source: viewsets.py

ViewSets

> After routing has determined which controller to use for a request, your controller is responsible for making sense of the request and producing the appropriate output.
>
> — Ruby on Rails Documentation

Django REST framework allows you to combine the logic for a set of related views in a single class, called a `ViewSet`. In other frameworks you may also find conceptually similar implementations named something like 'Resources' or 'Controllers'.

A `ViewSet` class is simply **a type of class-based View, that does not provide any method handlers** such as `.get()` or `.post()`, and instead provides actions such as `.list()` and `.create()`.

The method handlers for a `ViewSet` are only bound to the corresponding actions at the point of finalizing the view, using the `.as_view()` method.

Typically, rather than explicitly registering the views in a viewset in the urlconf, you'll register the viewset with a router class, that automatically determines the urlconf for you.

Example

Let's define a simple viewset that can be used to list or retrieve all the users in the system.

```
from django.contrib.auth.models import User
from django.shortcuts import get_object_or_404
from myapps.serializers import UserSerializer
from rest_framework import viewsets
from rest_framework.response import Response

class UserViewSet(viewsets.ViewSet):
    """
    A simple ViewSet for listing or retrieving users.
    """
    def list(self, request):
        queryset = User.objects.all()
        serializer = UserSerializer(queryset, many=True)
        return Response(serializer.data)

    def retrieve(self, request, pk=None):
        queryset = User.objects.all()
        user = get_object_or_404(queryset, pk=pk)
        serializer = UserSerializer(user)
        return Response(serializer.data)
```

If we need to, we can bind this viewset into two separate views, like so:

```
user_list = UserViewSet.as_view({'get': 'list'})
user_detail = UserViewSet.as_view({'get': 'retrieve'})
```

Typically we wouldn't do this, but would instead register the viewset with a router, and allow the urlconf to be automatically generated.

```
from myapp.views import UserViewSet
from rest_framework.routers import DefaultRouter

router = DefaultRouter()
router.register(r'users', UserViewSet)
urlpatterns = router.urls
```

Rather than writing your own viewsets, you'll often want to use the existing base classes that provide a default set of behavior. For example:

```
class UserViewSet(viewsets.ModelViewSet):
    """
    A viewset for viewing and editing user instances.
    """
    serializer_class = UserSerializer
    queryset = User.objects.all()
```

There are two main advantages of using a `ViewSet` class over using a `View` class.

- Repeated logic can be combined into a single class. In the above example, we only need to specify the `queryset` once, and it'll be used across multiple views.
- By using routers, we no longer need to deal with wiring up the URL conf ourselves.

Both of these come with a trade-off. Using regular views and URL confs is more explicit and gives you more control. ViewSets are helpful if you want to get up and running quickly, or when you have a large API and you want to enforce a consistent URL configuration throughout.

Marking extra actions for routing

The default routers included with REST framework will provide routes for a standard set of create/retrieve/update/destroy style operations, as shown below:

```
class UserViewSet(viewsets.ViewSet):
    """
    Example empty viewset demonstrating the standard
    actions that will be handled by a router class.

    If you're using format suffixes, make sure to also include
    the `format=None` keyword argument for each action.
    """

    def list(self, request):
        pass

    def create(self, request):
        pass

    def retrieve(self, request, pk=None):
        pass

    def update(self, request, pk=None):
        pass

    def partial_update(self, request, pk=None):
        pass

    def destroy(self, request, pk=None):
        pass
```

If you have ad-hoc methods that you need to be routed to, you can mark them as requiring routing using the `@detail_route` or `@list_route` decorators.

The `@detail_route` decorator contains `pk` in its URL pattern and is intended for methods which require a single instance. The `@list_route` decorator is intended for methods which operate on a list of objects.

For example:

```
from django.contrib.auth.models import User
from rest_framework import status
from rest_framework import viewsets
from rest_framework.decorators import detail_route, list_route
from rest_framework.response import Response
from myapp.serializers import UserSerializer, PasswordSerializer

class UserViewSet(viewsets.ModelViewSet):
    """
    A viewset that provides the standard actions
    """
    queryset = User.objects.all()
    serializer_class = UserSerializer

    @detail_route(methods=['post'])
    def set_password(self, request, pk=None):
        user = self.get_object()
        serializer = PasswordSerializer(data=request.data)
        if serializer.is_valid():
            user.set_password(serializer.data['password'])
            user.save()
            return Response({'status': 'password set'})
        else:
            return Response(serializer.errors,
                              status=status.HTTP_400_BAD_REQUEST)

    @list_route()
    def recent_users(self, request):
        recent_users = User.objects.all().order('-last_login')

        page = self.paginate_queryset(recent_users)
        if page is not None:
            serializer = self.get_serializer(page, many=True)
            return self.get_paginated_response(serializer.data)

        serializer = self.get_serializer(recent_users, many=True)
        return Response(serializer.data)
```

