Tutorial 5: Relationships & Hyperlinked APIs

At the moment relationships within our API are represented by using primary keys. In this part of the tutorial we'll improve the cohesion and discoverability of our API, by instead using hyperlinking for relationships.

Creating an endpoint for the root of our API

Right now we have endpoints for 'snippets' and 'users', but we don't have a single entry point to our API. To create one, we'll use a regular function-based view and the `@api_view` decorator we introduced earlier. In your `snippets/views.py` add:

```
from rest_framework.decorators import api_view
from rest_framework.response import Response
from rest_framework.reverse import reverse


@api_view(('GET',))
def api_root(request, format=None):
    return Response({
        'users': reverse('user-list', request=request, format=format),
        'snippets': reverse('snippet-list', request=request, format=format)
    })
```

Two things should be noticed here. First, we're using REST framework's `reverse` function in order to return fully-qualified URLs; second, URL patterns are identified by convenience names that we will declare later on in our `snippets/urls.py`.

Creating an endpoint for the highlighted snippets

The other obvious thing that's still missing from our pastebin API is the code highlighting endpoints.

Unlike all our other API endpoints, we don't want to use JSON, but instead just present an HTML representation. There are two styles of HTML renderer provided by REST framework, one for dealing with HTML rendered using templates, the other for dealing with pre-rendered HTML. The second renderer is the one we'd like to use for this endpoint.

The other thing we need to consider when creating the code highlight view is that there's no existing concrete generic view that we can use. We're not returning an object instance, but instead a property of an object instance.

Instead of using a concrete generic view, we'll use the base class for representing instances, and create our own `.get()` method. In your `snippets/views.py` add:

```
from rest_framework import renderers
from rest_framework.response import Response
```

```
class SnippetHighlight(generics.GenericAPIView):
    queryset = Snippet.objects.all()
    renderer_classes = (renderers.StaticHTMLRenderer,)

    def get(self, request, *args, **kwargs):
        snippet = self.get_object()
        return Response(snippet.highlighted)
```

As usual we need to add the new views that we've created in to our URLconf. We'll add a url pattern for our new API root in `snippets/urls.py`:

```
url(r'^$', views.api_root),
```

And then add a url pattern for the snippet highlights:

```
url(r'^snippets/(?P<pk>[0-9]+)/highlight/$', views.SnippetHighlight.as_view()),
```

Hyperlinking our API

Dealing with relationships between entities is one of the more challenging aspects of Web API design. There are a number of different ways that we might choose to represent a relationship:

- Using primary keys.
- Using hyperlinking between entities.
- Using a unique identifying slug field on the related entity.
- Using the default string representation of the related entity.
- Nesting the related entity inside the parent representation.
- Some other custom representation.

REST framework supports all of these styles, and can apply them across forward or reverse relationships, or apply them across custom managers such as generic foreign keys.

In this case we'd like to use a hyperlinked style between entities. In order to do so, we'll modify our serializers to extend `HyperlinkedModelSerializer` instead of the existing `ModelSerializer`.

The `HyperlinkedModelSerializer` has the following differences from `ModelSerializer`:

- It does not include the `pk` field by default.
- It includes a `url` field, using `HyperlinkedIdentityField`.
- Relationships use `HyperlinkedRelatedField`, instead of `PrimaryKeyRelatedField`.

We can easily re-write our existing serializers to use hyperlinking. In your `snippets/serializers.py` add:

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

```
    owner = serializers.ReadOnlyField(source='owner.username')
    highlight = serializers.HyperlinkedIdentityField(view_name='snippet-highlight',
format='html')

    class Meta:
        model = Snippet
        fields = ('url', 'highlight', 'owner',
                  'title', 'code', 'linenos', 'language', 'style')


class UserSerializer(serializers.HyperlinkedModelSerializer):
    snippets = serializers.HyperlinkedRelatedField(many=True,
view_name='snippet-detail', read_only=True)

    class Meta:
        model = User
        fields = ('url', 'username', 'snippets')
```

Notice that we've also added a new `'highlight'` field. This field is of the same type as the `url` field, except that it points to the `'snippet-highlight'` url pattern, instead of the `'snippet-detail'` url pattern.

Because we've included format suffixed URLs such as `'.json'`, we also need to indicate on the `highlight` field that any format suffixed hyperlinks it returns should use the `'.html'` suffix.

Making sure our URL patterns are named

If we're going to have a hyperlinked API, we need to make sure we name our URL patterns. Let's take a look at which URL patterns we need to name.

- The root of our API refers to `'user-list'` and `'snippet-list'`.
- Our snippet serializer includes a field that refers to `'snippet-highlight'`.
- Our user serializer includes a field that refers to `'snippet-detail'`.
- Our snippet and user serializers include `'url'` fields that by default will refer to `'{model_name}-detail'`, which in this case will be `'snippet-detail'` and `'user-detail'`.

After adding all those names into our URLconf, our final `snippets/urls.py` file should look like this:

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

# API endpoints
urlpatterns = format_suffix_patterns([
    url(r'^$', views.api_root),
    url(r'^snippets/$',
        views.SnippetList.as_view(),
        name='snippet-list'),
    url(r'^snippets/(?P<pk>[0-9]+)/$',
        views.SnippetDetail.as_view(),
        name='snippet-detail'),
```

```
    url(r'^snippets/(?P<pk>[0-9]+)/highlight/$',
        views.SnippetHighlight.as_view(),
        name='snippet-highlight'),
    url(r'^users/$',
        views.UserList.as_view(),
        name='user-list'),
    url(r'^users/(?P<pk>[0-9]+)/$',
        views.UserDetail.as_view(),
        name='user-detail')
])

# Login and logout views for the browsable API
urlpatterns += [
    url(r'^api-auth/', include('rest_framework.urls',
                               namespace='rest_framework')),
]
```

Adding pagination

The list views for users and code snippets could end up returning quite a lot of instances, so really we'd like to make sure we paginate the results, and allow the API client to step through each of the individual pages.

We can change the default list style to use pagination, by modifying our `tutorial/settings.py` file slightly. Add the following setting:

```
REST_FRAMEWORK = {
    'PAGE_SIZE': 10
}
```

Note that settings in REST framework are all namespaced into a single dictionary setting, named 'REST_FRAMEWORK', which helps keep them well separated from your other project settings.

We could also customize the pagination style if we needed too, but in this case we'll just stick with the default.

Browsing the API

If we open a browser and navigate to the browsable API, you'll find that you can now work your way around the API simply by following links.

You'll also be able to see the 'highlight' links on the snippet instances, that will take you to the highlighted code HTML representations.

In part 6 of the tutorial we'll look at how we can use ViewSets and Routers to reduce the amount of code we need to build our API.