

source: relations.py

Serializer relations

Bad programmers worry about the code. Good programmers worry about data structures and their relationships.

— [Linus Torvalds](#)

Relational fields are used to represent model relationships. They can be applied to `ForeignKey`, `ManyToManyField` and `OneToOneField` relationships, as well as to reverse relationships, and custom relationships such as `GenericForeignKey`.

Note: The relational fields are declared in `relations.py`, but by convention you should import them from the `serializers` module, using `from rest_framework import serializers` and refer to fields as `serializers.<FieldName>`.

Inspecting automatically generated relationships.

When using the `ModelSerializer` class, serializer fields and relationships will be automatically generated for you. Inspecting these automatically generated fields can be a useful tool for determining how to customize the relationship style.

To do so, open the Django shell, using `python manage.py shell`, then import the serializer class, instantiate it, and print the object representation...

```
>>> from myapp.serializers import AccountSerializer
>>> serializer = AccountSerializer()
>>> print repr(serializer) # Or `print(repr(serializer))` in Python 3.x.
AccountSerializer():
  id = IntegerField(label='ID', read_only=True)
  name = CharField(allow_blank=True, max_length=100, required=False)
  owner = PrimaryKeyRelatedField(queryset=User.objects.all())
```

API Reference

In order to explain the various types of relational fields, we'll use a couple of simple models for our examples. Our models will be for music albums, and the tracks listed on each album.

```
class Album(models.Model):
```

```

album_name = models.CharField(max_length=100)
artist = models.CharField(max_length=100)

class Track(models.Model):
    album = models.ForeignKey(Album, related_name='tracks')
    order = models.IntegerField()
    title = models.CharField(max_length=100)
    duration = models.IntegerField()

    class Meta:
        unique_together = ('album', 'order')
        ordering = ['order']

    def __unicode__(self):
        return '%d: %s' % (self.order, self.title)

```

StringRelatedField

`StringRelatedField` may be used to represent the target of the relationship using its `__unicode__` method.

For example, the following serializer.

```

class AlbumSerializer(serializers.ModelSerializer):
    tracks = serializers.StringRelatedField(many=True)

    class Meta:
        model = Album
        fields = ('album_name', 'artist', 'tracks')

```

Would serialize to the following representation.

```

{
    'album_name': 'Things We Lost In The Fire',
    'artist': 'Low',
    'tracks': [
        '1: Sunflower',
        '2: Whitetail',
        '3: Dinosaur Act',
        ...
    ]
}

```

This field is read only.

Arguments:

- `many` - If applied to a to-many relationship, you should set this argument to `True`.

PrimaryKeyRelatedField

`PrimaryKeyRelatedField` may be used to represent the target of the relationship using its primary key.

For example, the following serializer:

```
class AlbumSerializer(serializers.ModelSerializer):
    tracks = serializers.PrimaryKeyRelatedField(many=True, read_only=True)

    class Meta:
        model = Album
        fields = ('album_name', 'artist', 'tracks')
```

Would serialize to a representation like this:

```
{
  'album_name': 'The Roots',
  'artist': 'Undun',
  'tracks': [
    89,
    90,
    91,
    ...
  ]
}
```

By default this field is read-write, although you can change this behavior using the `read_only` flag.

Arguments:

- `queryset` - The queryset used for model instance lookups when validating the field input. Relationships must either set a queryset explicitly, or set `read_only=True`.
- `many` - If applied to a to-many relationship, you should set this argument to `True`.
- `allow_null` - If set to `True`, the field will accept values of `None` or the empty string for nullable relationships. Defaults to `False`.
- `pk_field` - Set to a field to control serialization/deserialization of the primary key's value. For example, `pk_field=UUIDField(format='hex')` would serialize a UUID primary key into its compact hex representation.

HyperlinkedRelatedField

`HyperlinkedRelatedField` may be used to represent the target of the relationship using a hyperlink.

For example, the following serializer:

```
class AlbumSerializer(serializers.ModelSerializer):
```

```

tracks = serializers.HyperlinkedRelatedField(
    many=True,
    read_only=True,
    view_name='track-detail'
)

class Meta:
    model = Album
    fields = ('album_name', 'artist', 'tracks')

```

Would serialize to a representation like this:

```

{
    'album_name': 'Graceland',
    'artist': 'Paul Simon',
    'tracks': [
        'http://www.example.com/api/tracks/45/',
        'http://www.example.com/api/tracks/46/',
        'http://www.example.com/api/tracks/47/',
        ...
    ]
}

```

By default this field is read-write, although you can change this behavior using the `read_only` flag.

