

---

# **graceful Documentation**

***Release 0.2.0***

**Michał Jaworski**

**Mar 28, 2017**



---

## Contents

---

<b>1</b>	<b>Contents:</b>	<b>1</b>
1.1	Graceful guide . . . . .	1
1.2	API reference . . . . .	21
<b>2</b>	<b>graceful</b>	<b>41</b>
<b>3</b>	<b>python3 only</b>	<b>43</b>
<b>4</b>	<b>usage</b>	<b>45</b>
<b>5</b>	<b>contributing</b>	<b>49</b>
<b>6</b>	<b>license</b>	<b>51</b>
<b>7</b>	<b>Indices and tables</b>	<b>53</b>
	<b>Python Module Index</b>	<b>55</b>



## Graceful guide

### Resources

Resources are main building blocks in falcon. This is also true with graceful.

The most basic resource of all is a `graceful.resources.base.BaseResource` and all other resource classes in in this package inherit from `BaseResource`. It will not provide you with full set graceful features (like object serialization, pagination, resource fields descriptions etc.) but it is a good starting point if you want to build everything by yourself but still need to have some consistent response structure and self-descriptive parameters.

In most cases (simple GET-allowed resources) you need only to provide your own http GET method handler like following:

```
from graceful.resources.base import BaseResource
from graceful.parameters import StringParam, IntParam

class SomeResource(BaseResource):
    # describe how HTTP query string parameters are handled
    some_param = StringParam("example string query string param")
    some_other_param = IntParam("example integer query string param")

    def on_get(self, req, resp):
        # retrieve dictionary of query string parameters parsed
        # and validated according to resource class description
        params = self.require_params(req)

        ## create your own response like always:
        # resp.body = "some content"

        ## or use following:
        # self.make_body(resp, params, {}, 'some content')
```

---

**Note:** Due to how falcon works there is **always** only single instance of a resource class for a single registered route. Please remember to not keep any state inside of this object (i.e. in `self`) between any steps of response generation.

---

## Generic API resources

graceful provides you with some set of generic resources in order to help you describe how structured is data in your API. All of them expect that some serializer instance is provided as a class level attribute. Serializer will handle describing resource fields and also translation between resource representation and internal object values that you use inside of your application.

### RetrieveAPI

RetrieveAPI represents single element serialized resource. In 'content' section of GET response it will return single object. On OPTIONS request it will return additional field named 'fields' that describes all serializer fields.

It expects from you to implement `.retrieve(self, params, meta, **kwargs)` method handler that retrieves single object (e.g. from some storage) that will be later serialized using provided serializer.

`retrieve()` accepts following arguments:

- **params** (*dict*): dictionary of parsed parameters accordingly to definitions provided as resource class attributes.
- **meta** (*dict*): dictionary of meta parameters anything added to this dict will be later included in response 'meta' section. This can already be prepopulated by method that calls this handler.
- **kwargs** (*dict*): dictionary of values retrieved from route url template by falcon. This is suggested way for providing resource identifiers.

Example usage:

```
db = SomeDBInterface()
api = application = falcon.API()

class FooResource(RetrieveAPI):
    serializer = RawSerializer()

    def retrieve(self, params, meta, foo_id, **kwargs):
        return db.Foo.get(id=foo_id)

# note url template param that will be passed to `FooResource.get_object()`
api.add_route('foo/{foo_id}', FooResource())
```

### RetrieveUpdateAPI

RetrieveUpdateAPI extends RetrieveAPI with capability to update objects with new data from resource representation provided in PUT request body.

It expects from you to implement same handlers as for RetrieveAPI and also new `.update(self, params, meta, validated, **kwargs)` method handler that updates single object (e.g. in some storage). Updated object may or may not be returned in response 'content' section (this is optional)

`update()` accepts following arguments:

- **params** (*dict*): dictionary of parsed parameters accordingly to definitions provided as resource class attributes.

- **meta** (*dict*): dictionary of meta parameters anything added to this dict will be later included in response 'meta' section. This can already be prepopulated by method that calls this handler.
- **validated** (*dict*): dictionary of internal object fields values after converting from representation with full validation performed accordingly to definition contained within serializer instance.
- **kwargs** (*dict*): dictionary of values retrieved from route url template by falcon. This is suggested way for providing resource identifiers.

If update will return any value it should have same form as return value of `retrieve()` because it will be again translated into representation with serializer.

Example usage:

```
db = SomeDBInterface()
api = application = falcon.API()

class FooResource(RetrieveUpdateAPI):
    serializer = RawSerializer()

    def retrieve(self, params, meta, foo_id, **kwargs):
        return db.Foo.get(id=foo_id)

    def update(self, params, meta, foo_id, **kwargs):
        return db.Foo.update(id=foo_id)

# note url template param that will be passed to `FooResource.get_object()`
api.add_route('foo/{foo_id}', FooResource())
```

## RetrieveUpdateDeleteAPI

`RetrieveUpdateDeleteAPI` extends `RetrieveUpdateAPI` with capability to delete objects using DELETE requests.

It expects from you to implement same handlers as for `RetrieveUpdateAPI` and also new `.delete(self, params, meta, **kwargs)` method handler that deletes single object (e.g. in some storage).

`delete()` accepts following arguments:

- **params** (*dict*): dictionary of parsed parameters accordingly to definitions provided as resource class attributes.
- **meta** (*dict*): dictionary of meta parameters anything added to this dict will be later included in response 'meta' section. This can already be prepopulated by method that calls this handler.
- **kwargs** (*dict*): dictionary of values retrieved from route url template by falcon. This is suggested way for providing resource identifiers.

Example usage:

```
db = SomeDBInterface()
api = application = falcon.API()

class FooResource(RetrieveUpdateAPI):
    serializer = RawSerializer()

    def retrieve(self, params, meta, foo_id, **kwargs):
        return db.Foo.get(id=foo_id)

    def update(self, params, meta, foo_id, **kwargs):
        return db.Foo.update(id=foo_id)
```

```
def delete(self, params, meta, **kwargs):
    db.Foo.delete(id=foo_id)

# note url template param that will be passed to `FooResource.get_object()`
api.add_route('foo/{foo_id}', FooResource())
```

## ListAPI

ListAPI represents list of resource instances. In ‘content’ section of GET response it will return list of serialized objects representations. On OPTIONS request it will return additional field named ‘fields’ that describes all serializer fields.

It expects from you to implement `.list(self, params, meta, **kwargs)` method handler that retrieves list (or any iterable) of objects (e.g. from some storage) that will be later serialized using provided serializer.

`list()` accepts following arguments:

- **params** (*dict*): dictionary of parsed parameters accordingly to definitions provided as resource class attributes.
- **meta** (*dict*): dictionary of meta parameters anything added to this dict will be later included in response ‘meta’ section. This can already be prepopulated by method that calls this handler.
- **kwargs** (*dict*): dictionary of values retrieved from route url template by falcon. This is suggested way for providing resource identifiers.

Example usage:

```
db = SomeDBInterface()
api = application = falcon.API()

class FooListResource(ListAPIResource):
    serializer = RawSerializer()

    def list(self, params, meta, **kwargs):
        return db.Foo.all(id=foo_id)

# note that in most cases there is no need to define
# variables in url template for list type of resources
api.add_route('foo/', FooListResource())
```

## ListCreateAPI

ListCreateAPI extends ListAPI with capability to create new objects with data from resource representation provided in POST request body.

It expects from you to implement same handlers as for ListAPI and also new `.create(self, params, meta, validated, **kwargs)` method handler that creates single object (e.g. in some storage). Created object may or may not be returned in response ‘content’ section (this is optional)

`create()` accepts following arguments:

- **params** (*dict*): dictionary of parsed parameters accordingly to definitions provided as resource class attributes.
- **meta** (*dict*): dictionary of meta parameters anything added to this dict will be later included in response ‘meta’ section. This can already be prepopulated by method that calls this handler.



