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.

Intro

Related Fields

Intro

We've already seen some related fields in Django Rest Framework, namely, our `tag` and `author` fields. You will have noticed they just come back as integers - their IDs. This is not ideal because if a client wants to find out what tag `3` means, they have to make another request to fetch it. This is especially difficult considering we have not implemented a `Tag` API!

Similarly if a client wanted to show the author's name next to each `Post`, they'd have to make another API request for each one.

Let's take a look at how DRF can help with this, and the different types of relationship fields it has available. These fields are set on `Serializer` classes.

Related Fields

`PrimaryKeyRelatedField`

This is actually what we've been using so far, the DRF `ModelSerializer` class sets it up automatically for us behind the scenes. When `PostSerializer` serializes an `author` or `tags`, it renders the related model's primary key field. It automatically determines if it's a `ForeignKey`, `OneToOneField` or `ManyToManyField` and serializes as a single value or list as appropriate.

When using a `ModelSerializer` this field is created read and write, but as we saw earlier, we can override individual field definitions. To make the `tags` read only, for example, we could add the `tags` field to the `PostSerializer` definition:

```
class PostSerializer(serializers.ModelSerializer):
    tags = serializers.PrimaryKeyRelatedField(read_only=True,
        many=True)

    class Meta:
        model = Post
        fields = "__all__"
        readonly = ["modified_at", "created_at"]
```

The arguments that can be passed to `PrimaryKeyRelatedField` are:

- `queryset`: The queryset used to look up the model instances. This is required, but if used with `ModelSerializer` then the queryset can be determined automatically.
- `many`: If it's a to-many relationship, this must be `True`.
- `allow_null`: if the underlying field allows null, then set this to `True` to allow `None` or empty strings in the field.
- `pk_field`: Pass in a different serializer field instance that will be used to serialize/deserialize the primary key field value. This is useful if the primary key on your model is not the default integer.

While `PrimaryKeyRelatedField` is the simplest way of serializing relationships, we've already pointed out some of the problems with it. Let's look at some alternatives.

`StringRelatedField`

`StringRelatedField` (`rest_framework.serializers.StringRelatedField`) is a serializer field that calls the `__str__()` method of the related object that's being serialized. It's easy to implement, for example on our `PostSerializer`:

```
class PostSerializer(serializers.ModelSerializer):
    tags = serializers.StringRelatedField(many=True)

    class Meta:
        model = Post
        fields = "__all__"
        readonly = ["modified_at", "created_at"]
```

Then our `Post` detail response looks like this:

```
{
    "id": 6,
    "tags": [
        "django",
        "python"
    ],
    "created_at": ...
}
```

This is a simple solution, however it is read only, so we can't use it to write data. This is because DRF doesn't know how to "reverse" the `__str__()` method to convert a string back to a primary key.

There's another field which has a couple of extra steps to set up but lets us read and write.

`SlugRelatedField`

This is inteded to work with a `SlugField` of a related object, although it can work with any unique field. The arguments that `SlugRelatedField` take are:

- `slug_field`: The name of the field on the related object, it should be unique (although as we'll see, DRF doesn't check this).
- `queryset`: The queryset used to fetch related models. It will be further filtered by the field's value.
- `many`: `True` for to-many relationships.
- `allow_null`: If set to `True` (it defaults to `False`) then allow setting the value to `None` or empty string (if the underlying field allows it).
- `read_only`: Set to `True` to make field read only.

We can set it up on `PostSerializer` like this:

```
class PostSerializer(serializers.ModelSerializer):
    tags = serializers.SlugRelatedField(
        slug_field="value", many=True,
        queryset=Tag.objects.all()
    )

    class Meta:
        model = Post
        fields = "__all__"
        readonly = ["modified_at", "created_at"]
```

We're mapping it to the `value` field on the `Tag` which you may recall (or you can check), is not unique. Provided you don't have any duplicate tag values in your database, then this will work fine. However, if we were to continue treating `value` as a slug, we'd want to add a `unique=True` constraint to it.

That being said, here's what the response looks like:

```
{
    "id":6,
    "tags": [
        "django",
        "test"],
    "created_at": ...
```

We can also update the tags, provided a `Tag` with the given value already exists in the database.

The last related field we're going to look at is `HyperlinkedRelatedField`.

HyperlinkRelatedField

`HyperlinkRelatedField`

This field serializes a related object to a URL at which we can retrieve the full detail of the object. It requires the name of a view to be provided, and this is used to generate the URL by passing in the primary key of the related object.