Arguments:

- `view_name` - The view name that should be used as the target of the relationship. If you're using [the standard router classes](#) this will be a string with the format `<modelname>-detail`. **required**.
- `queryset` - The queryset used for model instance lookups when validating the field input. Relationships must either set a queryset explicitly, or set `read_only=True`.
- `many` - If applied to a to-many relationship, you should set this argument to `True`.
- `allow_null` - If set to `True`, the field will accept values of `None` or the empty string for nullable relationships. Defaults to `False`.
- `lookup_field` - The field on the target that should be used for the lookup. Should correspond to a URL keyword argument on the referenced view. Default is `'pk'`.
- `lookup_url_kwarg` - The name of the keyword argument defined in the URL conf that corresponds to the lookup field. Defaults to using the same value as `lookup_field`.
- `format` - If using format suffixes, hyperlinked fields will use the same format suffix for the target unless overridden by using the `format` argument.

SlugRelatedField

`SlugRelatedField` may be used to represent the target of the relationship using a field on the target.

For example, the following serializer:

```
class AlbumSerializer(serializers.ModelSerializer):
    tracks = serializers.SlugRelatedField(
        many=True,
        read_only=True,
        slug_field='title'
    )

    class Meta:
        model = Album
        fields = ('album_name', 'artist', 'tracks')
```

Would serialize to a representation like this:

```
{
  'album_name': 'Dear John',
  'artist': 'Loney Dear',
  'tracks': [
    'Airport Surroundings',
    'Everything Turns to You',
    'I Was Only Going Out',
    ...
  ]
}
```

By default this field is read-write, although you can change this behavior using the `read_only` flag.

When using `SlugRelatedField` as a read-write field, you will normally want to ensure that the slug field corresponds to a model field with `unique=True`.

Arguments:

- `slug_field` - The field on the target that should be used to represent it. This should be a field that uniquely identifies any given instance. For example, `username`. **required**
- `queryset` - The queryset used for model instance lookups when validating the field input. Relationships must either set a queryset explicitly, or set `read_only=True`.
- `many` - If applied to a to-many relationship, you should set this argument to `True`.
- `allow_null` - If set to `True`, the field will accept values of `None` or the empty string for nullable relationships. Defaults to `False`.

HyperlinkedIdentityField

This field can be applied as an identity relationship, such as the `'url'` field on a `HyperlinkedModelSerializer`. It can also be used for an attribute on the object. For example, the following serializer:

```
class AlbumSerializer(serializers.HyperlinkedModelSerializer):
    track_listing = serializers.HyperlinkedIdentityField(view_name='track-list')
```

```
class Meta:
    model = Album
    fields = ('album_name', 'artist', 'track_listing')
```

Would serialize to a representation like this:

```
{
    'album_name': 'The Eraser',
    'artist': 'Thom Yorke',
    'track_listing': 'http://www.example.com/api/track_list/12/',
}
```

This field is always read-only.

Arguments:

- `view_name` - The view name that should be used as the target of the relationship. If you're using [the standard router classes](#) this will be a string with the format `<model_name>-detail`. **required.**
- `lookup_field` - The field on the target that should be used for the lookup. Should correspond to a URL keyword argument on the referenced view. Default is `'pk'`.
- `lookup_url_kwarg` - The name of the keyword argument defined in the URL conf that corresponds to the lookup field. Defaults to using the same value as `lookup_field`.
- `format` - If using format suffixes, hyperlinked fields will use the same format suffix for the target unless overridden by using the `format` argument.

Nested relationships

Nested relationships can be expressed by using serializers as fields.

If the field is used to represent a to-many relationship, you should add the `many=True` flag to the serializer field.