The decorators can additionally take extra arguments that will be set for the routed view only. For example...

```
    @detail_route(methods=['post'], permission_classes=[IsAdminOrIsSelf])
    def set_password(self, request, pk=None):
       ...
```

These decorators will route `GET` requests by default, but may also accept other HTTP methods, by using the `methods` argument. For example:

```
    @detail_route(methods=['post', 'delete'])
    def unset_password(self, request, pk=None):
       ...
```

The two new actions will then be available at the urls `^users/{pk}/set_password/$` and `^users/{pk}/unset_password/$`

API Reference

ViewSet

The `ViewSet` class inherits from `APIView`. You can use any of the standard attributes such as `permission_classes`, `authentication_classes` in order to control the API policy on the viewset.

The `ViewSet` class does not provide any implementations of actions. In order to use a `ViewSet` class you'll override the class and define the action implementations explicitly.

GenericViewSet

The `GenericViewSet` class inherits from `GenericAPIView`, and provides the default set of `get_object`, `get_queryset` methods and other generic view base behavior, but does not include any actions by default.

In order to use a `GenericViewSet` class you'll override the class and either mixin the required mixin classes, or define the action implementations explicitly.

ModelViewSet

The `ModelViewSet` class inherits from `GenericAPIView` and includes implementations for various actions, by mixing in the behavior of the various mixin classes.

The actions provided by the `ModelViewSet` class are `.list()`, `.retrieve()`, `.create()`, `.update()`, and `.destroy()`.

Example

Because `ModelViewSet` extends `GenericAPIView`, you'll normally need to provide at least the `queryset` and `serializer_class` attributes. For example:

```
class AccountViewSet(viewsets.ModelViewSet):
    """
    A simple ViewSet for viewing and editing accounts.
    """
    queryset = Account.objects.all()
    serializer_class = AccountSerializer
    permission_classes = [IsAccountAdminOrReadOnly]
```

Note that you can use any of the standard attributes or method overrides provided by `GenericAPIView`. For example, to use a `ViewSet` that dynamically determines the queryset it should operate on, you might do something like this:

```
class AccountViewSet(viewsets.ModelViewSet):
    """
    A simple ViewSet for viewing and editing the accounts
    associated with the user.
    """
    serializer_class = AccountSerializer
    permission_classes = [IsAccountAdminOrReadOnly]

    def get_queryset(self):
        return self.request.user.accounts.all()
```

Note however that upon removal of the `queryset` property from your `ViewSet`, any associated router will be unable to derive the base_name of your Model automatically, and so you will have to specify the `base_name` kwarg as part of your router registration.

Also note that although this class provides the complete set of create/list/retrieve/update/destroy actions by default, you can restrict the available operations by using the standard permission classes.

ReadOnlyModelViewSet

The `ReadOnlyModelViewSet` class also inherits from `GenericAPIView`. As with `ModelViewSet` it also includes implementations for various actions, but unlike `ModelViewSet` only provides the 'read-only' actions, `.list()` and `.retrieve()`.

Example

As with `ModelViewSet`, you'll normally need to provide at least the `queryset` and `serializer_class` attributes. For example:

```
class AccountViewSet(viewsets.ReadOnlyModelViewSet):
    """
    A simple ViewSet for viewing accounts.
    """
    queryset = Account.objects.all()
    serializer_class = AccountSerializer
```

Again, as with `ModelViewSet`, you can use any of the standard attributes and method overrides available to `GenericAPIView`.

Custom ViewSet base classes

You may need to provide custom `ViewSet` classes that do not have the full set of `ModelViewSet` actions, or that customize the behavior in some other way.

Example

To create a base viewset class that provides `create`, `list` and `retrieve` operations, inherit from `GenericViewSet`, and mixin the required actions:

```
class CreateListRetrieveViewSet(mixins.CreateModelMixin,
                                mixins.ListModelMixin,
                                mixins.RetrieveModelMixin,
                                viewsets.GenericViewSet):
    """
    A viewset that provides `retrieve`, `create`, and `list` actions.

    To use it, override the class and set the `.queryset` and
    `.serializer_class` attributes.
    """
    pass
```

By creating your own base `ViewSet` classes, you can provide common behavior that can be reused in multiple viewsets across your API.