source: fields.py

Serializer fields

> Each field in a Form class is responsible not only for validating data, but also for "cleaning" it — normalizing it to a consistent format.
>
> — Django documentation

Serializer fields handle converting between primitive values and internal datatypes. They also deal with validating input values, as well as retrieving and setting the values from their parent objects.

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

Core arguments

Each serializer field class constructor takes at least these arguments. Some Field classes take additional, field-specific arguments, but the following should always be accepted:

`read_only`

Read-only fields are included in the API output, but should not be included in the input during create or update operations. Any 'read_only' fields that are incorrectly included in the serializer input will be ignored.

Set this to `True` to ensure that the field is used when serializing a representation, but is not used when creating or updating an instance during deserialization.

Defaults to `False`

`write_only`

Set this to `True` to ensure that the field may be used when updating or creating an instance, but is not included when serializing the representation.

Defaults to `False`

`required`

Normally an error will be raised if a field is not supplied during deserialization. Set to false if this field is not required to be present during deserialization.

Setting this to `False` also allows the object attribute or dictionary key to be omitted from output when serializing the instance. If the key is not present it will simply not be included in the output representation.

Defaults to `True`.

`allow_null`

Normally an error will be raised if `None` is passed to a serializer field. Set this keyword argument to `True` if `None` should be considered a valid value.

Defaults to `False`

`default`

If set, this gives the default value that will be used for the field if no input value is supplied. If not set the default behavior is to not populate the attribute at all.

May be set to a function or other callable, in which case the value will be evaluated each time it is used.

Note that setting a `default` value implies that the field is not required. Including both the `default` and `required` keyword arguments is invalid and will raise an error.

`source`

The name of the attribute that will be used to populate the field. May be a method that only takes a `self` argument, such as `URLField('get_absolute_url')`, or may use dotted notation to traverse attributes, such as `EmailField(source='user.email')`.

The value `source='*'` has a special meaning, and is used to indicate that the entire object should be passed through to the field. This can be useful for creating nested representations, or for fields which require access to the complete object in order to determine the output representation.

Defaults to the name of the field.

`validators`

A list of validator functions which should be applied to the incoming field input, and which either raise a validation error or simply return. Validator functions should typically raise `serializers.ValidationError`, but Django's built-in `ValidationError` is also supported for compatibility with validators defined in the Django codebase or third party Django packages.

`error_messages`

A dictionary of error codes to error messages.

`label`

A short text string that may be used as the name of the field in HTML form fields or other

descriptive elements.

help_text

A text string that may be used as a description of the field in HTML form fields or other descriptive elements.

initial

A value that should be used for pre-populating the value of HTML form fields.

style

A dictionary of key-value pairs that can be used to control how renderers should render the field. The API for this should still be considered experimental, and will be formalized with the 3.1 release.

Two options are currently used in HTML form generation, `'input_type'` and `'base_template'`.

```
# Use <input type="password"> for the input.
password = serializers.CharField(
    style={'input_type': 'password'}
)

# Use a radio input instead of a select input.
color_channel = serializers.ChoiceField(
    choices=['red', 'green', 'blue']
    style = {'base_template': 'radio.html'}
}
```

Note: The `style` argument replaces the old-style version 2.x `widget` keyword argument. Because REST framework 3 now uses templated HTML form generation, the `widget` option that was used to support Django built-in widgets can no longer be supported. Version 3.1 is planned to include public API support for customizing HTML form generation.

Boolean fields

BooleanField

A boolean representation.

When using HTML encoded form input be aware that omitting a value will always be treated as setting a field to `False`, even if it has a `default=True` option specified. This is because HTML checkbox inputs represent the unchecked state by omitting the value, so REST framework treats omission as if it is an empty checkbox input.

Corresponds to `django.db.models.fields.BooleanField`.

Signature: `BooleanField()`

NullBooleanField

A boolean representation that also accepts `None` as a valid value.

Corresponds to `django.db.models.fields.NullBooleanField`.

Signature: `NullBooleanField()`

String fields

CharField

A text representation. Optionally validates the text to be shorter than `max_length` and longer than `min_length`.

Corresponds to `django.db.models.fields.CharField` or `django.db.models.fields.TextField`.