Example

For example, the following serializer:

```
class TrackSerializer(serializers.ModelSerializer):
    class Meta:
        model = Track
        fields = ('order', 'title')

class AlbumSerializer(serializers.ModelSerializer):
    tracks = TrackSerializer(many=True, read_only=True)

    class Meta:
```

```
model = Album
fields = ('album_name', 'artist', 'tracks')
```

Would serialize to a nested representation like this:

```
>>> album = Album.objects.create(album_name="The Grey Album", artist='Danger Mouse')
>>> Track.objects.create(album=album, order=1, title='Public Service Announcement',
duration=245)
<Track: Track object>
>>> Track.objects.create(album=album, order=2, title='What More Can I Say',
duration=264)
<Track: Track object>
>>> Track.objects.create(album=album, order=3, title='Encore', duration=159)
<Track: Track object>
>>> serializer = AlbumSerializer(instance=album)
>>> serializer.data
{
  'album_name': 'The Grey Album',
  'artist': 'Danger Mouse',
  'tracks': [
    {'order': 1, 'title': 'Public Service Announcement', 'duration': 245},
    {'order': 2, 'title': 'What More Can I Say', 'duration': 264},
    {'order': 3, 'title': 'Encore', 'duration': 159},
    ...
  ],
}
```

Writable nested serializers

By default nested serializers are read-only. If you want to support write-operations to a nested serializer field you'll need to create either or both of the `create()` and/or `update()` methods, in order to explicitly specify how the child relationships should be saved.

```
class TrackSerializer(serializers.ModelSerializer):
    class Meta:
        model = Track
        fields = ('order', 'title')

class AlbumSerializer(serializers.ModelSerializer):
    tracks = TrackSerializer(many=True)

    class Meta:
        model = Album
        fields = ('album_name', 'artist', 'tracks')

    def create(self, validated_data):
        tracks_data = validated_data.pop('tracks')
        album = Album.objects.create(**validated_data)
        for track_data in tracks_data:
            Track.objects.create(album=album, **track_data)
        return album
```

```
>>> data = {
    'album_name': 'The Grey Album',
    'artist': 'Danger Mouse',
    'tracks': [
        {'order': 1, 'title': 'Public Service Announcement', 'duration': 245},
        {'order': 2, 'title': 'What More Can I Say', 'duration': 264},
        {'order': 3, 'title': 'Encore', 'duration': 159},
    ],
}
>>> serializer = AlbumSerializer(data=data)
>>> serializer.is_valid()
True
>>> serializer.save()
<Album: Album object>
```

Custom relational fields

To implement a custom relational field, you should override `RelatedField`, and implement the `.to_representation(self, value)` method. This method takes the target of the field as the `value` argument, and should return the representation that should be used to serialize the target. The `value` argument will typically be a model instance.

If you want to implement a read-write relational field, you must also implement the `.to_internal_value(self, data)` method.

Example

For example, we could define a relational field to serialize a track to a custom string representation, using its ordering, title, and duration.

```
import time

class TrackListingField(serializers.RelatedField):
    def to_representation(self, value):
        duration = time.strftime('%M:%S', time.gmtime(value.duration))
        return 'Track %d: %s (%s)' % (value.order, value.name, duration)

class AlbumSerializer(serializers.ModelSerializer):
    tracks = TrackListingField(many=True)

    class Meta:
        model = Album
        fields = ('album_name', 'artist', 'tracks')
```

This custom field would then serialize to the following representation.

```
{
    'album_name': 'Sometimes I Wish We Were an Eagle',
    'artist': 'Bill Callahan',
```



```
'tracks': [
    'Track 1: Jim Cain (04:39)',
    'Track 2: Eid Ma Clack Shaw (04:19)',
    'Track 3: The Wind and the Dove (04:34)',
    ...
]
```

Further notes

The `queryset` argument

The `queryset` argument is only ever required for *writable* relationship field, in which case it is used for performing the model instance lookup, that maps from the primitive user input, into a model instance.

In version 2.x a serializer class could *sometimes* automatically determine the `queryset` argument *if* a `ModelSerializer` class was being used.

This behavior is now replaced with *always* using an explicit `queryset` argument for writable relational fields.

Doing so reduces the amount of hidden 'magic' that `ModelSerializer` provides, makes the behavior of the field more clear, and ensures that it is trivial to move between using the `ModelSerializer` shortcut, or using fully explicit `Serializer` classes.

