Learning Objectives

- **Define a read-only nested relationship**
- **Define a read-write nested relationship**
- **Implement an `update` method to avoid a race condition**
- **Use the `get_or_create` method to automatically create related objects**
- **Modify Blango to create and see comments through the API**
- **Modify Blango to create tags through the API**

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

Intro

Instead of giving API clients a reference to a related entity with an ID or a URL, we can nest the related entity directly into the current one. This is easy to do with a read-only relationship, and takes a little bit more work for a read-write relationship, but is still not too hard.

Read-Only Nested Relationships

Read-Only Nested Relationships

Creating a read-only nested relationship essentially just means using a `Serializer` subclass as a field.

For example, in Blango, we would like to include the comments in our `Post` detail API (but not in our `Post` list, as we'd need to make lots of extra database queries). To do this, first we need a way of serializing comments: a `CommentSerializer` class:

```
class CommentSerializer(serializers.ModelSerializer):
    creator = UserSerializer(read_only=True)

    class Meta:
        model = Comment
        fields = ["id", "creator", "content", "modified_at",
"created_at"]
        readonly = ["modified_at", "created_at"]
```

Then, to have different fields on different views, we can use different serializers for the same model. In our case, we can create a `PostDetailSerializer` which inherits from the `PostSerializer` we were using before, but adds a `comments` field:

```
class PostDetailSerializer(PostSerializer):
    comments = CommentSerializer(many=True, read_only=True)
```

This will pick up the `Comment` objects from the `comments` field on the `Post` object, and then serialize them with the `CommentSerializer`. Note the use of the `many=True` argument since it's a to-many relationship, and `read_only=True` (for now).

Then we'll change our `PostDetail` view to use the `PostDetailSerializer`.

```
class PostDetail(generics.RetrieveUpdateDestroyAPIView):
    serializer_class = PostDetailSerializer
    # other attributes remain unchanged
```

Then our `Post` list response remains unchanged, but our `Post` detail response now contains comments:

```
{
    "id": 6,
    "tags": [
        "django",
        "test"
    ],
    "author":
        "http://127.0.0.1:8000/api/v1/users/ben@example.com",
    "comments": [
        {
            "id": 7,
            "creator": {
                "first_name": "",
                "last_name": "",
                "email": "ben@example.com"
            },
            "content": "Good point, this needs to be said more
        often.",
            "modified_at": "2021-07-23T20:47:33.093000Z",
            "created_at": "2021-07-23T20:47:33.092000Z"
        },
       ...
    ],
    ...
```

You can see with just a few changes we can nest related objects inside our object. Now, let's look at the additional work to make the field writable.

Read-Write Nested Relationships

Read-Write Nested Relationships

In order to save a nested relationship, we need to implement the `update()` and/or `create()` method(s) on our main serializer (i.e. `PostDetailSerializer` in our case).

Since we're just using `PostDetailSerializer` in our detail view, we only need to implement the `update()` method, since `create()` will only be called for a `POST` request on the list view.

How the `update()` method is implemented is really up to you (the project developer) to decide. You could decide that data in the database should match the `PUT` body after save, so if an empty `comments` list were to be sent then all comments for that `Post` would be deleted. But this would also cause a race condition.

Take a scenario where user *A* sends back the `Post` body with comments 1, 2, and a new comment, then user *B* sends their `Post` body with comments 1, 2, and *their* new comment. The second set of comments would override the first so user A's comments would be lost.

Instead, we'll just treat any comment that doesn't have an `id` field set as a new one, and add it to the `Post`.

First we'll need to make a change to the `CommentSerializer`: add an `id` field.

```
class CommentSerializer(serializers.ModelSerializer):
    id = serializers.IntegerField(required=False)
    creator = UserSerializer(read_only=True)

    class Meta:
        model = Comment
        fields = ["id", "creator", "content", "modified_at",
        "created_at"]
        readonly = ["modified_at", "created_at"]
```

But didn't we have an `id` field before? Yes, but we're overriding it to set `required` to `False`. Without this change, the `validated_data` that's passed to the `update()` method won't contain the `id` at all, so we won't be able to determine which comments are new or not.

Next let's look at how to implement the `update()` method on `PostDetailSerializer`. We'll see it in full then go through it in more detail.

```
class PostDetailSerializer(PostSerializer):
    comments = CommentSerializer(many=True)

    def update(self, instance, validated_data):
        comments = validated_data.pop("comments")

        instance = super(PostDetailSerializer,
        self).update(instance, validated_data)

        for comment_data in comments:
            if comment_data.get("id"):
                # comment has an ID so was pre-existing
                continue
            comment = Comment(**comment_data)
            comment.creator = self.context["request"].user
            comment.content_object = instance
            comment.save()

        return instance
```

First, note the `CommentSerializer` field has had its `read_only=True` argument removed.

Now onto the `update()` method. First we remove the `comments` from the `validated_data`. This is so that it's not passed to our parent `update()` method which would attempt to save the comments – we want to handle the comment creation ourselves.

Then, we call the super `update()` method which updates the `instance` (our `Post` object).

