validators.py

Validators

> *Validators can be useful for re-using validation logic between different types of fields.*
>
> — *Django documentation*

Most of the time you're dealing with validation in REST framework you'll simply be relying on the default field validation, or writing explicit validation methods on serializer or field classes.

However, sometimes you'll want to place your validation logic into reusable components, so that it can easily be reused throughout your codebase. This can be achieved by using validator functions and validator classes.

Validation in REST framework

Validation in Django REST framework serializers is handled a little differently to how validation works in Django's `ModelForm` class.

With `ModelForm` the validation is performed partially on the form, and partially on the model instance. With REST framework the validation is performed entirely on the serializer class. This is advantageous for the following reasons:

- It introduces a proper separation of concerns, making your code behavior more obvious.
- It is easy to switch between using shortcut `ModelSerializer` classes and using explicit `Serializer` classes. Any validation behavior being used for `ModelSerializer` is simple to replicate.

- Printing the `repr` of a serializer instance will show you exactly what validation rules it applies. There's no extra hidden validation behavior being called on the model instance.

When you're using `ModelSerializer` all of this is handled automatically for you. If you want to drop down to using `Serializer` classes instead, then you need to define the validation rules explicitly.

Example

As an example of how REST framework uses explicit validation, we'll take a simple model class that has a field with a uniqueness constraint.

```
class CustomerReportRecord(models.Model):
    time_raised = models.DateTimeField(default=timezone.now, editable=False)
    reference = models.CharField(unique=True, max_length=20)
    description = models.TextField()
```

Here's a basic `ModelSerializer` that we can use for creating or updating instances of `CustomerReportRecord`:

```
class CustomerReportSerializer(serializers.ModelSerializer):
    class Meta:
        model = CustomerReportRecord
```

If we open up the Django shell using `manage.py shell` we can now

```
>>> from project.example.serializers import CustomerReportSerializer
>>> serializer = CustomerReportSerializer()
>>> print(repr(serializer))
CustomerReportSerializer():
    id = IntegerField(label='ID', read_only=True)
    time_raised = DateTimeField(read_only=True)
    reference = CharField(max_length=20, validators=[<UniqueValidator(queryset=CustomerRepo
    description = CharField(style={'type': 'textarea'})
```

The interesting bit here is the `reference` field. We can see that the uniqueness constraint is being explicitly enforced by a validator on the serializer field.

Because of this more explicit style REST framework includes a few validator classes that are not available in core Django. These classes are detailed below.

UniqueValidator

This validator can be used to enforce the `unique=True` constraint on model fields. It takes a single required argument, and an optional `messages` argument:

- `queryset` *required* - This is the queryset against which uniqueness should be enforced.

- `message` - The error message that should be used when validation fails.
- `lookup` - The lookup used to find an existing instance with the value being validated. Defaults to `'exact'`.

This validator should be applied to *serializer fields*, like so:

```
from rest_framework.validators import UniqueValidator

slug = SlugField(
    max_length=100,
    validators=[UniqueValidator(queryset=BlogPost.objects.all())]
)
```

UniqueTogetherValidator

This validator can be used to enforce `unique_together` constraints on model instances. It has two required arguments, and a single optional `messages` argument:

- `queryset` *required* - This is the queryset against which uniqueness should be enforced.
- `fields` *required* - A list or tuple of field names which should make a unique set. These must exist as fields on the serializer class.
- `message` - The error message that should be used when validation fails.

The validator should be applied to *serializer classes*, like so:

```
from rest_framework.validators import UniqueTogetherValidator

class ExampleSerializer(serializers.Serializer):
    # ...
    class Meta:
        # ToDo items belong to a parent list, and have an ordering defined
        # by the 'position' field. No two items in a given list may share
        # the same position.
        validators = [
            UniqueTogetherValidator(
                queryset=ToDoItem.objects.all(),
                fields=('list', 'position')
            )
        ]
```

Note: The `UniqueTogetherValidation` class always imposes an implicit constraint that all the fields it applies to are always treated as required. Fields with `default` values are an exception to this as they always supply a value even when omitted from user input.

UniqueForDateValidator

UniqueForMonthValidator

UniqueForYearValidator

These validators can be used to enforce the `unique_for_date`, `unique_for_month` and `unique_for_year` constraints on model instances. They take the following arguments:

- `queryset` *required* - This is the queryset against which uniqueness should be enforced.
- `field` *required* - A field name against which uniqueness in the given date range will be validated. This must exist as a field on the serializer class.
- `date_field` *required* - A field name which will be used to determine date range for the uniqueness constrain. This must exist as a field on the serializer class.
- `message` - The error message that should be used when validation fails.

The validator should be applied to *serializer classes*, like so:

```
from rest_framework.validators import UniqueForYearValidator

class ExampleSerializer(serializers.Serializer):
    # ...
    class Meta:
        # Blog posts should have a slug that is unique for the current year.
        validators = [
            UniqueForYearValidator(
                queryset=BlogPostItem.objects.all(),
                field='slug',
                date_field='published'
            )
        ]
```

The date field that is used for the validation is always required to be present on the serializer class. You can't simply rely on a model class `default=...`, because the value being used for the default wouldn't be generated until after the validation has run.

There are a couple of styles you may want to use for this depending on how you want your API to behave. If you're using `ModelSerializer` you'll probably simply rely on the defaults that REST framework generates for you, but if you are using `Serializer` or simply want more explicit control, use on of the styles demonstrated below.