Signature: `CharField(max_length=None, min_length=None, allow_blank=False, trim_whitespace=True)`

- `max_length` - Validates that the input contains no more than this number of characters.
- `min_length` - Validates that the input contains no fewer than this number of characters.
- `allow_blank` - If set to `True` then the empty string should be considered a valid value. If set to `False` then the empty string is considered invalid and will raise a validation error. Defaults to `False`.
- `trim_whitespace` - If set to `True` then leading and trailing whitespace is trimmed. Defaults to `True`.

The `allow_null` option is also available for string fields, although its usage is discouraged in favor of `allow_blank`. It is valid to set both `allow_blank=True` and `allow_null=True`, but doing so means that there will be two differing types of empty value permissible for string representations, which can lead to data inconsistencies and subtle application bugs.

EmailField

A text representation, validates the text to be a valid e-mail address.

Corresponds to `django.db.models.fields.EmailField`

Signature: `EmailField(max_length=None, min_length=None, allow_blank=False)`

RegexField

A text representation, that validates the given value matches against a certain regular expression.

Corresponds to `django.forms.fields.RegexField`.

Signature: `RegexField(regex, max_length=None, min_length=None, allow_blank=False)`

The mandatory `regex` argument may either be a string, or a compiled python regular expression object.

Uses Django's `django.core.validators.RegexValidator` for validation.

SlugField

A `RegexField` that validates the input against the pattern `[a-zA-Z0-9_-]+`.

Corresponds to `django.db.models.fields.SlugField`.

Signature: `SlugField(max_length=50, min_length=None, allow_blank=False)`

URLField

A `RegexField` that validates the input against a URL matching pattern. Expects fully qualified URLs of the form `http://<host>/<path>`.

Corresponds to `django.db.models.fields.URLField`. Uses Django's `django.core.validators.URLValidator` for validation.

Signature: `URLField(max_length=200, min_length=None, allow_blank=False)`

UUIDField

A field that ensures the input is a valid UUID string. The `to_internal_value` method will return a `uuid.UUID` instance. On output the field will return a string in the canonical hyphenated format, for example:

```
"de305d54-75b4-431b-adb2-eb6b9e546013"
```

Signature: `UUIDField(format='hex_verbose')`

- `format`: Determines the representation format of the uuid value
  - `'hex_verbose'` - The cannoncical hex representation, including hyphens: `"5ce0e9a5-5ffa-654b-cee0-1238041fb31a"`
  - `'hex'` - The compact hex representation of the UUID, not including hyphens: `"5ce0e9a55ffa654bcee01238041fb31a"`

- `'int'` - A 128 bit integer representation of the UUID: `"123456789012312313134124512351145145114"`
- `'urn'` - RFC 4122 URN representation of the UUID: `"urn:uuid:5ce0e9a5-5ffa-654b-cee0-1238041fb31a"` Changing the `format` parameters only affects representation values. All formats are accepted by `to_internal_value`

IPAddressField

A field that ensures the input is a valid IPv4 or IPv6 string.

Corresponds to `django.forms.fields.IPAddressField` and `django.forms.fields.GenericIPAddressField`.

Signature: `IPAddressField(protocol='both', unpack_ipv4=False, **options)`

- `protocol` Limits valid inputs to the specified protocol. Accepted values are 'both' (default), 'IPv4' or 'IPv6'. Matching is case insensitive.
- `unpack_ipv4` Unpacks IPv4 mapped addresses like ::ffff:192.0.2.1. If this option is enabled that address would be unpacked to 192.0.2.1. Default is disabled. Can only be used when protocol is set to 'both'.

Numeric fields

IntegerField

An integer representation.

Corresponds to `django.db.models.fields.IntegerField`, `django.db.models.fields.SmallIntegerField`, `django.db.models.fields.PositiveIntegerField` and `django.db.models.fields.PositiveSmallIntegerField`.

Signature: `IntegerField(max_value=None, min_value=None)`

- `max_value` Validate that the number provided is no greater than this value.
- `min_value` Validate that the number provided is no less than this value.

FloatField

A floating point representation.

Corresponds to `django.db.models.fields.FloatField`.