- **validated** (*dict*): dictionary of internal object fields values after converting from representation with full validation performed accordingly to definition contained within serializer instance.
- **kwargs** (*dict*): dictionary of values retrieved from route url template by falcon. This is suggested way for providing resource identifiers.

If `create()` will return any value it should have same form as return value of `retrieve()` because it will be again translated into representation with serializer.

Example usage:

```
db = SomeDBInterface()
api = application = falcon.API()

class FooListResource(ListAPIResource):
    serializer = RawSerializer()

    def list(self, params, meta, **kwargs):
        return db.Foo.all(id=foo_id)

    def create(self, params, meta, validated, **kwargs):
        return db.Foo.create(**validated)

# note that in most cases there is no need do define
# variables in url template for list type of resources
api.add_route('foo/', FooListResource())
```

## Paginated generic resources

PaginatedListAPI and PaginatedListCreateAPI are versions of respectively ListAPI and ListAPI classes that supply with simple pagination build with following parameters:

- **page\_size**: size of a single response page
- **page**: page count

They also will 'meta' section with following information on GET requests:

- **page\_size**
- **page**
- **next** - url query string for next page (only if meta['is\_more'] exists and is True)
- **prev** - url query string for previous page (None if first page)

Paginated variations of generic list resource do not assume anything about your resources so actual pagination must still be implemented inside of `list()` handlers. Anyway this class allows you to manage params and meta for pagination in consistent way across all of your resources if you only decide to use it:

```
db = SomeDBInterface()
api = application = falcon.API()

class FooPaginatedResource(PaginatedListAPI):
    serializer = RawSerializer()

    def list(self, params, meta, **kwargs):
        query = db.Foo.all(id=foo_id).offset(
            params['page'] * params['page_size']
        ).limit(
            params['page_size']
```

```
)

# use meta['has_more'] to find out if there are
# any pages behind this one
if db.Foo.count() > (params['page'] + 1) * params['page_size']:
    meta['has_more'] = True

return query

api.add_route('foo/', FooPaginatedResource())
```

---

**Note:** If you don't like anything about this opinionated meta section that paginated generic resources provide, you can always override it with own `add_pagination_meta(params, meta)` method handler.

---

## Generic resources without serialization

If you don't like how serializers work there are also two very basic generic resources that does not rely on serializers: `Resource` and `ListResource`. They can be extended with mixins found in `graceful.resources.mixins` module and provide same method handlers like generic resources that utilize serializers (`list()`, `retrieve()`, `update()` etc.) but do not perform anything more beyond content-type level serialization.

## Parameters

Parameters provide a way to describe and evaluate all request query params that can be used in your API resources.

New parameters are added to resources as class attributes:

```
from graceful.parameters import StringParam, IntParam
from graceful.resources.base import BaseResource

class SomeResource(BaseResource):
    filter_by_name = StringParam("Filter resource instances by their name")
    depth = IntParam("Set depth of search")
```

Class attribute names map directly to names expected in the query string. For example the valid query strings in scope of preceding definition could be:

- `filter_by_name=cats`
- `filter_by_name=dogs&depth=2`

All param classes accept this set of arguments:

- **details** (*str*): verbose description of parameter. Should contain all information that may be important to your API user and will be used for describing resource on `OPTIONS` requests and `.describe()` call.
- **label** (*str*): human readable label for this parameter (it will be used for describing resource on `OPTIONS` requests).

*Note that it is recommended to use parameter names that are self-explanatory instead of relying on param labels.*

- **required** (*bool*): if set to `True` then all `GET`, `POST`, `PUT`, `PATCH` and `DELETE` requests will return `400 Bad Request` response if query param is not provided.
- **default** (*str*): set default value for param if it is not provided in request as query parameter. This **MUST** be a raw string value that will be then parsed by `.value()` handler.

If default is set and `required` is `True` it will raise `ValueError` as having required parameters with default value has no sense.

- **param** (*str*): set to `True` if multiple occurrences of this parameter can be included in query string, as a result values for this parameter will be always included as a list in `params` dict. Defaults to `False`.

---

**Note:** If `many==False` and client includes multiple values for this parameter in query string then only one of those values will be returned, and it is undefined which one.

---

For list of all available parameter classes please refer to `graceful.parameters` module reference.

If you are using the bare falcon HTTP method handlers and subclass directly from `graceful.resources.base.BaseResource` then you can access all deserialized query parameters as dictionary using `require_params(req)` method:

```
from graceful.parameters import StringParam, IntParam
from graceful.resources.base import BaseResource

class SomeResource(BaseResource):
    filter_by_name = StringParam("Filter resource instances by their name")
    depth = IntParam("Set depth of search")

    def on_get(self, req, resp):
        params = self.require_params(req)
```

The `self.require_params(req)` will try to retrieve all of described query parameters, validate them and populate with defaults if they were not found in the query string. This method will also take care of raising the `falcon.errors.HTTPInvalidParam` if:

- parameter specified as `required=True` was not provided
- parameter could not be parsed/validated (i.e. `value()` handler raised `ValueError`)

Note that you do not need to handle this exception manually. It will be later automatically transformed to 400 Bad Request by falcon if not caught by `try .. except` clause.

If you are using generic resource classes from `graceful.resources.generic` like `ListAPI` or `RetrieveAPI` the params retrieval step is done automatically and you do not need to care. Parameters dict will be provided in applicable retrieval/modification method handler (`list()`, `update()`, `retrieve` etc.) and these methods will be executed only if call to `self.require_params(req)` succeeded without raising any exceptions.

## Custom parameters

Although *graceful* ships with some set of predefined parameter classes it is very likely that you need something that is not yet covered because:

- it is *not yet* covered
- is very specific to your application
- it can be implemented in many ways and it is impossible to decide which is best without being too opinionated.

New parameter types can be created by subclassing `BaseParam` and implementing `.value(raw_value)` method handler. `ValueError` raised in this handler will eventually result in 400 Bad Request response.

Two additional class-level attributes help making more verbose parameter description:

- **type** - string containing name of primitive data type like: “int”, “string”, “float” etc. For most custom parameters this will be simply “string” and it is used only for descriptions so make sure it is something truly generic or well described in your API documentation
- **spec** - two-tuple containing link name, and link url to any external documentation that you may find helpful for developers.

Here is example of custom parameter that handles validation of alpha2 country codes using pycountry module:

```
import pycountry

class LanguageParam(BaseParam):
    """
    This param normalizes language code passed to is and checks if it is valid
    """

    type = 'ISO 639-2 alpha2 language code'
    spec = (
        'ISO 639-2 alpha2 code list',
        'http://www.loc.gov/standards/iso639-2/php/code_list.php',
    )

    def value(self, raw_value):
        try:
            # normalize code since we store then lowercase
            normalized = raw_value.lower()
            # first of all check if country so no query will be made if it is
            # invalid
            pycountry.languages.get(alpha2=normalized)

            return normalized

        except KeyError:
            raise ValueError(
                "'{code}' is not valid alpha2 language code"
                "".format(code=raw_value)
            )
```

## Parameter validation

Custom parameters are great for defining new data types that can be passed through HTTP query string or handling very specific cases like country codes, mime types, or even database filters. Still it may be sometimes an overkill to define new parameter class to do something as simple as ensure min/max bounds for numeric value or define as set of allowed choices.

All of basic parameters available in graceful accept `validators` keyword argument that accepts a list of validation functions. These function will be always called upon parameter retrieval. This functionality allows you to quickly extend the semantic of your parameters without the need of subclassing.

