"\n",

"linenos": false,

"language": "python",

"style": "friendly"

},

{

"id": 2,

"title": "",

"code": "print \"hello, world\"\n",

"linenos": false,

"language": "python",

"style": "friendly"

}

]

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

Similarly, 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"

}

If you add a --debug switch to the http requests above, you will be able to see the request type in request headers.

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

[Browsability](http://www.django-rest-framework.org/tutorial/2-requests-and-responses/#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](http://www.django-rest-framework.org/topics/browsable-api/) topic for more information about the browsable API feature and how to customize it.

[Tutorial 3: Class-based Views](http://www.django-rest-framework.org/tutorial/3-class-based-views/#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](https://en.wikipedia.org/wiki/Don't_repeat_yourself).

[Rewriting our API using class-based views](http://www.django-rest-framework.org/tutorial/3-class-based-views/#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.

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)

So far, so good. It looks pretty similar to the previous case, but we've got better separation between the different HTTP methods. We'll also need to update the instance view in views.py.

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 snippets/urls.py slightly now that 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)

Okay, we're done. If you run the development server everything should be working just as before.

[Using mixins](http://www.django-rest-framework.org/tutorial/3-class-based-views/#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.

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)

We'll take a moment to examine exactly what's happening here. We're building our view using GenericAPIView, and adding in ListModelMixin and CreateModelMixin.

The base class provides the core functionality, and the mixin classes provide the .list() and .create() actions. We're then explicitly binding the get and post methods to the appropriate actions. Simple enough stuff so far.

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)

Pretty similar. Again we're using the GenericAPIView class to provide the core functionality, and adding in mixins to provide the .retrieve(), .update() and .destroy() actions.

[Using generic class-based views](http://www.django-rest-framework.org/tutorial/3-class-based-views/#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.

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

Wow, that's pretty concise. We've gotten a huge amount for free, and our code looks like good, clean, idiomatic Django.

[Tutorial 4: Authentication & Permissions](http://www.django-rest-framework.org/tutorial/4-authentication-and-permissions/#tutorial-4-authentication-permissions)

Currently our API doesn't have any restrictions on who can edit or delete code snippets. We'd like to have some more advanced behavior in order to make sure that:

* Code snippets are always associated with a creator.
* Only authenticated users may create snippets.
* Only the creator of a snippet may update or delete it.
* Unauthenticated requests should have full read-only access.

[Adding information to our model](http://www.django-rest-framework.org/tutorial/4-authentication-and-permissions/#adding-information-to-our-model)

We're going to make a couple of changes to our Snippet model class. First, let's add a couple of fields. One of those fields will be used to represent the user who created the code snippet. The other field will be used to store the highlighted HTML representation of the code.

Add the following two fields to the Snippet model in models.py.

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

highlighted = models.TextField()

We'd also need to make sure that when the model is saved, that we populate the highlighted field, using the pygments code highlighting library.

We'll need some extra imports:

from pygments.lexers import get\_lexer\_by\_name

from pygments.formatters.html import HtmlFormatter

from pygments import highlight

And now we can add a .save() method to our model class:

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 = self.linenos and 'table' or False

options = self.title and {'title': self.title} or {}

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

full=True, \*\*options)

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

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

When that's all done we'll need to update our database tables. Normally we'd create a database migration in order to do that, but for the purposes of this tutorial, let's just delete the database and start again.

rm -f db.sqlite3

rm -r snippets/migrations

python manage.py makemigrations snippets

python manage.py migrate

You might also want to create a few different users, to use for testing the API. The quickest way to do this will be with the createsuperuser command.

python manage.py createsuperuser

[Adding endpoints for our User models](http://www.django-rest-framework.org/tutorial/4-authentication-and-permissions/#adding-endpoints-for-our-user-models)

Now that we've got some users to work with, we'd better add representations of those users to our API. Creating a new serializer is easy. In serializers.py add:

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')

Because 'snippets' is a *reverse* relationship on the User model, it will not be included by default when using the ModelSerializer class, so we needed to add an explicit field for it.

We'll also add a couple of views to views.py. We'd like to just use read-only views for the user representations, so we'll use the ListAPIView and RetrieveAPIView generic class-based views.

from django.contrib.auth.models import User

class UserList(generics.ListAPIView):

queryset = User.objects.all()

serializer\_class = UserSerializer

class UserDetail(generics.RetrieveAPIView):

queryset = User.objects.all()

serializer\_class = UserSerializer

Make sure to also import the UserSerializer class

from snippets.serializers import UserSerializer

Finally we need to add those views into the API, by referencing them from the URL conf. Add the following to the patterns in urls.py.

url(r'^users/$', views.UserList.as\_view()),

url(r'^users/(?P<pk>[0-9]+)/$', views.UserDetail.as\_view()),

[Associating Snippets with Users](http://www.django-rest-framework.org/tutorial/4-authentication-and-permissions/#associating-snippets-with-users)

Right now, if we created a code snippet, there'd be no way of associating the user that created the snippet, with the snippet instance. The user isn't sent as part of the serialized representation, but is instead a property of the incoming request.

The way we deal with that is by overriding a .perform\_create() method on our snippet views, that allows us to modify how the instance save is managed, and handle any information that is implicit in the incoming request or requested URL.