Signature: `FloatField(max_value=None, min_value=None)`

- `max_value` Validate that the number provided is no greater than this value.
- `min_value` Validate that the number provided is no less than this value.

DecimalField

A decimal representation, represented in Python by a `Decimal` instance.

Corresponds to `django.db.models.fields.DecimalField`.

Signature: `DecimalField(max_digits, decimal_places, coerce_to_string=None, max_value=None, min_value=None)`

- `max_digits` The maximum number of digits allowed in the number. Note that this number must be greater than or equal to decimal_places.
- `decimal_places` The number of decimal places to store with the number.
- `coerce_to_string` Set to `True` if string values should be returned for the representation, or `False` if `Decimal` objects should be returned. Defaults to the same value as the `COERCE_DECIMAL_TO_STRING` settings key, which will be `True` unless overridden. If `Decimal` objects are returned by the serializer, then the final output format will be determined by the renderer.
- `max_value` Validate that the number provided is no greater than this value.
- `min_value` Validate that the number provided is no less than this value.

Example usage

To validate numbers up to 999 with a resolution of 2 decimal places, you would use:

```
serializers.DecimalField(max_digits=5, decimal_places=2)
```

And to validate numbers up to anything less than one billion with a resolution of 10 decimal places:

```
serializers.DecimalField(max_digits=19, decimal_places=10)
```

This field also takes an optional argument, `coerce_to_string`. If set to `True` the representation will be output as a string. If set to `False` the representation will be left as a `Decimal` instance and the final representation will be determined by the renderer.

If unset, this will default to the same value as the `COERCE_DECIMAL_TO_STRING` setting, which is `True` unless set otherwise.

Date and time fields

DateTimeField

A date and time representation.

Corresponds to `django.db.models.fields.DateTimeField`.

Signature: `DateTimeField(format=None, input_formats=None)`

- `format` - A string representing the output format. If not specified, this defaults to the same value as the `DATETIME_FORMAT` settings key, which will be `'iso-8601'` unless set. Setting to a format string indicates that `to_representation` return values should be coerced to string output. Format strings are described below. Setting this value to `None` indicates that Python `datetime` objects should be returned by `to_representation`. In this case the datetime encoding will be determined by the renderer.
- `input_formats` - A list of strings representing the input formats which may be used to parse the date. If not specified, the `DATETIME_INPUT_FORMATS` setting will be used, which defaults to `['iso-8601']`.

`DateTimeField` format strings.

Format strings may either be Python strftime formats which explicitly specify the format, or the special string `'iso-8601'`, which indicates that ISO 8601 style datetimes should be used. (eg `'2013-01-29T12:34:56.000000Z'`)

When a value of `None` is used for the format `datetime` objects will be returned by `to_representation` and the final output representation will determined by the renderer class.

In the case of JSON this means the default datetime representation uses the ECMA 262 date time string specification. This is a subset of ISO 8601 which uses millisecond precision, and includes the 'Z' suffix for the UTC timezone, for example: `2013-01-29T12:34:56.123Z`.

`auto_now` and `auto_now_add` model fields.

When using `ModelSerializer` or `HyperlinkedModelSerializer`, note that any model fields with `auto_now=True` or `auto_now_add=True` will use serializer fields that are `read_only=True` by default.

If you want to override this behavior, you'll need to declare the `DateTimeField` explicitly on the serializer. For example:

```
class CommentSerializer(serializers.ModelSerializer):
    created = serializers.DateTimeField()

    class Meta:
        model = Comment
```

DateField

A date representation.

Corresponds to `django.db.models.fields.DateField`

Signature: `DateField(format=None, input_formats=None)`

- `format` - A string representing the output format. If not specified, this defaults to the same value as the `DATE_FORMAT` settings key, which will be `'iso-8601'` unless set. Setting to a format string indicates that `to_representation` return values should be coerced to string output. Format strings are described below. Setting this value to `None` indicates that Python `date` objects should be returned by `to_representation`. In this case the date encoding will be determined by the renderer.
- `input_formats` - A list of strings representing the input formats which may be used to parse the date. If not specified, the `DATE_INPUT_FORMATS` setting will be used, which defaults to `['iso-8601']`.

