Learning Objectives

- **Define a viewset**
- **Identify the methods available to viewsets and their associated HTTP request**
- **Differentiate a modelviewset from a viewset**
- **Define the interaction between a router and a viewset**
- **Create a viewset that uses the appropriate serializer**
- **Add extra actions to a viewset**

Clone Blango Repo

Clone Blango Repo

Before we continue, you need to clone the `blango` repo so you have all of your code. You will need the SSH information for your repo.

In the Terminal

- Clone the repo. Your command should look something like this:

```
git clone git@github.com:<your_github_username>/blango.git
```

- You should see a `blango` directory appear in the file tree.

You are now ready for the next assignment.

Viewsets

Viewsets

Django Rest Framework has one more trick up its sleeve in regards to reducing the amount of code to write: viewsets. This allows you to define a single class which will handle both the list and detail API for a given model.

To define a viewset, subclass `rest_framework.viewsets.ViewSet` and implement the action methods you want the viewset to respond to. The methods available to implement are:

- `list(self, request)`: Called during an object list `GET` request.
- `create(self, request)`: Called for a `POST` request.
- `retrieve(self, request, pk=None)`: Called when performing a `GET` of a single object.
- `update(self, request, pk=None)`: Called for a `PUT` request of a single object.
- `partial_update(self, request, pk=None)`: Called for a `PATCH` request to partially update a single object.
- `destroy(self, request, pk=None)`: Called for a `DELETE` request.

You can implement all, or just some of these, depending on what you want your viewset to be able to support. Each method should return a `rest_framework.response.Response` instance (like what we used when we first built our function-based DRF API).

For example, here's a viewset that allows listing all `Tag` objects in Blango (assuming we've written a `TagSerializer` class already).

```
class TagViewSet(viewsets.ViewSet):
    def list(self, request):
        queryset = Tag.objects.all()
        serializer = TagSerializer(queryset, many=True)
        return Response(serializer.data)
```

However, subclassing `ViewSet` doesn't save us much code compared to what we have so far, instead, we should use a `ModelViewSet`. This has already implemented all the methods we listed above, and can automatically read, create, update and delete model instances.

Here's how to define a viewset for the `Tag` model that implements list, create, retrieve, update and destroy:

```
class TagViewSet(viewsets.ModelViewSet):
    queryset = Tag.objects.all()
    serializer_class = TagSerializer
```

That's all that needs to be done. How do we map URLs to a viewset? It is possible to "extract" the views from the viewset and map them individually. We need to set up mappings from HTTP methods to our viewset methods (`list()`, `create()`, etc). For example:

```
tag_list = TagViewSet.as_view({
    "get": "list",
    "post": "create"
})

tag_detail = TagViewSet.as_view({
    "get": "retrieve",
    "put": "update",
    "patch": "partial_update",
    "delete": "destroy"
})
```

Then map the URLs to these psuedo-views:

```
    path("tags/", tag_list, name="tag_list"),
    path("tags/<int:pk>/", tag_detail, name="tag_detail"),
```

But as usual, DRF provides a method to make this simpler: routers.

Routers

Routers

A DRF `Router` will inspect a viewset and determine what endpoints it has available, then create the right URL patterns automatically. Usually you will want to use a `rest_framework.routers.DefaultRouter` class. Also provided is `rest_framework.routers.SimpleRouter`, but we'll come back to the differences in a moment.

To use a router take three steps.

1. Instantiate the router:

```
router = DefaultRouter()
```

2. Register viewsets using a prefix. The prefix will be used as the initial component of the path. For example to register our `TagViewSet` under the `tags` path:

```
router.register("tags", TagViewSet)
```

3. Access the generated URL patterns with the `urls` attribute on the router. These can be appended to `urlpatterns` or used with the `include()` function. For example:

```
urlpatterns += [
    # other patterns
    path("", include(router.urls)),
]
```

We can actually `register()` multiple `ViewSets` and all the URL patterns will be available in the `urls` attribute.

To return to the difference between `SimpleRouter` and `DefaultRouter`: `DefaultRouter` will also include a default base route with no prefix, which returns a list of links to the registered viewsets. It also adds the format routes (i.e. fetching a JSON formatted response by adding the `.json` extension to the URL).

Try It Out

We've seen how easy it is to set up API endpoints using viewsets and routers, so let's now set up the `Tag` API in Blango.