A validator is any callable that accepts single positional argument that will be a value returned from call to the `value()` handler of parameter class. If validation function fails it is supposed to return `graceful.errors.ValidationError` that will be later translated to proper HTTP error response. Following is example of simple validation function which ensures that parameter string is palindrome:

```
from graceful.resources.base import BaseResource
from graceful.parameters import StrParam
from graceful.errors import ValidationError
```

```
def is_palindrome(value):
    if value != value[::-1]:
        raise ValidationError("{} is not a palindrome")

class FamousPhrases(Resource):
    palindrome_query = StrParam(
        "Palindrome text query", validators=[is_palindrome]
    )
```

Validators always work on deserialized values and this allows to easily reuse the same code across different types of parameters and also between fields (see: [Field validation](#)). Graceful takes advantage of this fact and already provides you with a small set of fully reusable validators that can be used to validate both your parameters and serialization fields. For more details see [graceful.validators](#) module reference.

## Handling multiple occurrences of parameters

The simplest way to allow user to specify multiple occurrences of single parameter is to use `many` keyword argument. It is available for every base parameter class initialization and it is good practice to not override this argument in custom parameter classes using custom initialization.

If `many` is set to `True` for given parameter the resulting `params` dictionary available in main method handlers of generic resources or through `self.require_params(req)` method will contain list of values for given resource instead of single value.

For instance, if you are building some text search API and expect client to provide multiple search string in single query you can describe your basic API as follows:

```
from graceful.parameters import StringParam
from graceful.resources.base import BaseResource

class SearchResource(BaseResource):
    search = StringParam("text search string", many=True)
```

With such definition your client can provide list of multiple values for the `search` param using multiple instances of `search=<value>` in his query string e.g:

```
search=matt&search=damon&search=affleck
```

**Important:** if `many` is set to `False` the value stored under corresponding key will **always** represent the single parameter value. It is important to note that providing multiple values for same parameter in the query string by your API client is not considered an error even if parameter is described as `many=False`. In that case only one value will be included in parameters dictionary and it is not defined which one. When documenting your API you need to take special care when informing which parameter supports multiple value and which not. You should also make sure to inform API users of possibility of undefined behaviour when not following your instructions.

## Order of values and ordered data

Remember that multiple values coming from parameter defined using `many=True` should be always considered independent from each other. This means that **order of resulting parameter values is always undefined**. If you need to handle parameters that represent specifically ordered list you probably need custom parameter class that will provide you with required serialization. Such representation is generally independent from the `many` argument of such custom parameter class.

The reason for that design decision is because when order of data is important then usually the order by itself represents is a named quality or entity.

The best way to understand this is by example. For instance let's assume that we are building some simple API that allows to search through some inventory of clothes store. If we would like to allow clients to filter items by their colors it completely makes sense to use following definition of query parameter:

```
color = StringParam("One of main color items", many=True)
```

But if you are building some spatial search engine you might want to allow user to search for data in region defined as a polygon. Polygon can be simply represented by just an ordered list of points. But does it makes sense to define your polygon as point parameter with `many=True`? Probably not. In case where order of data is important you need some custom parameter class that will explicitly define how to handle such parameters. The naive implementation for polygon parameter could be as follows: The naive

```
from graceful.parameters import BaseParam

class PolygonParam(BaseParam):
    """ Represents polygon parameter in string form of "x1,y1;x2,y2;..."
    """
    type = 'polygon'

    def value(self, raw_value):
        return [
            [float(x) for x in point.split(',')]
            for point in raw_value.split(';')
        ]
```

Such approach you will eventually make your code and API:

- Easier to understand - you will end up using parameter names that better explain what you and your API users are dealing with.
- Easier to document - parameter class can be inspected for the purpose of auto documentation. Their basic attributes (`type` and `spec`) are already included in default `OPTIONS` handler.
- Easier to extend - if you suddenly realize that you need to support multiple ordered sets of same type of data it is as simple as adding additional `many=True` to declaration of parameter that represents some data container

## Custom containers

With the `many=True` option multiple values for the same parameter will be returned as list. But sometimes you may want to do additional processing when `many` option is enables. For instance you may want to concatenate all string searches to single string, make sure all values are unique or join some ORM query sets using logical operator.

Of course it is completely valid approach to make such operation in your HTTP method handler (in case of using `BaseResource`) or in your specific retrieval/update handler (in case of using generic resource classes). This is usually very simple:

```
from graceful.parameters import StringParam
from graceful.resources.generic import PaginatedListAPI

class CatList(PaginatedListAPI):
    """
    List of all cats in our API
    """
```

```

breed = StringParam(
    "set this param to filter cats by breed"
    many=True
)

def list(self, params, meta, **kwargs):
    unique_breeds = set(param['breed'])
    ...

```

Unfortunately, when you have a lot of different parameters that need similar handling (e.g. various ORM-specific filter objects) this can become tedious and lead to excessive code duplication. The easiest way overcome this problem is to use custom container handler for multiple parameter occurrences. This can be done in your custom parameter class by overriding its default `container` attribute.

The container handler can be both type object or a new method. It must accept list of values as its single positional argument.

Following is an example `StringParam` re-implementation which additionally makes sure that multiple occurrences of the same parameter are all unique. Uniqueness is simply achieved by using built-in `set` type as its `container` attribute:

```

from graceful.parameters import BaseParam

class PolygonParam(BaseParam):
    """ Represents polygon parameter in string form of "x1,y1;x2,y2;..."
    """
    container = set

```

As already said, container handler can be a method too. This is very useful for handling more complex use cases. For instance `solrq` is a nice utility for creating `Apache Solr` search engine queries in Python. If your API somehow exposes Solr search it would be nice to make parameter class that converts query string params directly to `solrq.Q` objects. `solrq` allows also to easily join multiple query objects using binary AND and OR operators in similar fashion to Django's queryset filters:

```

>>> Q(text='cat') | Q(text='dog')
<Q: text:cat OR text:dog>

```

It really makes sense to take advantage of such feature in your parameter class that wraps GET params in `solrq.Q` instances whenever `many=True` option is enabled. Following is example of custom parameter class that allows to collapse multiple values of search queries to single `solrq.Q` instance with predefined operator:

```

from graceful.params import StringParam

import operator
from functools import reduce

class FilterQueryParam(StringParam):
    """
    Param that represents Solr filter queries logically
    joined together depending on value of `op` argument
    """
    def __init__(
        self,
        details,
        solr_field,
        op=operator.and_,
        **kwargs
    ):

```

```
    if solr_field is None:
        raise ValueError("{} needs a `field` param cannot be None".format(
            self.__class__.__name__))
    )

    self.solr_field = solr_field
    self.op = op

    super(FilterQueryParam, self).__init__(
        details, **kwargs
    )

    def value(self, raw_value):
        return Q({self.solr_field: raw_value})

    def container(self, values):
        return reduce(self.op, values) if len(values) > 1 else values[0]
```

With such definition creating simple Solr-backed search API using graceful and without extensive object serialization becomes pretty simple:

```
import operator

from solrq import Value as V
from pysolr import Solr
from graceful.resources.generic import ListAPI
from graceful.serializers import BaseSerializer

solr = Solr()

class VerbatimSerializer():
    """ Represents object as it is assuming that we deal with simple dicts
    """
    def to_representation(self, obj):
        return obj

class Search(ListAPI):
    serializer = VerbatimSerializer()

    text = FilterQueryParam(
        "Basix text search argument (many values => AND)",
        many=True,
        solr_field='text'
        default=V('*', safe=True)
    )

    category = StringParam(
        "set this param to filter cats by breed (many values => OR)"
        many=True,
        solr_field='category'
        default=V('*', safe=True),
        op=operator.or_,
    )

    def list(self, params, meta, **kwargs):
        return list(solr.search(params['text'] & params['category']))
```



## Serializers and fields

The purpose of serializers and fields is to describe how structured is data that your API resources can return and accept. They together describe what we could call a “resource representation”.