`DateField` format strings

Format strings may either be Python strftime formats which explicitly specify the format, or the special string `'iso-8601'`, which indicates that ISO 8601 style dates should be used. (eg `'2013-01-29'`)

TimeField

A time representation.

Corresponds to `django.db.models.fields.TimeField`

Signature: `TimeField(format=None, input_formats=None)`

- `format` - A string representing the output format. If not specified, this defaults to the same value as the `TIME_FORMAT` settings key, which will be `'iso-8601'` unless set. Setting to a format string indicates that `to_representation` return values should be coerced to string output. Format strings are described below. Setting this value to `None` indicates that Python `time` objects should be returned by `to_representation`. In this case the time encoding will be determined by the renderer.
- `input_formats` - A list of strings representing the input formats which may be used to parse the date. If not specified, the `TIME_INPUT_FORMATS` setting will be used, which defaults to `['iso-8601']`.

`TimeField` format strings

Format strings may either be Python strftime formats which explicitly specify the format, or the special string `'iso-8601'`, which indicates that ISO 8601 style times should be used. (eg `'12:34:56.000000'`)

DurationField

A Duration representation. Corresponds to `django.db.models.fields.DurationField`

The `validated_data` for these fields will contain a `datetime.timedelta` instance. The representation is a string following this format `'[DD] [HH:[MM:]]ss[.uuuuuu]'`.

Note: This field is only available with Django versions >= 1.8.

Signature: `DurationField()`

Choice selection fields

ChoiceField

A field that can accept a value out of a limited set of choices.

Used by `ModelSerializer` to automatically generate fields if the corresponding model field includes a `choices=…` argument.

Signature: `ChoiceField(choices)`

- `choices` - A list of valid values, or a list of `(key, display_name)` tuples.
- `allow_blank` - If set to `True` then the empty string should be considered a valid value. If set to `False` then the empty string is considered invalid and will raise a validation error. Defaults to `False`.

Both the `allow_blank` and `allow_null` are valid options on `ChoiceField`, although it is highly recommended that you only use one and not both. `allow_blank` should be preferred for textual choices, and `allow_null` should be preferred for numeric or other non-textual choices.

MultipleChoiceField

A field that can accept a set of zero, one or many values, chosen from a limited set of choices. Takes a single mandatory argument. `to_internal_value` returns a `set` containing the selected values.

Signature: `MultipleChoiceField(choices)`

- `choices` - A list of valid values, or a list of `(key, display_name)` tuples.
- `allow_blank` - If set to `True` then the empty string should be considered a valid value. If set to `False` then the empty string is considered invalid and will raise a validation error. Defaults to `False`.

As with `ChoiceField`, both the `allow_blank` and `allow_null` options are valid, although it is highly

recommended that you only use one and not both. `allow_blank` should be preferred for textual choices, and `allow_null` should be preferred for numeric or other non-textual choices.

File upload fields

Parsers and file uploads.

The `FileField` and `ImageField` classes are only suitable for use with `MultiPartParser` or `FileUploadParser`. Most parsers, such as e.g. JSON don't support file uploads. Django's regular FILE_UPLOAD_HANDLERS are used for handling uploaded files.

FileField

A file representation. Performs Django's standard FileField validation.

Corresponds to `django.forms.fields.FileField`.

Signature: `FileField(max_length=None, allow_empty_file=False, use_url=UPLOADED_FILES_USE_URL)`

- `max_length` - Designates the maximum length for the file name.
- `allow_empty_file` - Designates if empty files are allowed.
  - `use_url` - If set to `True` then URL string values will be used for the output representation. If set to `False` then filename string values will be used for the output representation. Defaults to the value of the `UPLOADED_FILES_USE_URL` settings key, which is `True` unless set otherwise.

ImageField

An image representation. Validates the uploaded file content as matching a known image format.

Corresponds to `django.forms.fields.ImageField`.

Signature: `ImageField(max_length=None, allow_empty_file=False, use_url=UPLOADED_FILES_USE_URL)`

- `max_length` - Designates the maximum length for the file name.
- `allow_empty_file` - Designates if empty files are allowed.
  - `use_url` - If set to `True` then URL string values will be used for the output representation. If set to `False` then filename string values will be used for the output representation. Defaults to the value of the `UPLOADED_FILES_USE_URL` settings key, which is `True` unless set otherwise.