Start in `blog/api/serializers.py`. Create a new `TagSerializer`; make sure you import the `Tag` model too:

```
class TagSerializer(serializers.ModelSerializer):
    class Meta:
        model = Tag
        fields = "__all__"
```

Next open `blog/api/views.py` and import `viewsets` from `rest_framework`:

Open api/views.py

```
from rest_framework import generics, viewsets
```

And make sure you're importing the new `TagSerializer` class and the `Tag` model:

```
from blog.api.serializers import (
    PostSerializer,
    UserSerializer,
    PostDetailSerializer,
    TagSerializer,
)
from blog.models import Post, Tag
```

Then you can define the `TagViewSet`.

```
class TagViewSet(viewsets.ModelViewSet):
    queryset = Tag.objects.all()
    serializer_class = TagSerializer
```

Now we'll make use of the router. Open `blog/api/urls.py` and import the `DefaultRouter` class:

Open api/urls.py

```
from rest_framework.routers import DefaultRouter
```

And the `TagViewSet` class:

```
from blog.api.views import PostList, PostDetail, UserDetail,
        TagViewSet
```

Then instantiate the router and register the viewset.

```
router = DefaultRouter()
router.register("tags", TagViewSet)
```

Finally, add the router's `urls` to the `urlpatterns`:

```
urlpatterns += [
    path("auth/", include("rest_framework.urls")),
    # ... other patterns omitted
    path("", include(router.urls)),
]
```

That's it. Load up the `/api/v1/tags/` path in a browser or Postman, and you should see a list of the tags in the system. If you like, you can set up the various routes in Postman to try creating, editing and deleting them too.

View Blog

Customizing Viewsets

Customizing Viewsets

Could we convert our other views to viewsets? We wouldn't gain much from converting the `UserDetail` view as it's just a single class anyway, and we would have to write extra code to disable most of the functionality. But we can convert our `Post` views to viewsets. We'll just need to handle having a different serializer for the list and detail methods.

We can do this by implementing the `get_serializer_class()` method on the new viewset. We can vary what's returned by looking at `self.action` string. This will be one of `list`, `create`, `retrieve`, etc as per the methods that can be implemented on the viewset. If the `action` is `list` or `create`, we should use the `PostSerializer`, otherwise use `PostDetailSerializer`.

```
class PostViewSet(viewsets.ModelViewSet):
    permission_classes = [AuthorModifyOrReadOnly |
        IsAdminUserForObject]
    queryset = Post.objects.all()

    def get_serializer_class(self):
        if self.action in ("list", "create"):
            return PostSerializer
        return PostDetailSerializer
```

This class can replace both our `Post` related views, and we can remove the URL patterns too, as they'll be automatically added using the router.

Try It Out

Let's refactor our `Post` API down to a single viewset. In `blog/api/views.py` add the `PostViewSet` class:

```
class PostViewSet(viewsets.ModelViewSet):
    permission_classes = [AuthorModifyOrReadOnly |
        IsAdminUserForObject]
    queryset = Post.objects.all()

    def get_serializer_class(self):
        if self.action in ("list", "create"):
            return PostSerializer
        return PostDetailSerializer
```

Then you can delete the `PostList` and `PostDetail` views.

Next go to `blog/api/urls.py`. Remove the import of `PostDetail` and `PostList` (they'd fail anyway). Be sure to import `PostViewSet`.

Open api/urls.py

```
from blog.api.views import UserDetail, TagViewSet, PostViewSet
```

Then remove the URL pattern for them, these two lines:

```
    path("posts/", PostList.as_view(), name="api_post_list"),
    path("posts/<int:pk>", PostDetail.as_view(),
        name="api_post_detail"),
```

Your initial `urlpatterns` creation should now just contain the map to the `UserDetail` view.

```
urlpatterns = [
    path("users/<str:email>", UserDetail.as_view(),
        name="api_user_detail"),
]
```

Finally register the `PostViewSet` on the router:

```
router.register("posts", PostViewSet)
```

Jump into Postman or load the API in the browser and try it out. You should notice that the `Posts` API behaves the same as it did before, however we've managed to reduce the amount of code that we need to write. This also helps ensure consistency when it comes to building and naming the URLs.

View Blog

On that note, how are the URLs that a router generates, named? DRF uses the name of model from the queryset, then the suffixes `-list` and `-detail` are added. For example, our "post detail" view now has the name `post-detail`, rather than `api_post_detail` as it did before. We could customize this by providing a `basename` string as the third argument to the `register()` method.

Extra Viewset Actions

Extra Viewset Actions

Viewsets also support extra "actions", that is, extra methods that can be added with their own URL. The methods are implemented on the viewset, and look similar to a method that you'd implement on a view class. The method will accept a `request`, and optionally, the `pk` of the object (depending on how you want the method to be used). We also need to decorate the method with the `rest_framework.decorators.action` function, which will indicate to the router that a URL needs to be mapped to it, and also how the method should receive data.

Let's look at an example and then discuss it in more detail. We'll create an extra method on the `TagViewSet` that allows us to get a list of all `Post` objects that have the specific `Tag`. Here's how it's implemented.

```
from rest_framework.decorators import action
from rest_framework.response import Response


class TagViewSet(viewsets.ModelViewSet):
    queryset = Tag.objects.all()
    serializer_class = TagSerializer

    @action(methods=["get"], detail=True, name="Posts with the
        Tag")
    def posts(self, request, pk=None):
        tag = self.get_object()
        post_serializer = PostSerializer(
            tag.posts, many=True, context={"request": request}
        )
        return Response(post_serializer.data)
```

The path that is generated for this method is `/api/v1/tags/<pk>/posts/` (the name of the method being appended to the detail URL for a `Tag`). The behavior of the method should be easy to understand, even though we're doing a couple of new things.