They also helps binding this resource representation with internal objects that you use in your application no matter what you have there - dicts, native, class instances, ORM objects, documents, whatever. There is only one requirement: there must be a way to represent them as a set of independent fields and their values. In other words: dictionaries.

Example of simple serializer:

```
from graceful.serializers import BaseSerializer
from graceful.fields import RawField, IntField, FloatField

class CatSerializer(BaseSerializer):
    species = RawField("non normalized cat species")
    age = IntField("cat age in years")
    height = FloatField("cat height in cm")
```

Serializers are intended to be used with generic resources provided by `graceful.resources.generic` module so only handlers for retrieving, updating, creating etc. of objects from validated data is needed:

Functionally equivalent example using generic resources:

```
from graceful.resources.generic import RetrieveUpdateAPI
from graceful.serializers import BaseSerializer
from graceful.fields import RawField, FloatField

class Cat(object):
    def __init__(self, name, height):
        self.name = name
        self.height = height

class CatSerializer(BaseSerializer):
    name = RawField("name of a cat")
    height = FloatField("height in cm")

class CatResource(RetrieveUpdateAPI):
    serializer = CatSerializer()

    def retrieve(self, params, meta, **kwargs):
        return Cat('molly', 30)

    def update(self, params, meta, validated, **kwargs):
        return Cat(**validated)
```

Anyway serializers can be used outside of generic resources but some additional work need to be done then:

```
import falcon

from graceful.resources.base import BaseResource

class CatResource(BaseResource):
    serializer = CatSerializer()

    def on_get(self, req, resp, **kwargs):
        # this in probably should be read from storage
        cat = Cat('molly', 30)
```

```
self.make_body(
    req, resp,
    meta={},
    content=self.serializer.to_representation(cat),
)

def on_put(self, req, resp, **kwargs)
    validated = self.require_validated(req)
    updated_cat = Cat(**validated)

    self.make_body(
        req, resp,
        meta={},
        # may be nothing or again representation of new cat
        content=self.serializer.to_representation(new_cat),
    )

    req.status = falcon.HTTP_CREATED
```

## Field arguments

All field classes accept this set of arguments:

- **details** (*str, required*): verbose description of field.
- **label** (*str, optional*): human readable label for this field (it will be used for describing resource on OPTIONS requests).

*Note that it is recommended to use field names that are self-explanatory instead of relying on param labels.*

- **source** (*str, optional*): name of internal object key/attribute that will be passed to field's `.to_representation(value)` call. Special '\*' value is allowed that will pass whole object to field when making representation. If not set then default source will be a field name used as a serializer's attribute.
- **validators** (*list, optional*): list of validator callables.
- **many** (*bool, optional*) set to True if field is in fact a list of given type objects

---

**Note:** `source='*'` is in fact a dirty workaround and will not work well on validation when new object instances needs to be created/updated using POST/PUT requests. This works quite well with simple retrieve/list type resources but in more sophisticated cases it is better to use custom object properties as sources to encapsulate such fields.

---

## Field validation

Additional validation of field value can be added to each field as a list of callables. Any callable that accepts single argument can be a validator but in order to provide correct HTTP responses each validator should raise `graceful.errors.ValidationError` exception on validation call.

---

**Note:** Concept of validation for fields is understood here as a process of checking if data of valid type (successfully parsed/processed by `.from_representation` handler) does meet some other constraints (length, bounds, unique, etc).

---

Example of simple validator usage:

```

from graceful.errors import ValidationError
from graceful.serializers import BaseSerializer
from graceful.fields import FloatField

def tiny_validator(value):
    if value > 20.0:
        raise ValidationError

class TinyCats(BaseSerializer):
    """ This resource accepts only cats that has height <= 20 cm """
    height = FloatField("cat height", validators=[tiny_validator])

```

graceful provides some small set of predefined validator helpers in `graceful.validators` module.

## Resource validation

In most cases field level validation is all that you need but sometimes you need to perform object level validation that needs to access multiple fields that are already deserialized and validated. Suggested way to do this in graceful is to override serializer's `.validate()` method and raise `graceful.errors.ValidationError` when your validation fails. This exception will be then automatically translated to HTTP Bad Request response on resource-level handlers. Here is example:

```

class DrinkSerializer():
    alcohol = StringField("main ingredient", required=True)
    mixed_with = StringField("what makes it tasty", required=True)

    def validate(self, object_dict, partial=False):
        # note: always make sure to call super `validate()`
        # so whole validation of fields works as expected
        super().validate(object_dict, partial)

        # here is a place for your own validation
        if (
            object_dict['alcohol'] == 'whisky' and
            object_dict['mixed_with'] == 'cola'
        ):
            raise ValidationError("bartender refused!")

```

## Custom fields

Custom field types can be created by subclassing of `BaseField` class and implementing of two method handlers:

- `.from_representation(raw)`: returns internal data type from raw string provided in request
- `.to_representation(data)`: returns representation of internal data type

Example of custom field that assumes that data in internal object is stored as a serialized JSON string that we would like to (de)serialize:

```

import json

from graceful.fields import BaseField

class JSONField(BaseField):

```

```
def from_representation(raw):  
    return json.dumps(raw)  
  
def to_representation(data):  
    return json.loads(data)
```

## Content types

graceful currently talks only JSON. If you want to support other content-types then the only way is to override `BaseResource.make_body()`, `BaseResource.require_representation()` and optionally `BaseResource.on_options()` etc. methods. Suggested way would be to create a class mixin that can be added to every of your resources but ideally it would be great if someone contributed code that adds reasonable content negotiation and pluggable content-type serialization.

## Documenting your API

Providing clear and readable documentation is a very important topic for every API creator. Graceful does not come with a built-in autodoc feature yet, but is built in a way that allows you to create your documentation very easily.

Every important building block that creates your API definition in graceful (resource, parameter, and field classes) comes with a special `describe()` method that returns a dictionary of all important metadata necessary to create clear and readable documentation. Additionally, generic API resources (*[RetrieveAPI](#)*, *[ListAPI](#)*, *[ListCreateAPI](#)* and so on) are aware of their associated serializers to ease the whole process of documenting your service.

## Using self-descriptive resources

The easiest way to access API metadata programmatically is to issue `OPTIONS` request to the API endpoint of choice. Example how to do that was already presented in project's [README](#) file and main documentation page. Using this built-in capability of graceful's resources it should be definitely easy to populate your HTML/JS based documentation portal with API metadata.

This is the preferred way to construct documentation portals for your API. It has many advantages compared to documentation self-hosted within the same application as your API service. Just to name a few:

- Documentation deployment is decoupled from deployment of your API service. Documentation portal can be stored in a completely different project and does not even need to be hosted on the same machines as your API.
- Documentation portal may require completely different requirements that could be in conflict with you.
- APIs are often secured on different layers and using different authentication and authorization schemes. But documentations for such APIs are very often left open. If you keep them both separated it will allow you to reduce complexity of both projects.
- Changes to documentation layout and aesthetics do not require new deployments of whole service. This makes your operations more robust.

The popular [Swagger](#) project is built with a similar idea in mind. If you like this project and are already familiar with it, you should be able to easily translate API metadata returned by graceful to a format that is accepted by Swagger.

## Self-hosted documentation

Decoupling documentation portal from your API service is in many cases the most reliable option. Anyway, there are many use cases where such an approach might be simply inconvenient. For instance, if you distribute your project as a

downloadable package (e.g. through PyPI) you may want to make it easily accessible for new users without the need of bootstrapping multiple processes and services.