In order to demonstrate this, we'll need another API view, which also means another serializer. We'll make an API for the `User` model to demonstrate how to use `author` as a `HyperlinkRelatedField` field.

First, the serializer:

```
class UserSerializer(serializers.ModelSerializer):
    class Meta:
        model = User
        fields = ["first_name", "last_name", "email"]
```

Then we don't want to be able to list users, instead, let's just implement the detail API:

```
class UserDetail(generics.RetrieveAPIView):
    queryset = User.objects.all()
    serializer_class = UserSerializer
```

Notice here we're using a generic view class we haven't seen before, `RetrieveAPIView`. This allows retrieval of objects but not updating or deleting them.

Then the last thing to do is add the URL pattern:

```
    path("users/<int:pk>", UserDetail.as_view(),
        name="api_user_detail"),
```

Back to the `HyperlinkRelatedField` setup. In its simplest form we can just add one to the `PostSerializer` like this:

```
class PostSerializer(serializers.ModelSerializer):
    author = serializers.HyperlinkedRelatedField(
        queryset=User.objects.all(),
        view_name="api_user_detail"
    )
    # other fields omitted for brevity
```

When serializing the `author` now, DRF will use Django's URL reversing functionality, using the name of the view, `api_user_detail`. It will automatically pass the author's primary key into the URL.

Now when we `GET` a `Post`, it looks like this:

```
{
    "id": 6,
    "tags": [
        "django",
        "test"
    ],
    "author": "http://127.0.0.1:8000/api/v1/users/6",
    "created_at": "..."
}
```

We can then visit that URL to retrieve the author's `User` object in JSON format. And, if we're using the DRF web UI, the link will even be made clickable to make it easier to navigate.

The way that we've implemented our API though, is a security issue. Although we don't have a way of listing all users, we are vulnerable to an enumeration attack: an attacker could try to fetch `/api/v1/users/1`, then `/api/v1/users/2`, and so on, and download information about every user in our system.

Instead, let's change it so the email address of the `User` is used in the URL. We'll need to customize the `HyperlinkRelatedField` to make this work.

Customizing `HyperlinkRelatedField`

First we need to make some changes to the `UserDetail` view. We need to tell DRF to perform the User query on `email` instead of `pk`. To do this we add the attribute `lookup_field = "email"` to the `UserDetail` class:

```
class UserDetail(generics.RetrieveAPIView):
    lookup_field = "email"
    queryset = User.objects.all()
    serializer_class = UserSerializer
```

Then update the URL pattern to pass `email` instead of `pk`.

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

Now let's briefly look at the arguments that can be passed to `HyperlinkedRelatedField`.

- `view_name`: The name of the view to use to build the URL, as per the URL patterns. This is required.
- `queryset`: The queryset used to fetch related models. It will be further filtered by the field's value.
- `many`: Set `True` for a to-many relationship.
- `allow_null`: Allow `None`/empty string to be set for the value (if the underlying field allows it).
- `lookup_field`: The name of the field on the related object to use in the URL building. This defaults to `pk`.
- `lookup_url_kwarg`: The name of the keyword argument in the URL pattern to pass the `lookup_field` value to. This default to the same value as `lookup_field`.
- `format`: The format argument to be passed, e.g. `json`. This can be left blank and will default to the same format as the original response.

You can see by looking at these arguments that to use the email address in the URL we should provide the `lookup_field` argument, with a value of `email`. Thus our serializer class will look like this:

```
class PostSerializer(serializers.ModelSerializer):
    tags = serializers.SlugRelatedField(
        slug_field="value", many=True,
        queryset=Tag.objects.all()
    )

    author = serializers.HyperlinkedRelatedField(
        queryset=User.objects.all(),
        view_name="api_user_detail",
        lookup_field="email"
    )

    class Meta:
        model = Post
        fields = "__all__"
        readonly = ["modified_at", "created_at"]
```

`HyperlinkedRelatedField` is read and write. To update the `author`, we just need to supply the full URL to the author's detail page. DRF can parse this to find out the user's email and perform the lookup for us.

Try It Out

Try It Out

Let's make use of some custom serializers to make our Blango `Post` API a bit more readable.

As a precursor, if you want, you can give the `value` field on the `Tag` model a `unique` constraint:

```
    value = models.TextField(max_length=100, unique=True)
```

Make sure to run the `makemigrations` and `migrate` command afterward if you do decide to do this.

