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 documentation instead.

Note: The code for this tutorial is available in the tomchristie/rest-framework-tutorial repository on GitHub. The completed implementation is also online as a sandbox version for testing, available here.

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
```

Note: To exit the virtualenv environment at any time, just type `deactivate`. For more information see the virtualenv documentation.

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')),
]
```

Okay, we're ready to roll.

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

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):
    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
```

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

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
# [{'pk': 1, 'title': u'', 'code': u'foo = "bar"\n', 'linenos': False, 'language':
u'python', 'style': u'friendly'}, {'pk': 2, 'title': u'', 'code': u'print "hello,
world"\n', 'linenos': False, 'language': u'python', 'style': u'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.

```
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

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)
```

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)

    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),
]
```

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

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 1.4.3, 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 using curl or httpie. 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

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 in part 2 of the tutorial.