In such cases it might be reasonable to generate documentation in format that is convenient to the user by the same process that serves your API requests. The same features that allow you to easily access API metadata via `OPTIONS` requests allow you to introspect resources within your application process and populate any kind of documents.

The most obvious approach is to create some HTML templates, fill them with data retrieved from `describe()` method of each resource and serve them directly to the user via HTTP.

Graceful can't do all of that out of the box (maybe in future) but general process is very simple and does not require a lot of code. Additionally, you have full control over what tools you want to use to build documentation.

In this section we will show how it could be done using some popular tools like [Jinja](#) and [python-hoedown](#) but no one forces you to use specific template language or text markup. Choose anything you like and anything you are comfortable with. All code that is featured in this guide is also available in the [demo](#) directory in the project repository.

## Serving HTML and using Jinja templates in falcon

Graceful isn't a full-fledged framework like Django or Flask. It is only a toolkit that allows you to define REST APIs in a clean and convenient way. Only that and nothing more.

Neither Graceful nor Falcon have built-in support for generating HTML responses because it is not their main use case. But serving HTML isn't by any means different from responding with JSON, XML, YAML, or any other content type. What you need to do is to put your HTML to the body section of your response and set proper value of the `Content-Type` header. Here is simple example of falcon resource that serves some html:

```
import falcon

class HtmlResource:
    def on_get(self, req, resp):
        resp.body = """
        <!DOCTYPE html>
        <html>
        <head><title>Hello World!</title></head>
        <body>
        <h1>Hello World!</h1>
        </body>
        </html>
        """
        resp.status = falcon.HTTP_200
        resp.content_type = 'text/html'
```

Of course no one wants to generate documentation relying solely on `str.format()`. One useful feature that many web frameworks offer is some kind of templating engine that allows you to easily format different kinds of documents. If you want to build beautiful documentation you will eventually need a one. For the purpose of this example we will use Jinja that is usually a very good choice and is very easy to start with.

In our documentation pages, we don't want to support any query string parameters or define CRUD semantics. So we don't need any of Graceful's generic classes, parameters of serializers. Let's build simple falcon resource that will allow us to respond with templated HTML response that may be populated with some predefined (or dynamic) context:

```
from jinja2 import Environment, FileSystemLoader

# environment allows us to load template files, 'templates' is a dir
# where we want to store them
env = Environment(loader=FileSystemLoader('templates'))
```

```
class Templated(object):
    template_name = None

    def __init__(self, template_name=None, context=None):
        # note: this is to ensure that template_name can be set as
        #       class level attribute in derived class
        self.template_name = template_name or self.template_name
        self.context = context or {}

    def render(self, req, resp):
        template = env.get_template(self.template_name)
        return template.render(**self.context)

    def on_get(self, req, resp):
        resp.body = self.render(req, resp)
        resp.content_type = 'text/html'
```

Assuming we have `index.html` Jinja template stored in the `templates` directory we can start to serve your first HTML from falcon by adding `Templated` resource instance to your app router:

```
api.add_route("/", Templated('index.html'))
```

## Populating templates with resources metadata

Once you are able to generate HTML pages from template it's time to populate them with resource metadata. Every resource class instance in Graceful provides `describe()` method that returns dictionary that contains metadata with information about it's resource structure (fields), accepted HTTP methods, query string parameters, and so on. The general structure is as follows:

```
{
    "details": ...           # => Resource class docstring
    "fields": {              # => Description of resource representation fields
        "<field_name>": {
            "details": ...,   # => Field definition 'details' string
            "label": ...,     # => Field definition 'label' string
            "spec": ...,      # => Additional specification tuple associated
                               # with specific field class. It is usually
                               # standard name (e.g. ISO 639-2), and URL to its
                               # official documentation
            "type": ...,      # => Generic type name like 'string', 'bool', etc.
        },
        ...
    },
    "methods": [...],        # => List of accepted HTTP methods (uppercase)
    "name": "CatList",       # => Resource class name
    "params": {              # => Description of accepted query string params
        "<param_name>": {
            "default": ...,   # => Default parameter value
            "details": ...,   # => Param definition 'details' string
            "label": ...,
            "required": ...,  # => Flag indicating if parameter is requires (bool)
            "spec": ...,      # => Additional specification tuple associated
                               # with specific param class. It is usually
                               # standard name (e.g. ISO 639-2), and URL to its
                               # official documentation
        },
        ...
    },
    ...
}
```

```

        "type": "...",      # => Generic type name like 'string', 'bool', etc.
    },
    },
    "path": ...,           # => URI leading to resource (only available
                           #   on OPTIONS requests)
    "type": ...,           # => General type of resource representation form.
                           #   It may be "object" for single resource
                           #   representation or "list" for endpoints that
                           #   return list of resource representations.
}

```

Knowing that resource descriptions have well defined and consistent structure we can add them to predefined context of our Templated resource. Because all API resources are always associated with their URIs (which are unique per resource class), it is a good approach to group descriptions by their URI templates from falcon router.

Let's assume we want to document Cats API example presented in main documentation page. Here is falcon's router configuration that adds Cats API resources and additional templated documentation resource that can render our service metadata in human readable form:

```

api.add_route("/v1/cats/{cat_id}", V1.Cat())
api.add_route("/v1/cats/", V1.CatList())
api.add_route("/", Templated('index.html', {
    'endpoints': {
        "/v1/cats/": V1.CatList().describe(),
        "/v1/cats/{cat_id}": V1.Cat().describe(),
    }
}))

```

For APIs that contain a lot of multiple resources it is always better to follow “don't repeat yourself” principle:

```

api = application = falcon.API()

endpoints = {
    "/v1/cats/{cat_id}": V1.Cat(),
    "/v1/cats/": V1.CatList(),
}

for uri, endpoint in endpoints:
    api.add_route(uri, endpoint)

api.add_route("/", Templated('index.html', {
    'endpoints': {
        uri: endpoint.describe()
        for uri, endpoint
        in endpoints.items()
    }
}))

```

The last thing you need to do is to create a template that will be used to render your documentation. Here is a minimal Jinja template for Cats API that provides general overview on the API structure with plain HTML and without any fancy styling:

```

<!DOCTYPE html>
<html>
<head lang="en">
    <meta charset="UTF-8">
    <title>Cats API</title>
</head>

```

```
<body>

<h1>Cats API documentation</h1>

<p> Welcome to Cats API documentation </p>

{% for uri, endpoint in endpoints.items() %}
  <h2>{{ endpoint.name }}: <code>{{ uri }}</code></h2>

  <p>
    <strong>Accepted methods:</strong>
    <code>{{ endpoint.methods }}</code>
  </p>

  <p> {{ endpoint.details }}</p>

  <h3>Accepted params</h3>
  {% if endpoint.params %}
    <ul>
      {% for name, param in endpoint.params.items() %}
        <li>{{ name }} ({{ param.type }}): {{ param.details }}</li>
      {% endfor %}
    </ul>
  {% endif %}

  <h3>Accepted fields</h3>
  {% if endpoint.fields %}
    <ul>
      {% for name, field in endpoint.fields.items() %}
        <li>{{ name }} ({{ field.type }}): {{ field.details }}</li>
      {% endfor %}
    </ul>
  {% endif %}
{% endfor %}
</body>
</html>
```

## Formatting resource class docstrings

Building good service documentation is not an easy task but Graceful tries to make it at least a bit easier by providing you with some tools to introspect your service. Thanks to this you can take resource metadata and convert it to human readable form.

But your work does not end on providing the list of acceptable fields and parameters. Very often you may need to provide some more information about specific resource type like specific limits, usage example or rationale behind your design decisions. The best place to do that is the resource docstring that is always included in the result of `describe()` method call. This is very convenient way of managing even large parts of your documentation.