Requires either the `Pillow` package or `PIL` package. The `Pillow` package is recommended, as `PIL` is no longer actively maintained.

Composite fields

ListField

A field class that validates a list of objects.

Signature: `ListField(child)`

- `child` - A field instance that should be used for validating the objects in the list. If this argument is not provided then objects in the list will not be validated.

For example, to validate a list of integers you might use something like the following:

```
scores = serializers.ListField(
   child=serializers.IntegerField(min_value=0, max_value=100)
)
```

The `ListField` class also supports a declarative style that allows you to write reusable list field classes.

```
class StringListField(serializers.ListField):
    child = serializers.CharField()
```

We can now reuse our custom `StringListField` class throughout our application, without having to provide a `child` argument to it.

DictField

A field class that validates a dictionary of objects. The keys in `DictField` are always assumed to be string values.

Signature: `DictField(child)`

- `child` - A field instance that should be used for validating the values in the dictionary. If this argument is not provided then values in the mapping will not be validated.

For example, to create a field that validates a mapping of strings to strings, you would write something like this:

```
document = DictField(child=CharField())
```

You can also use the declarative style, as with `ListField`. For example:

```
class DocumentField(DictField):
    child = CharField()
```

Miscellaneous fields

ReadOnlyField

A field class that simply returns the value of the field without modification.

This field is used by default with `ModelSerializer` when including field names that relate to an attribute rather than a model field.

Signature: `ReadOnlyField()`

For example, is `has_expired` was a property on the `Account` model, then the following serializer would automatically generate it as a `ReadOnlyField`:

```
class AccountSerializer(serializers.ModelSerializer):
    class Meta:
        model = Account
        fields = ('id', 'account_name', 'has_expired')
```

HiddenField

A field class that does not take a value based on user input, but instead takes its value from a default value or callable.

Signature: `HiddenField()`

For example, to include a field that always provides the current time as part of the serializer validated data, you would use the following:

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

The `HiddenField` class is usually only needed if you have some validation that needs to run based on some pre-provided field values, but you do not want to expose all of those fields to the end user.

For further examples on `HiddenField` see the validators documentation.

ModelField

A generic field that can be tied to any arbitrary model field. The `ModelField` class delegates the task

of serialization/deserialization to its associated model field. This field can be used to create serializer fields for custom model fields, without having to create a new custom serializer field.

This field is used by `ModelSerializer` to correspond to custom model field classes.

Signature: `ModelField(model_field=<Django ModelField instance>)`

The `ModelField` class is generally intended for internal use, but can be used by your API if needed. In order to properly instantiate a `ModelField`, it must be passed a field that is attached to an instantiated model. For example:
`ModelField(model_field=MyModel()._meta.get_field('custom_field'))`

SerializerMethodField

This is a read-only field. It gets its value by calling a method on the serializer class it is attached to. It can be used to add any sort of data to the serialized representation of your object.

Signature: `SerializerMethodField(method_name=None)`

- `method_name` - The name of the method on the serializer to be called. If not included this defaults to `get_<field_name>`.

The serializer method referred to by the `method_name` argument should accept a single argument (in addition to `self`), which is the object being serialized. It should return whatever you want to be included in the serialized representation of the object. For example:

```
from django.contrib.auth.models import User
from django.utils.timezone import now
from rest_framework import serializers

class UserSerializer(serializers.ModelSerializer):
    days_since_joined = serializers.SerializerMethodField()

    class Meta:
        model = User

    def get_days_since_joined(self, obj):
        return (now() - obj.date_joined).days
```

Custom fields

If you want to create a custom field, you'll need to subclass `Field` and then override either one or both of the `.to_representation()` and `.to_internal_value()` methods. These two methods are used to convert between the initial datatype, and a primitive, serializable datatype. Primitive datatypes will typically be any of a number, string, boolean, `date`/`time`/`datetime` or `None`. They may

also be any list or dictionary like object that only contains other primitive objects. Other types might be supported, depending on the renderer that you are using.

The `.to_representation()` method is called to convert the initial datatype into a primitive, serializable datatype.

The `to_internal_value()` method is called to restore a primitive datatype into its internal python representation. This method should raise a `serializers.ValidationError` if the data is invalid.