We'll then set up the user API, first by creating a `UserSerializer` class in the `blog/api/serializers.py` file, like this:

Open api/serializers.py

```
class UserSerializer(serializers.ModelSerializer):
    class Meta:
        model = User
        fields = ["first_name", "last_name", "email"]
```

Make sure to import the `User` and `Tag` models at the top of the file too:

```
from blog.models import Post, Tag
from blango_auth.models import User
```

We're including just a subset of fields, because there's sensitive data like the password hash which we don't want to include.

Then we'll override the `tags` and `author` fields on the `PostSerializer` class.

```
class PostSerializer(serializers.ModelSerializer):
    tags = serializers.SlugRelatedField(
        slug_field="value", many=True,
        queryset=Tag.objects.all()
    )

    author = serializers.HyperlinkedRelatedField(
        queryset=User.objects.all(),
        view_name="api_user_detail", lookup_field="email"
    )

    class Meta:
        model = Post
        fields = "__all__"
        readonly = ["modified_at", "created_at"]
```

The `Post` API won't work now, until we have added the `api_user_detail` view, so let's do that now.

Create the `UserDetail` view class inside `blog/api/views.py`:

Open api/views.py

```
class UserDetail(generics.RetrieveAPIView):
    lookup_field = "email"
    queryset = User.objects.all()
    serializer_class = UserSerializer
```

Make sure to import the `User` model and `UserSerializer` class at the top of the file too:

```
from blango_auth.models import User
from blog.api.serializers import PostSerializer, UserSerializer
```

Finally, open `blog/api/urls.py` and add the mapping for `UserDetail` to the `urlpatterns`:

Open api/urls.py

```
urlpatterns = [
    path("posts/", PostList.as_view(), name="api_post_list"),
    path("posts/<int:pk>", PostDetail.as_view(),
        name="api_post_detail"),
    path("users/<str:email>", UserDetail.as_view(),
        name="api_user_detail"),
]
```

Try it out with Postman or the DRF GUI. You could add a new folder to Postman, containing a *User Detail* request.

View Blog

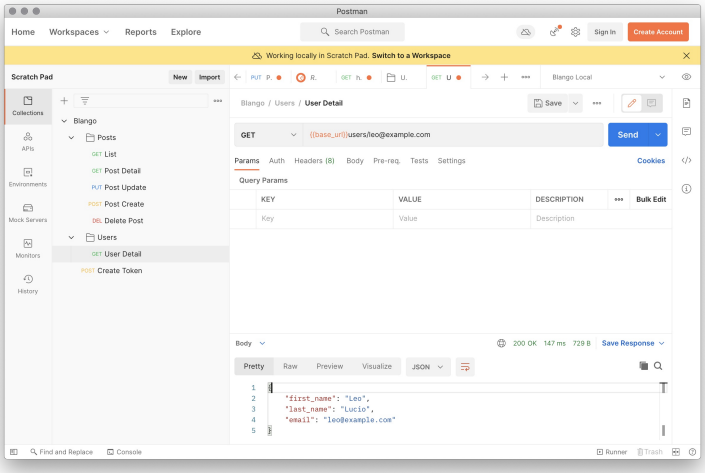

Figure 1

The detail response should look like this:

```
{
    "first_name": "Leo",
    "last_name": "Lucio",
    "email": "leo@example.com"
}
```

Then try out a `Post` request, you should see the author come back as a URL, and the tags as a list of values. Try updating the tags and/or author. When providing a list of tags, you'll need to give tags that already exist; and when updating the author, provide a URL to an author that is already in the system.

Further customization of `HyperlinkRelatedField`

There may be some instances where the `HyperlinkRelatedField`'s way of generating a URL is too simple for your needs: usually if you need to use two or more arguments in URL generation. For example, if we wanted to allow access to `Posts` for a particular author, we might have a URL path like `/api/v1/<author_email>/posts/<post_id>`. In this case, we'd need to provide both the author's email and a `Post` ID to generate the URL.

We won't go through a full example of how to achieve this, but in short, you need to subclass `HyperlinkedRelatedField` and implemented two methods. The first is `get_url(self, obj, view_name, request, format)`, which will return the URL of the related object. The second is `get_object(self,`

`view_name, view_args, view_kwargs)` which should use the view name and arguments to fetch the related object and return it. The official DRF documentation provides full explanation.

Django Rest Framework has another way of showing related fields: by nesting them inside the original response itself. We'll look at that in the next section.

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 related fields"
```

- Push to GitHub:

```
git push
```