But when docstrings get longer and longer it is good idea to add a bit more structure to them instead of keeping them unformatted. A good idea is to use some lightweight markup language that is easy-to-read in plain text (so it is easy to edit by developer) but provides you with enough rendering capabilities to make your documentation look good for actual API user. A very popular choice for a lightweight markup is [Markdown](#).

It seems that everyone loves Markdown, but apparently there is no Markdown parser (at least available in Python) that would not suck terribly in some of its aspects. Anyway, Python binding to [hoedown](#) (that is fork of sundown, that is fork of upskirt, that is now a libsoldout...) has acceptable quality and can be successfully used for that purpose.



The best news is that it is insanely easy to integrate it with Jinja. The only thing you need to do is to create new template filter that will allow you to convert any string to HTML inside of you template. It could be something like following:

```
import hoedown
from jinja2 import Environment, FileSystemLoader

# environment allows us to load template files, 'templates' is a dir
# where we want to store them
env = Environment(loader=FileSystemLoader('templates'))

md = hoedown.Markdown(
    CustomRenderer(),
    extensions=hoedown.EXT_FENCED_CODE | hoedown.EXT_HIGHLIGHT
)

def markdown_filter(data):
    return md.render(data)

env.filters['markdown'] = markdown_filter
```

With such definition you can use your new filter anywhere in template where you expect string to be multiline Mark-down markup:

```
{% for uri, endpoint in endpoints.items() %}
    <h2>{{ endpoint.name }}: <code>{{ uri }}</code></h2>

    <p> {{ endpoint.details|markdown }}</p>
{% endfor %}
```

You can also use that technique to format multiline strings supplied as `details` arguments to fields and parameters definitions. Graceful will properly strip excessive leading whitespaces from them so you can easily use any indentation-sensitive markup language (like reStructuredText).

## API reference

### graceful package

#### graceful.fields module

**class** `graceful.fields.BaseField`(*details*, *label=None*, *source=None*, *validators=None*, *many=False*, *read\_only=False*)

Bases: `object`

Base field class for subclassing.

To create new field type subclass *BaseField* and implement following methods:

- `from_representation()`: converts representation (used in request/response body) to internal value.
- `to_representation()`: converts internal value to representation that will be used in response body.

#### Parameters

- **details** (*str*) – human readable description of field (it will be used for describing resource on OPTIONS requests).

- **label** (*str*) – human readable label of a field (it will be used for describing resource on OPTIONS requests).

*Note: it is recommended to use field names that are self-explanatory instead of relying on field labels.*

- **source** (*str*) – name of internal object key/attribute that will be passed to field on `.to_representation()` call. Special '\*' value is allowed that will pass whole object to field when making representation. If not set then default source will be a field name used as a serializer's attribute.
- **validators** (*list*) – list of validator callables.
- **many** (*bool*) – set to True if field is in fact a list of given type objects
- **read\_only** (*bool*) – True if field is read only and cannot be set/modified by POST and PUT requests

Example:

```
class BoolField(BaseField):
    def from_representation(self, data):
        if data in {'true', 'True', 'yes', '1', 'Y'}:
            return True:
        elif data in {'false', 'False', 'no', '0', 'N'}:
            return False:
        else:
            raise ValueError(
                "{data} is not valid boolean field".format(
                    data=data
                )
            )

    def to_representation(self, value):
        return ["True", "False"][value]
```

**describe** (*\*\*kwargs*)

Describe this field instance for purpose of self-documentation.

**Parameters** **kwargs** (*dict*) – dictionary of additional description items for extending default description

**Returns** *dict* – dictionary of description items

Suggested way for overriding description fields or extending it with additional items is calling super class method with new/overridden fields passed as keyword arguments like following:

```
class DummyField(BaseField):
    def description(self, **kwargs):
        super().describe(is_dummy=True, **kwargs)
```

**from\_representation** (*data*)

Convert representation value to internal value.

---

**Note:** This is method handler stub and should be redefined in the `BaseField` subclasses.

---

**spec** = None

**to\_representation** (*value*)

Convert representation value to internal value.

---

**Note:** This is method handler stub and should be redefined in the `BaseField` subclasses.

---

**type** = `None`

**validate** (*value*)

Perform validation on value by running all field validators.

Single validator is a callable that accepts one positional argument and raises “`ValidationError`” when validation fails.

Error message included in exception will be included in http error response

**Parameters** **value** – internal value to validate

**Returns** `None`

---

**Note:** Concept of validation for fields is understood here as a process of checking if data of valid type (successfully parsed/processed by `.from_representation` handler) does meet some other constraints (length, bounds, uniqueness, etc). So this method is always called with result of `.from_representation()` passed as its argument.

---

**class** `graceful.fields.BoolField` (*details*, *representations=None*, *\*\*kwargs*)

Bases: `graceful.fields.BaseField`

Represents boolean type of field.

By default accepts a wide range of incoming True/False representations:

- False: ['False', 'false', 'FALSE', 'F', 'f', '0', 0, 0.0, False]

- True: ['True', 'true', 'TRUE', 'T', 't', '1', 1, True]

By default, the output representations of internal object’s value are Python’s False/True values that will be later serialized to form that is native for content-type of use.

This behavior can be changed using `representations` field argument. Note that when using `representations` parameter you need to make strict decision and there is no ability to accept multiple options for true/false representations. Anyway, it is recommended approach to strictly define these values.

**Parameters** **representations** (*tuple*) – two-tuple with representations for (False, True) values, that will be used instead of default values

**from\_representation** (*data*)

Convert representation value to `bool` if it has expected form.

**to\_representation** (*value*)

Convert internal boolean value to one of defined representations.

**type** = `‘bool’`

**class** `graceful.fields.FloatField` (*details*, *max\_value=None*, *min\_value=None*, *\*\*kwargs*)

Bases: `graceful.fields.BaseField`

Represents float type of field.

Accepts both floats and strings as an incoming float number representation and always returns float as a representation of internal objects’s value that will be later serialized to form that is native for content-type of use.

This field accepts optional arguments that simply add new *max* and *min* value validation.

#### Parameters

- **max\_value** (*int*) – optional max value for validation
- **min\_value** (*int*) – optional min value for validation

**from\_representation** (*data*)

Convert representation value to float.

**to\_representation** (*value*)

Convert internal value to float.

**type** = 'float'

**class** graceful.fields.**IntField** (*details*, *max\_value=None*, *min\_value=None*, *\*\*kwargs*)

Bases: [graceful.fields.BaseField](#)

Represents integer type of field.

Field of this type accepts both integers and strings as an incoming integer representation and always returns int as a representation of internal objects's value that will be later serialized to form that is native for content-type of use.

This field accepts optional arguments that simply add new *max* and *min* value validation.

#### Parameters

- **max\_value** (*int*) – optional max value for validation
- **min\_value** (*int*) – optional min value for validation

**from\_representation** (*data*)

Convert representation value to int.

**to\_representation** (*value*)

Convert internal value to int.

**type** = 'int'

**class** graceful.fields.**RawField** (*details*, *label=None*, *source=None*, *validators=None*, *many=False*,  
*read\_only=False*)

Bases: [graceful.fields.BaseField](#)

Represents raw field subtype.

Any value from resource object will be returned as is without any conversion and no control over serialized value type is provided. Can be used only with very simple data types like int, float, str etc. but can eventually cause problems if value provided in representation has type that is not accepted in application.

Effect of using this can differ between various content-types.

**from\_representation** (*data*)

Return representation value as-is (note: content-type dependent).

**to\_representation** (*value*)

Return internal value as-is (note: content-type dependent).