Note that the `WritableField` class that was present in version 2.x no longer exists. You should subclass `Field` and override `to_internal_value()` if the field supports data input.

Examples

Let's look at an example of serializing a class that represents an RGB color value:

```
class Color(object):
    """
    A color represented in the RGB colorspace.
    """
    def __init__(self, red, green, blue):
        assert(red >= 0 and green >= 0 and blue >= 0)
        assert(red < 256 and green < 256 and blue < 256)
        self.red, self.green, self.blue = red, green, blue

class ColorField(serializers.Field):
    """
    Color objects are serialized into 'rgb(#, #, #)' notation.
    """
    def to_representation(self, obj):
        return "rgb(%d, %d, %d)" % (obj.red, obj.green, obj.blue)

    def to_internal_value(self, data):
        data = data.strip('rgb(').rstrip(')')
        red, green, blue = [int(col) for col in data.split(',')]
        return Color(red, green, blue)
```

By default field values are treated as mapping to an attribute on the object. If you need to customize how the field value is accessed and set you need to override `.get_attribute()` and/or `.get_value()`.

As an example, let's create a field that can be used represent the class name of the object being serialized:

```
class ClassNameField(serializers.Field):
    def get_attribute(self, obj):
        # We pass the object instance onto `to_representation`,
        # not just the field attribute.
        return obj
```

```
    def to_representation(self, obj):
        """
        Serialize the object's class name.
        """
        return obj.__class__.__name__
```

Raising validation errors

Our `ColorField` class above currently does not perform any data validation. To indicate invalid data, we should raise a `serializers.ValidationError`, like so:

```
def to_internal_value(self, data):
    if not isinstance(data, six.text_type):
        msg = 'Incorrect type. Expected a string, but got %s'
        raise ValidationError(msg % type(data).__name__)

    if not re.match(r'^rgb\([0-9]+,[0-9]+,[0-9]+\)$', data):
        raise ValidationError('Incorrect format. Expected `rgb(#,#,#)`.')

    data = data.strip('rgb(').rstrip(')')
    red, green, blue = [int(col) for col in data.split(',')]

    if any([col > 255 or col < 0 for col in (red, green, blue)]):
        raise ValidationError('Value out of range. Must be between 0 and 255.')

    return Color(red, green, blue)
```

The `.fail()` method is a shortcut for raising `ValidationError` that takes a message string from the `error_messages` dictionary. For example:

```
default_error_messages = {
    'incorrect_type': 'Incorrect type. Expected a string, but got {input_type}',
    'incorrect_format': 'Incorrect format. Expected `rgb(#,#,#)`.',
    'out_of_range': 'Value out of range. Must be between 0 and 255.'
}

def to_internal_value(self, data):
    if not isinstance(data, six.text_type):
        msg = 'Incorrect type. Expected a string, but got %s'
        self.fail('incorrect_type', input_type=type(data).__name__)

    if not re.match(r'^rgb\([0-9]+,[0-9]+,[0-9]+\)$', data):
        self.fail('incorrect_format')

    data = data.strip('rgb(').rstrip(')')
    red, green, blue = [int(col) for col in data.split(',')]

    if any([col > 255 or col < 0 for col in (red, green, blue)]):
        self.fail('out_of_range')

    return Color(red, green, blue)
```

This style keeps you error messages more cleanly separated from your code, and should be preferred.

Third party packages

The following third party packages are also available.

DRF Compound Fields

The drf-compound-fields package provides "compound" serializer fields, such as lists of simple values, which can be described by other fields rather than serializers with the `many=True` option. Also provided are fields for typed dictionaries and values that can be either a specific type or a list of items of that type.

DRF Extra Fields

The drf-extra-fields package provides extra serializer fields for REST framework, including `Base64ImageField` and `PointField` classes.

djangrestframework-recursive

the djangorestframework-recursive package provides a `RecursiveField` for serializing and deserializing recursive structures

django-rest-framework-gis

The django-rest-framework-gis package provides geographic addons for django rest framework like a `GeometryField` field and a GeoJSON serializer.

django-rest-framework-hstore

The django-rest-framework-hstore package provides an `HStoreField` to support django-hstore `DictionaryField` model field.