Using with a writable date field.

If you want the date field to be writable the only thing worth noting is that you should ensure that it is always available in the input data, either by setting a `default` argument, or by setting `required=True`.

```
published = serializers.DateTimeField(required=True)
```

Using with a read-only date field.

If you want the date field to be visible, but not editable by the user, then set `read_only=True` and additionally set a `default=...` argument.

```
published = serializers.DateTimeField(read_only=True, default=timezone.now)
```

The field will not be writable to the user, but the default value will still be passed through to the `validated_data`.

Using with a hidden date field.

If you want the date field to be entirely hidden from the user, then use `HiddenField`. This field type does not accept user input, but instead always returns its default value to the `validated_data` in the serializer.

```
published = serializers.HiddenField(default=timezone.now)
```

Note: The `UniqueFor<Range>Validation` classes impose an implicit constraint that the fields they are applied to are always treated as required. Fields with `default` values are an exception to this as they always supply a value even when omitted from user input.

Advanced field defaults

Validators that are applied across multiple fields in the serializer can sometimes require a field input that should not be provided by the API client, but that *is* available as input to the validator.

Two patterns that you may want to use for this sort of validation include:

- Using `HiddenField`. This field will be present in `validated_data` but *will not* be used in the serializer output representation.
- Using a standard field with `read_only=True`, but that also includes a `default=…` argument. This field *will* be used in the serializer output representation, but cannot be set directly by the user.

REST framework includes a couple of defaults that may be useful in this context.

CurrentUserDefault

A default class that can be used to represent the current user. In order to use this, the 'request' must have been provided as part of the context dictionary when instantiating the serializer.

```
owner = serializers.HiddenField(
    default=serializers.CurrentUserDefault()
)
```

CreateOnlyDefault

A default class that can be used to *only set a default argument during create operations*. During updates the field is omitted.

It takes a single argument, which is the default value or callable that should be used during create operations.

```
created_at = serializers.DateTimeField(
    read_only=True,
    default=serializers.CreateOnlyDefault(timezone.now)
)
```

Limitations of validators

There are some ambiguous cases where you'll need to instead handle validation explicitly, rather than relying on the default serializer classes that `ModelSerializer` generates.

In these cases you may want to disable the automatically generated validators, by specifying an empty list for the serializer `Meta.validators` attribute.

Optional fields

By default "unique together" validation enforces that all fields be `required=True`. In some cases, you might want to explicit apply `required=False` to one of the fields, in which case the desired behaviour of the validation is ambiguous.

In this case you will typically need to exclude the validator from the serializer class, and instead write any validation logic explicitly, either in the `.validate()` method, or else in the view.

For example:

```
class BillingRecordSerializer(serializers.ModelSerializer):
    def validate(self, data):
        # Apply custom validation either here, or in the view.

    class Meta:
        fields = ('client', 'date', 'amount')
        extra_kwargs = {'client': {'required': 'False'}}
        validators = []  # Remove a default "unique together" constraint.
```

Updating nested serializers

When applying an update to an existing instance, uniqueness validators will exclude the current instance from the uniqueness check. The current instance is available in the context of the uniqueness check, because it exists as an attribute on the serializer, having initially been passed using `instance=...` when instantiating the serializer.

In the case of update operations on *nested* serializers there's no way of applying this exclusion, because the instance is not available.

Again, you'll probably want to explicitly remove the validator from the serializer class, and write the code the for the validation constraint explicitly, in a `.validate()` method, or in the view.

Debugging complex cases

If you're not sure exactly what behavior a `ModelSerializer` class will generate it is usually a good idea to run `manage.py shell`, and print an instance of the serializer, so that you can inspect the fields and validators that it automatically generates for you.

```
>>> serializer = MyComplexModelSerializer()
>>> print(serializer)
class MyComplexModelSerializer:
    my_fields = ...
```

Also keep in mind that with complex cases it can often be better to explicitly define your serializer classes, rather than relying on the default `ModelSerializer` behavior. This involves a little more code, but ensures that the resulting behavior is more transparent.

Writing custom validators

You can use any of Django's existing validators, or write your own custom validators.

Function based

A validator may be any callable that raises a `serializers.ValidationError` on failure.

```
def even_number(value):
    if value % 2 != 0:
        raise serializers.ValidationError('This field must be an even number.')
```

Field-level validation

You can specify custom field-level validation by adding `.validate_<field_name>` methods to your `Serializer` subclass. This is documented in the Serializer docs

Class-based

To write a class-based validator, use the `__call__` method. Class-based validators are useful as they allow you to parameterize and reuse behavior.

```
class MultipleOf(object):
    def __init__(self, base):
        self.base = base

    def __call__(self, value):
        if value % self.base != 0:
            message = 'This field must be a multiple of %d.' % self.base
            raise serializers.ValidationError(message)
```

Using `set_context()`

In some advanced cases you might want a validator to be passed the serializer field it is being used with as additional context. You can do so by declaring a `set_context` method on a class-based validator.

```
def set_context(self, serializer_field):
    # Determine if this is an update or a create operation.
    # In `__call__` we can then use that information to modify the validation behavior.
    self.is_update = serializer_field.parent.instance is not None
```

Documentation built with **MkDocs**.