**type** = 'raw'

**class** graceful.fields.**StringField** (*details*, *label=None*, *source=None*, *validators=None*,  
*many=False*, *read\_only=False*)

Bases: [graceful.fields.BaseField](#)

Represents string field subtype without any extensive validation.

```

from_representation (data)
    Convert representation value to str.

to_representation (value)
    Convert representation value to str.

type = 'string'

```

## graceful.parameters module

```

class graceful.parameters.Base64EncodedParam(details, label=None, required=False, default=None, many=False, validators=None)

```

Bases: `graceful.parameters.BaseParam`

Describes string parameter with value encoded using Base64 encoding.

**spec** = ('RFC-4648 Section 4', 'https://tools.ietf.org/html/rfc4648#section-4')

```

value (raw_value)
    Decode param with Base64.

```

```

class graceful.parameters.BaseParam(details, label=None, required=False, default=None, many=False, validators=None)

```

Bases: object

Base parameter class for subclassing.

To create new parameter type subclass `BaseParam` and implement `.value()` method handler.

### Parameters

- **details** (*str*) – verbose description of parameter. Should contain all information that may be important to your API user and will be used for describing resource on OPTIONS requests and `.describe()` call.
- **label** (*str*) – human readable label for this parameter (it will be used for describing resource on OPTIONS requests).

*Note that it is recommended to use parameter names that are self-explanatory instead of relying on param labels.*

- **required** (*bool*) – if set to `True` then all GET, POST, PUT, PATCH and DELETE requests will return 400 Bad Request response if query param is not provided. Defaults to `False`.
- **default** (*str*) – set default value for param if it is not provided in request as query parameter. This MUST be a raw string value that will be then parsed by `.value()` handler. If default is set and `required` is `True` it will raise `ValueError` as having required parameters with default value has no sense.

- **many** (*str*) –  
**set to `True` if multiple occurrences of this parameter** can be included in query string, as a result values for this parameter will be always included as a list in `params` dict. Defaults to `False`.

---

**Note:** If `many=False` and client includes multiple values for this parameter in query string then only one of those values will be returned, and it is undefined which one.

---

Example:

```
class BoolParam(BaseParam):
    def value(self, data):
        if data in {'true', 'True', 'yes', '1', 'Y'}:
            return True
        elif data in {'false', 'False', 'no', '0', 'N'}:
            return False
        else:
            raise ValueError(
                "{data} is not valid boolean field".format(
                    data=data
                )
            )
```

**container**

alias of list

**describe** (*\*\*kwargs*)

Describe this parameter instance for purpose of self-documentation.

**Parameters** **kwargs** (*dict*) – dictionary of additional description items for extending default description

**Returns** *dict* – dictionary of description items

Suggested way for overriding description fields or extending it with additional items is calling super class method with new/overridden fields passed as keyword arguments like following:

```
class DummyParam(BaseParam):
    def description(self, **kwargs):
        super().describe(is_dummy=True, **kwargs)
```

**spec** = None**type** = None**validated\_value** (*raw\_value*)

Return parsed parameter value and run validation handlers.

Error message included in exception will be included in http error response

**Parameters** **value** – raw parameter value to parse validate

**Returns** None

---

**Note:** Concept of validation for params is understood here as a process of checking if data of valid type (successfully parsed/processed by `.value()` handler) does meet some other constraints (length, bounds, uniqueness, etc.). It will internally call its `value()` handler.

---

**value** (*raw\_value*)

Raw value deserialization method handler.

**Parameters** **raw\_value** (*str*) – raw value from GET parameters

```
class graceful.parameters.BoolParam(details, label=None, required=False, default=None,
                                     many=False, validators=None)
```

Bases: `graceful.parameters.BaseParam`

Describes parameter with value expressed as bool.

Accepted string values for boolean parameters are as follows:

```

•False: ['True', 'true', 'TRUE', 'T', 't', '1']
•True: ['False', 'false', 'FALSE', 'F', 'f', '0', '0.0']

```

In case raw parameter value does not match any of these strings the `value()` method will raise `ValueError` method.

**type = 'bool'**

**value** (*raw\_value*)

Decode param as bool value.

```

class graceful.parameters.DecimalParam(details, label=None, required=False, default=None,
                                       many=False, validators=None)

```

Bases: `graceful.parameters.BaseParam`

Describes parameter with value expressed as decimal number.

**type = 'decimal'**

**value** (*raw\_value*)

Decode param as decimal value.

```

class graceful.parameters.FloatParam(details, label=None, required=False, default=None,
                                       many=False, validators=None)

```

Bases: `graceful.parameters.BaseParam`

Describes parameter with value expressed as float number.

**type = 'float'**

**value** (*raw\_value*)

Decode param as float value.

```

class graceful.parameters.IntParam(details, label=None, required=False, default=None,
                                    many=False, validators=None)

```

Bases: `graceful.parameters.BaseParam`

Describes parameter with value expressed as integer number.

**type = 'integer'**

**value** (*raw\_value*)

Decode param as integer value.

```

class graceful.parameters.StringParam(details, label=None, required=False, default=None,
                                       many=False, validators=None)

```

Bases: `graceful.parameters.BaseParam`

Describes parameter that will always be returned as-is (string).

Additional validation can be added to param instance using `validators` argument during initialization:

```

from graceful.parameters import StringParam
from graceful.validators import match_validator
from graceful.resources.generic import Resource

class ExampleResource(Resource):
    word = StringParam(
        'one "word" parameter',
        validators=[match_validator('\w+')],
    )

```

**type = 'string'**

**value** (*raw\_value*)  
Return param value as-is (str).

## graceful.serializers module

**class** graceful.serializers.**BaseSerializer**

Bases: object

Base serializer class for describing internal object serialization.

Example:

```
from graceful.serializers import BaseSerializer
from graceful.fields import RawField, IntField, FloatField

class CatSerializer(BaseSerializer):
    species = RawField("non normalized cat species")
    age = IntField("cat age in years")
    height = FloatField("cat height in cm")
```

**describe** ()

Describe all serialized fields.

It returns dictionary of all fields description defined for this serializer using their own `describe()` methods with respect to order in which they are defined as class attributes.

**Returns** *OrderedDict* – serializer description

**fields**

Return dictionary of field definition objects of this serializer.

**from\_representation** (*representation*)

Convert given representation dict into internal object.

Internal object is simply a dictionary of values with respect to field sources.

This does not check if all required fields exist or values are valid in terms of value validation (see: `BaseField.validate()`) but still requires all of passed representation values to be well formed representation (success call to `field.from_representation`).

In case of malformed representation it will run additional validation only to provide a full detailed exception about all that might be wrong with provided representation.

**Parameters** **representation** (*dict*) – dictionary with field representation values

**Raises** `DeserializationError` – when at least one representation field is not formed as expected by field object. Information about additional forbidden/missing/invalid fields is provided as well.

**get\_attribute** (*obj, attr*)

Get attribute of given object instance.

Reason for existence of this method is the fact that ‘attribute’ can be also object’s key from if is a dict or any other kind of mapping.

Note: it will return None if attribute key does not exist

**Parameters** **obj** (*object*) – internal object to retrieve data from

**Returns** internal object’s key value or attribute



**set\_attribute** (*obj*, *attr*, *value*)

Set value of attribute in given object instance.

Reason for existence of this method is the fact that ‘attribute’ can be also a object’s key if it is a dict or any other kind of mapping.

#### Parameters

- **obj** (*object*) – object instance to modify
- **attr** (*str*) – attribute (or key) to change
- **value** – value to set

**to\_representation** (*obj*)

Convert given internal object instance into representation dict.

Representation dict may be later serialized to the content-type of choice in the resource HTTP method handler.

This loops over all fields and retrieves source keys/attributes as field values with respect to optional field sources and converts each one using `field.to_representation()` method.

