Tutorial 6: ViewSets & Routers

REST framework includes an abstraction for dealing with `ViewSets`, that allows the developer to concentrate on modeling the state and interactions of the API, and leave the URL construction to be handled automatically, based on common conventions.

`ViewSet` classes are almost the same thing as `View` classes, except that they provide operations such as `read`, or `update`, and not method handlers such as `get` or `put`.

A `ViewSet` class is only bound to a set of method handlers at the last moment, when it is instantiated into a set of views, typically by using a `Router` class which handles the complexities of defining the URL conf for you.

Refactoring to use ViewSets

Let's take our current set of views, and refactor them into view sets.

First of all let's refactor our `UserList` and `UserDetail` views into a single `UserViewSet`. We can remove the two views, and replace them with a single class:

```
from rest_framework import viewsets

class UserViewSet(viewsets.ReadOnlyModelViewSet):
    """
    This viewset automatically provides `list` and `detail` actions.
    """
    queryset = User.objects.all()
    serializer_class = UserSerializer
```

Here we've used the `ReadOnlyModelViewSet` class to automatically provide the default 'read-only' operations. We're still setting the `queryset` and `serializer_class` attributes exactly as we did when we were using regular views, but we no longer need to provide the same information to two separate classes.

Next we're going to replace the `SnippetList`, `SnippetDetail` and `SnippetHighlight` view classes. We can remove the three views, and again replace them with a single class.

```
from rest_framework.decorators import detail_route
```

```
class SnippetViewSet(viewsets.ModelViewSet):
    """
    This viewset automatically provides `list`, `create`, `retrieve`,
    `update` and `destroy` actions.

    Additionally we also provide an extra `highlight` action.
    """
    queryset = Snippet.objects.all()
    serializer_class = SnippetSerializer
    permission_classes = (permissions.IsAuthenticatedOrReadOnly,
                          IsOwnerOrReadOnly,)

    @detail_route(renderer_classes=[renderers.StaticHTMLRenderer])
    def highlight(self, request, *args, **kwargs):
        snippet = self.get_object()
        return Response(snippet.highlighted)

    def perform_create(self, serializer):
        serializer.save(owner=self.request.user)
```

This time we've used the `ModelViewSet` class in order to get the complete set of default read and write operations.

Notice that we've also used the `@detail_route` decorator to create a custom action, named `highlight`. This decorator can be used to add any custom endpoints that don't fit into the standard `create`/`update`/`delete` style.

Custom actions which use the `@detail_route` decorator will respond to `GET` requests by default. We can use the `methods` argument if we wanted an action that responded to `POST` requests.

The URLs for custom actions by default depend on the method name itself. If you want to change the way url should be constructed, you can include url_path as a decorator keyword argument.

Binding ViewSets to URLs explicitly

The handler methods only get bound to the actions when we define the URLConf. To see what's going on under the hood let's first explicitly create a set of views from our ViewSets.

In the `urls.py` file we bind our `ViewSet` classes into a set of concrete views.

```
from snippets.views import SnippetViewSet, UserViewSet, api_root
from rest_framework import renderers

snippet_list = SnippetViewSet.as_view({
    'get': 'list',
    'post': 'create'
})
snippet_detail = SnippetViewSet.as_view({
    'get': 'retrieve',
    'put': 'update',
    'patch': 'partial_update',
```

```
    'delete': 'destroy'
})
snippet_highlight = SnippetViewSet.as_view({
    'get': 'highlight'
}, renderer_classes=[renderers.StaticHTMLRenderer])
user_list = UserViewSet.as_view({
    'get': 'list'
})
user_detail = UserViewSet.as_view({
    'get': 'retrieve'
})
```

Notice how we're creating multiple views from each `ViewSet` class, by binding the http methods to the required action for each view.

Now that we've bound our resources into concrete views, we can register the views with the URL conf as usual.

```
urlpatterns = format_suffix_patterns([
    url(r'^$', api_root),
    url(r'^snippets/$', snippet_list, name='snippet-list'),
    url(r'^snippets/(?P<pk>[0-9]+)/$', snippet_detail, name='snippet-detail'),
    url(r'^snippets/(?P<pk>[0-9]+)/highlight/$', snippet_highlight, name='snippet-highlight
    url(r'^users/$', user_list, name='user-list'),
    url(r'^users/(?P<pk>[0-9]+)/$', user_detail, name='user-detail')
])
```

Using Routers

Because we're using `ViewSet` classes rather than `View` classes, we actually don't need to design the URL conf ourselves. The conventions for wiring up resources into views and urls can be handled automatically, using a `Router` class. All we need to do is register the appropriate view sets with a router, and let it do the rest.

Here's our re-wired `urls.py` file.

```
from django.conf.urls import url, include
from snippets import views
from rest_framework.routers import DefaultRouter

# Create a router and register our viewsets with it.
router = DefaultRouter()
router.register(r'snippets', views.SnippetViewSet)
router.register(r'users', views.UserViewSet)

# The API URLs are now determined automatically by the router.
# Additionally, we include the login URLs for the browsable API.
urlpatterns = [
    url(r'^', include(router.urls))
```

```
    url(r'^api-auth/', include('rest_framework.urls', namespace='rest_framework'))
]
```

Registering the viewsets with the router is similar to providing a urlpattern. We include two arguments - the URL prefix for the views, and the viewset itself.

The `DefaultRouter` class we're using also automatically creates the API root view for us, so we can now delete the `api_root` method from our `views` module.

Trade-offs between views vs viewsets

Using viewsets can be a really useful abstraction. It helps ensure that URL conventions will be consistent across your API, minimizes the amount of code you need to write, and allows you to concentrate on the interactions and representations your API provides rather than the specifics of the URL conf.

That doesn't mean it's always the right approach to take. There's a similar set of trade-offs to consider as when using class-based views instead of function based views. Using viewsets is less explicit than building your views individually.

In part 7 of the tutorial we'll look at how we can add an API schema, and interact with our API using a client library or command line tool.