On the SnippetList view class, add the following method:

def perform\_create(self, serializer):

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

The create() method of our serializer will now be passed an additional 'owner' field, along with the validated data from the request.

[Updating our serializer](http://www.django-rest-framework.org/tutorial/4-authentication-and-permissions/#updating-our-serializer)

Now that snippets are associated with the user that created them, let's update our SnippetSerializer to reflect that. Add the following field to the serializer definition in serializers.py:

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

**Note**: Make sure you also add 'owner', to the list of fields in the inner Meta class.

This field is doing something quite interesting. The source argument controls which attribute is used to populate a field, and can point at any attribute on the serialized instance. It can also take the dotted notation shown above, in which case it will traverse the given attributes, in a similar way as it is used with Django's template language.

The field we've added is the untyped ReadOnlyField class, in contrast to the other typed fields, such as CharField, BooleanFieldetc... The untyped ReadOnlyField is always read-only, and will be used for serialized representations, but will not be used for updating model instances when they are deserialized. We could have also used CharField(read\_only=True) here.

[Adding required permissions to views](http://www.django-rest-framework.org/tutorial/4-authentication-and-permissions/#adding-required-permissions-to-views)

Now that code snippets are associated with users, we want to make sure that only authenticated users are able to create, update and delete code snippets.

REST framework includes a number of permission classes that we can use to restrict who can access a given view. In this case the one we're looking for is IsAuthenticatedOrReadOnly, which will ensure that authenticated requests get read-write access, and unauthenticated requests get read-only access.

First add the following import in the views module

from rest\_framework import permissions

Then, add the following property to **both** the SnippetList and SnippetDetail view classes.

permission\_classes = (permissions.IsAuthenticatedOrReadOnly,)

[Adding login to the Browsable API](http://www.django-rest-framework.org/tutorial/4-authentication-and-permissions/#adding-login-to-the-browsable-api)

If you open a browser and navigate to the browsable API at the moment, you'll find that you're no longer able to create new code snippets. In order to do so we'd need to be able to login as a user.

We can add a login view for use with the browsable API, by editing the URLconf in our project-level urls.py file.

Add the following import at the top of the file:

from django.conf.urls import include

And, at the end of the file, add a pattern to include the login and logout views for the browsable API.

urlpatterns += [

url(r'^api-auth/', include('rest\_framework.urls')),

]

The r'^api-auth/' part of pattern can actually be whatever URL you want to use.

Now if you open up the browser again and refresh the page you'll see a 'Login' link in the top right of the page. If you log in as one of the users you created earlier, you'll be able to create code snippets again.

Once you've created a few code snippets, navigate to the '/users/' endpoint, and notice that the representation includes a list of the snippet ids that are associated with each user, in each user's 'snippets' field.

[Object level permissions](http://www.django-rest-framework.org/tutorial/4-authentication-and-permissions/#object-level-permissions)

Really we'd like all code snippets to be visible to anyone, but also make sure that only the user that created a code snippet is able to update or delete it.

To do that we're going to need to create a custom permission.

In the snippets app, create a new file, 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

Now we can add that custom permission to our snippet instance endpoint, by editing the permission\_classes property on the SnippetDetail view class:

permission\_classes = (permissions.IsAuthenticatedOrReadOnly,

IsOwnerOrReadOnly,)

Make sure to also import the IsOwnerOrReadOnly class.

from snippets.permissions import IsOwnerOrReadOnly

Now, if you open a browser again, you find that the 'DELETE' and 'PUT' actions only appear on a snippet instance endpoint if you're logged in as the same user that created the code snippet.

[Authenticating with the API](http://www.django-rest-framework.org/tutorial/4-authentication-and-permissions/#authenticating-with-the-api)

Because we now have a set of permissions on the API, we need to authenticate our requests to it if we want to edit any snippets. We haven't set up any [authentication classes](http://www.django-rest-framework.org/api-guide/authentication/), so the defaults are currently applied, which are SessionAuthentication and BasicAuthentication.

When we interact with the API through the web browser, we can login, and the browser session will then provide the required authentication for the requests.

If we're interacting with the API programmatically we need to explicitly provide the authentication credentials on each request.

If we try to create a snippet without authenticating, we'll get an error:

http POST http://127.0.0.1:8000/snippets/ code="print 123"

{

"detail": "Authentication credentials were not provided."

}

We can make a successful request by including the username and password of one of the users we created earlier.

http -a admin:password123 POST http://127.0.0.1:8000/snippets/ code="print 789"

{

"id": 1,

"owner": "admin",

"title": "foo",

"code": "print 789",

"linenos": false,

"language": "python",

"style": "friendly"

}

[Summary](http://www.django-rest-framework.org/tutorial/4-authentication-and-permissions/#summary)

We've now got a fairly fine-grained set of permissions on our Web API, and end points for users of the system and for the code snippets that they have created.

In [part 5](http://www.django-rest-framework.org/tutorial/5-relationships-and-hyperlinked-apis/) of the tutorial we'll look at how we can tie everything together by creating an HTML endpoint for our highlighted snippets, and improve the cohesion of our API by using hyperlinking for the relationships within the system.