Then we iterate over the list of `comments` - each item in the list is a dictionary. If it has an `id` we assume the `Comment` already exists. If we wanted to be more thorough we could query the database for a comment with the given `id` and then possibly raise an exception if it didn't exist, but that's not necessary.

If there's no `id` then it must be a new comment. We create the `Comment` instance with the `comment_data` dictionary.

We set the `creator` to the current user of the request. Note here we've introduced something new: `self.context["request"]` holds the current request that the serializer is being used with. Not that this wouldn't be set if you were to use the serializer outside of the context of a request, but since we're not doing that we don't have to worry.

Next we set the `content_object` of the `Comment` to the current `Post` (`instance`), then save it.

Finally, we return the updated `instance`.

You can see that it’s a little of extra code to make a nested related field writeable, but it’s not too complicated. You can implement the `update()` (or `create()`) methods to suit whatever your model needs. They might end up being a bit more complex, for example when dealing with many-to-many fields, but it will come down to your individual application. We’ll finish the module with one last tweak: having tags created automatically on save.

Automatically Creating Related Objects

Automatically Creating Related Objects

One slight drawback of our tags field is that users can’t set a tag on a Post unless it already exists. We can create a serializer field subclass that does this for us. In our case, since we’re using SlugRelatedField we’ll subclass that to create our new field. Then, we’ll override its to_internal_value() method. This method takes the value that was provided in the API, then converts it into a value to save on the model (i.e. from a string to a Tag instance).

Here’s what such a field class would look like:

```
class TagField(serializers.SlugRelatedField):
    def to_internal_value(self, data):
        try:
            return
        self.get_queryset().get_or_create(value=data.lower())[0]
        except (TypeError, ValueError):
            self.fail(f"Tag value {data} is invalid")
```

▼ **Get or Create**

Remember that get_or_create() will fetch an instance from the database given the search parameters, or create one if it doesn’t exists. It returns a 2-element tuple (object, created), where object is the Tag (which might have just been created) and created is True if the object was created or False if not. We just want the object so we return the first element from the tuple.

The fail() method is just a shortcut method that DRF provides, to raise a ValidationError.

Then to use it on our PostSerializer, we’ll just update the tags field:

```python
class PostSerializer(serializers.ModelSerializer):
    tags = TagField(
        slug_field="value", many=True,
        queryset=Tag.objects.all()
    )
    # other methods/attributes omitted
```

This will allow new tags to be created on the fly, since we don't have an API with which to create them. DRF is still able to automatically handle the many-to-many relationships for us.

Try It Out

Try It Out

Let’s enhance your version of Django with all these new features: the ability to see and add comments, and create tags, through the API.

Start in `blog/api/serializers.py`. At the start of the file, you’ll need to import the `Comment` model:

```
from blog.models import Post, Tag, Comment
```

Then define the `TagField` underneath.

```
class TagField(serializers.SlugRelatedField):
    def to_internal_value(self, data):
        try:
            return self.get_queryset().get_or_create(value=data.lower())[0]
        except (TypeError, ValueError):
            self.fail(f"Tag value {data} is invalid")
```

Next we need to add the `CommentSerializer`:

```
class CommentSerializer(serializers.ModelSerializer):
    id = serializers.IntegerField(required=False)
    creator = UserSerializer(read_only=True)

    class Meta:
        model = Comment
        fields = ["id", "creator", "content", "modified_at", "created_at"]
        readonly = ["modified_at", "created_at"]
```

Note that since the serializers have a dependency their ordering in the file is important. `UserSerializer` should come first, followed by `CommentSerializer` then `PostSerializer`. The required order does not apply to the `TagField` serializer; it can go anywhere. Finally, at the end of the file, create the `PostDetailSerializer` class:

```
class PostDetailSerializer(PostSerializer):
    comments = CommentSerializer(many=True)

    def update(self, instance, validated_data):
        comments = validated_data.pop("comments")

        instance = super(PostDetailSerializer,
        self).update(instance, validated_data)

        for comment_data in comments:
            if comment_data.get("id"):
                # comment has an ID so was pre-existing
                continue
            comment = Comment(**comment_data)
            comment.creator = self.context["request"].user
            comment.content_object = instance
            comment.save()

        return instance
```

Switch to `blog/api/views.py`. We need to make two changes here. First, import the `PostDetailSerializer` class:

Open api/views.py

```
from blog.api.serializers import PostSerializer, UserSerializer,
        PostDetailSerializer
```

Then update the `PostDetail` class to use it:

```
class PostDetail(generics.RetrieveUpdateDestroyAPIView):
    permission_classes = [AuthorModifyOrReadOnly |
        IsAdminUserForObject]
    queryset = Post.objects.all()
    serializer_class = PostDetailSerializer
```

Note that you don't need to update the `PostList` class.

Now you can try out all these changes with Postman. If you `POST` or `PUT` a new `Post`, you can set `tags` that don't exist and they will be created. And, you should be able to `PUT` to create comments. The comments just need `content` field and the rest will be populated for you.

That's all for nested relationships and, indeed, this module. In the next module we'll start by looking at how to make customizations to the DRF GUI and how to simplify our code even further.

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 nested relationships"
```

- Push to GitHub:

```
git push
```