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`.

```
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 `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)
```

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

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

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.

Next we'll move onto part 4 of the tutorial, where we'll take a look at how we can deal with authentication and permissions for our API.