Reverse relations

Note that reverse relationships are not automatically included by the `ModelSerializer` and `HyperlinkedModelSerializer` classes. To include a reverse relationship, you must explicitly add it to the fields list. For example:

```
class AlbumSerializer(serializers.ModelSerializer):
    class Meta:
        fields = ('tracks', ...)
```

You'll normally want to ensure that you've set an appropriate `related_name` argument on the relationship, that you can use as the field name. For example:

```
class Track(models.Model):
    album = models.ForeignKey(Album, related_name='tracks')
    ...
```

If you have not set a related name for the reverse relationship, you'll need to use the automatically generated related name in the `fields` argument. For example:

```
class AlbumSerializer(serializers.ModelSerializer):
    class Meta:
        fields = ('track_set', ...)
```

See the Django documentation on [reverse relationships](#) for more details.

Generic relationships

If you want to serialize a generic foreign key, you need to define a custom field, to determine explicitly how you want to serialize the targets of the relationship.

For example, given the following model for a tag, which has a generic relationship with other arbitrary models:

```
class TaggedItem(models.Model):
    """
    Tags arbitrary model instances using a generic relation.

    See: https://docs.djangoproject.com/en/dev/ref/contrib/contenttypes/
    """
    tag_name = models.SlugField()
    content_type = models.ForeignKey(ContentType)
    object_id = models.PositiveIntegerField()
    tagged_object = GenericForeignKey('content_type', 'object_id')

    def __unicode__(self):
        return self.tag
```

And the following two models, which may be have associated tags:

```
class Bookmark(models.Model):
    """
    A bookmark consists of a URL, and 0 or more descriptive tags.
    """
    url = models.URLField()
    tags = GenericRelation(TaggedItem)

class Note(models.Model):
    """
    A note consists of some text, and 0 or more descriptive tags.
    """
    text = models.CharField(max_length=1000)
    tags = GenericRelation(TaggedItem)
```

We could define a custom field that could be used to serialize tagged instances, using the type of

each instance to determine how it should be serialized.

```
class TaggedObjectRelatedField(serializers.RelatedField):
    """
    A custom field to use for the `tagged_object` generic relationship.
    """

    def to_representation(self, value):
        """
        Serialize tagged objects to a simple textual representation.
        """
        if isinstance(value, Bookmark):
            return 'Bookmark: ' + value.url
        elif isinstance(value, Note):
            return 'Note: ' + value.text
        raise Exception('Unexpected type of tagged object')
```

If you need the target of the relationship to have a nested representation, you can use the required serializers inside the `.to_representation()` method:

```
def to_representation(self, value):
    """
    Serialize bookmark instances using a bookmark serializer,
    and note instances using a note serializer.
    """
    if isinstance(value, Bookmark):
        serializer = BookmarkSerializer(value)
    elif isinstance(value, Note):
        serializer = NoteSerializer(value)
    else:
        raise Exception('Unexpected type of tagged object')

    return serializer.data
```

Note that reverse generic keys, expressed using the `GenericRelation` field, can be serialized using the regular relational field types, since the type of the target in the relationship is always known.

For more information see [the Django documentation on generic relations](#).

ManyToManyFields with a Through Model

By default, relational fields that target a `ManyToManyField` with a `through` model specified are set to read-only.

If you explicitly specify a relational field pointing to a `ManyToManyField` with a `through` model, be sure to set `read_only` to `True`.

Advanced Hyperlinked fields

If you have very specific requirements for the style of your hyperlinked relationships you can override `HyperlinkedRelatedField`.

There are two methods you'll need to override.

`get_url(self, obj, view_name, request, format)`

This method should return the URL that corresponds to the given object.

May raise a `NoReverseMatch` if the `view_name` and `lookup_field` attributes are not configured to correctly match the URL conf.

`get_object(self, queryset, view_name, view_args, view_kwargs)`

This method should the object that corresponds to the matched URL conf arguments.

May raise an `ObjectDoesNotExist` exception.

Example

For example, if all your object URLs used both a account and a slug in the the URL to reference the object, you might create a custom field like this:

```
class CustomHyperlinkedField(serializers.HyperlinkedRelatedField):
    def get_url(self, obj, view_name, request, format):
        kwargs = {'account': obj.account, 'slug': obj.slug}
        return reverse(view_name, kwargs=kwargs, request=request, format=format)

    def get_object(self, queryset, view_name, view_args, view_kwargs):
        account = view_kwargs['account']
        slug = view_kwargs['slug']
        return queryset.get(account=account, slug=slug)
```

Third Party Packages

The following third party packages are also available.

DRF Nested Routers

The [drf-nested-routers package](#) provides routers and relationship fields for working with nested resources.