We have access to the `pk` from the URL, so we could fetch the `Tag` object from the database ourselves. However, the `ModelViewSet` class provides a helper method that will do that for us – `get_object()` – so we use that instead.

Then we use the `PostSerializer` to serialize the many `Post` objects on `tag.post`. Since `PostSerializer` uses a `HyperlinkRelatedField` it needs access to the current `request` so we need to pass that in a `context` dictionary. Then we just return a DRF `Response` with the serialized data.

As we mentioned, the method needs to be decorated with the `action` decorator to indicate that a URL should be set up for it. The `action` method takes these parameters to configure how the method works:

- `methods`: A list of HTTP methods that the action will respond to. Defaults to `["get"]`.
- `detail`: Determines if the action should apply to detail requests (if `True`) or list (if `False`). This argument is required.
- `url_path`: Manually specify the path to be used in the URL. Defaults to the method name (e.g. `posts`).
- `url_name`: Manually specify the name of the URL pattern. Defaults to the method name with underscores replaced by dashes. The full name of our method's URL is `tag-posts`.
- `name`: A name to display in the *Extra Actions* menu in the DRF GUI. Defaults to the name of the method.

Try It Out

Let's implement the `posts` method on your version of Blango. Open `blog/api/views.py`. Start by adding the new imports at the top of the file:

```
from rest_framework.decorators import action
from rest_framework.response import Response
```

Then add the `posts` method on `TagViewSet`.

```
class TagViewSet(viewsets.ModelViewSet):
    # existing attributes omitted

    @action(methods=["get"], detail=True, name="Posts with the
        Tag")
    def posts(self, request, pk=None):
        tag = self.get_object()
        post_serializer = PostSerializer(
            tag.posts, many=True, context={"request": request}
        )
        return Response(post_serializer.data)
```

Those are the only changes we need to make, the router will automatically read the new method and generate the URL pattern for us. Load up a Tag detail view in DRF in a browser and you'll see the *Extra Actions* menu. You can then click **Posts with this Tag** to go to the posts list for the tag.

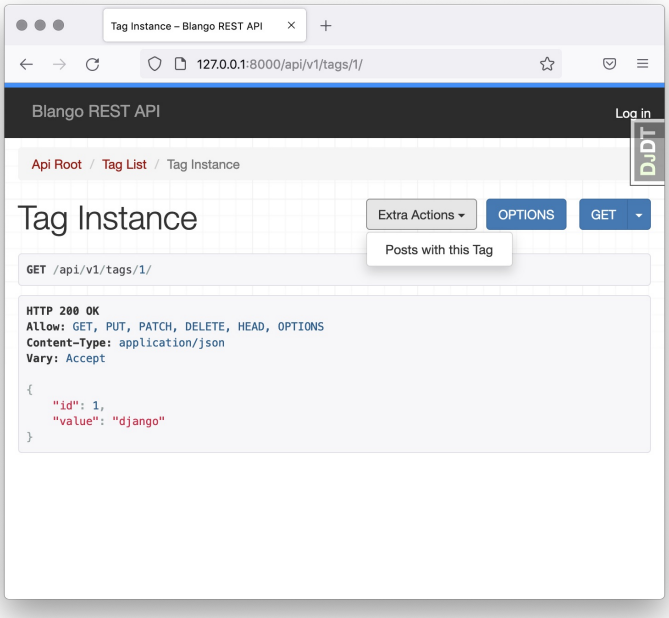
View Blog

info

Open Blango in a Separate Tab

You can only use the **Posts with this Tag** action if you open the blog in its own tab. Click the blue arrow next to the refresh icon to open Blango in a separate tab.

blue arrow icon



extra actions

Or you can visit the URL directly, of course.

Viewsets and routers seem great, but what are the drawbacks of them compared to views?

Viewsets vs Views

Viewsets do provide a useful abstraction and can be a really quick way of building an API, however as with any type of “automatic” or “one size fits all” approach, they may not always do what you want. We decided not to convert our user API to a viewset as we did not require all the functionality

it would have given us, and in fact we would have to write extra code to strip it out. So as usual, it comes down to the individual case to decide whether to use a viewset or not.

If you're unsure, check out the viewsets documentation for a more in-depth look at how to customize viewsets to further suit your needs.

Wrap-Up

That's the end of the module and indeed the whole course. We continue our look at Django Rest Framework in the next course, starting with how to test DRF views.

Pushing to GitHub

Pushing to GitHub

Before continuing, you must push your work to GitHub. In the terminal:

- Commit your changes:

```
git add .
git commit -m "Finish viewsets and routers"
```

- Push to GitHub:

```
git push
```