**Parameters** **obj** (*object*) – internal object that needs to be represented

**Returns** *dict* – representation dictionary

**validate** (*object\_dict*, *partial=False*)

Validate given internal object returned by `to_representation()`.

Internal object is validated against missing/forbidden/invalid fields values using fields definitions defined in serializer.

#### Parameters

- **object\_dict** (*dict*) – internal object dictionary to perform to validate
- **partial** (*bool*) – if set to True then incomplete object\_dict is accepted and will not raise any exceptions when one of fields is missing

**Raises** `DeserializationError`

**class** `graceful.serializers.MetaSerializer`

Bases: `type`

Metaclass for handling serialization with field objects.

**static** `__new__` (*mcs*, *name*, *bases*, *namespace*)

Create new class object instance and alter its namespace.

**classmethod** `__prepare__` (*mcs*, *name*, *bases*, *\*\*kwargs*)

Prepare class namespace in a way that ensures order of attributes.

This needs to be an `OrderedDict` so `_get_fields()` method can construct fields storage that preserves the same order of fields as defined in code.

Note: this is python3 thing and support for ordering of params in descriptions will not be backported to python2 even if this framework will get python2 support.

## graceful.validators module

`graceful.validators.min_validator` (*min\_value*)

Return validator function that ensures lower bound of a number.

Result validation function will validate the internal value of resource instance field with the value `>= min_value` check

**Parameters** `min_value` – minimal value for new validator

`graceful.validators.max_validator(max_value)`

Return validator function that ensures upper bound of a number.

Result validation function will validate the internal value of resource instance field with the value `>= min_value` check.

**Parameters** `max_value` – maximum value for new validator

`graceful.validators.choices_validator(choices)`

Return validator function that will check if value in choices.

**Parameters** `max_value` (`list`, `set`, `tuple`) – allowed choices for new validator

`graceful.validators.match_validator(expression)`

Return validator function that will check if matches given expression.

**Parameters** `match` – if string then this will be converted to regular expression using `re.compile`. Can be also any object that has `match()` method like already compiled regular expression or custom matching object/class.

## **graceful.errors module**

**exception** `graceful.errors.DeserializationError` (*missing=None, forbidden=None, invalid=None, failed=None*)

Bases: `ValueError`

Raised when error happened during deserialization of representation.

**as\_bad\_request()**

Translate this error to falcon's HTTP specific error exception.

**exception** `graceful.errors.ValidationError`

Bases: `ValueError`

Raised when validation error occurred.

**as\_bad\_request()**

Translate this error to falcon's HTTP specific error exception.

---

**Note:** Exceptions returned by this method should be used to inform about resource validation failures. In case of param validation failures the `as_invalid_param()` method should be used.

---

**as\_invalid\_param(param\_name)**

Translate this error to falcon's HTTP specific error exception.

---

**Note:** Exceptions returned by this method should be used to inform about param validation failures. In case of resource validation failures the `as_bad_request()` method should be used.

---

**Parameters** `param_name` (*str*) – HTTP query string parameter name

## graceful.resources package

### graceful.resources.base module

**class** `graceful.resources.base.BaseResource`

Bases: `object`

Base resource class with core param and response functionality.

This base class handles resource responses, parameter deserialization, and validation of request included representations if serializer is defined.

**allowed\_methods** ()

Return list of allowed HTTP methods on this resource.

This is only for purpose of making resource description.

**Returns** *list* – list of allowed HTTP method names (uppercase)

**describe** (*req=None, resp=None, \*\*kwargs*)

Describe API resource using resource introspection.

Additional description on derived resource class can be added using keyword arguments and calling `super().describe()` method call like following:

```
class SomeResource(BaseResource):
    def describe(req, resp, **kwargs):
        return super().describe(
            req, resp, type='list', **kwargs
        )
```

#### Parameters

- **req** (*falcon.Request*) – request object
- **resp** (*falcon.Response*) – response object
- **kwargs** (*dict*) – dictionary of values created from resource url template

**Returns** *dict* – dictionary with resource description information

**make\_body** (*resp, params, meta, content*)

Construct response body in *resp* object using JSON serialization.

#### Parameters

- **resp** (*falcon.Response*) – response object where to include serialized body
- **params** (*dict*) – dictionary of parsed parameters
- **meta** (*dict*) – dictionary of metadata to be included in ‘meta’ section of response
- **content** (*dict*) – dictionary of response content (resource representation) to be included in ‘content’ section of response

**Returns** `None`

**on\_options** (*req, resp, \*\*kwargs*)

Respond with JSON formatted resource description on OPTIONS request.

#### Parameters

- **req** (*falcon.Request*) – Optional request object. Defaults to `None`.

- **resp** (*falcon.Response*) – Optional response object. Defaults to None.
- **kwargs** (*dict*) – Dictionary of values created by falcon from resource uri template.

**Returns** None

**params**

Return dictionary of parameter definition objects.

**require\_meta\_and\_content** (*content\_handler, params, \*\*kwargs*)

Require ‘meta’ and ‘content’ dictionaries using proper handler.

**Parameters**

- **content\_handler** (*callable*) – function that accepts *params*, *meta*, *\*\*kwargs* argument and returns dictionary for content response section
- **params** (*dict*) – dictionary of parsed resource parameters
- **kwargs** (*dict*) – dictionary of values created from resource url template

**Returns**

*tuple (meta, content)* –

**two-tuple with dictionaries of meta and** *content* response sections

**require\_params** (*req*)

Require all defined parameters from request query string.

Raises *falcon.errors.HTTPMissingParam* exception if any of required parameters is missing and *falcon.errors.HTTPInvalidParam* if any of parameters could not be understood (wrong format).

**Parameters** **req** (*falcon.Request*) – request object

**require\_representation** (*req*)

Require raw representation dictionary from falcon request object.

This does not perform any field parsing or validation but only uses allowed content-encoding handler to decode content body.

---

**Note:** Currently only JSON is allowed as content type.

---

**Parameters** **req** (*falcon.Request*) – request object

**Returns** *dict* – raw dictionary of representation supplied in request body

**require\_validated** (*req, partial=False*)

Require fully validated internal object dictionary.

Internal object dictionary creation is based on content-decoded representation retrieved from request body. Internal object validation is performed using resource serializer.

**Parameters**

- **req** (*falcon.Request*) – request object
- **partial** (*bool*) – self to True if partially complete representation is accepted (e.g. for patching instead of full update). Missing fields in representation will be skipped.

**Returns**

*dict* –

**dictionary of fields and values representing internal object.** Each value is a result of `field.from_representation` call.

**serializer = None**

**class** `graceful.resources.base.MetaResource`

Bases: `type`

Metaclass for handling parametrization with parameter objects.

**static** `__new__ (mcs, name, bases, namespace)`

Create new class object instance and alter its namespace.

**classmethod** `__prepare__ (mcs, name, bases, **kwargs)`

Prepare class namespace in a way that ensures order of attributes.

This needs to be an *OrderedDict* so `_get_params()` method can construct params storage that preserves the same order of parameters as defined in code.

Note: this is python3 thing and support for ordering of params in descriptions will not be backported to python2 even if this framework will get python2 support.

#### Parameters

- **bases** – all base classes of created resource class
- **namespace** (*dict*) – namespace as dictionary of attributes

### graceful.resources.generic module

**class** `graceful.resources.generic.ListAPI`

Bases: `graceful.resources.mixins.ListMixin`, `graceful.resources.base.BaseResource`

Generic List API with resource serialization.

Generic resource that uses serializer for resource description, serialization and validation.

Allowed methods:

- GET: list multiple resource instances representations (handled with `.list()` method handler)

**describe** (*req=None, resp=None, \*\*kwargs*)

Extend default endpoint description with serializer description.

**on\_get** (*req, resp, \*\*kwargs*)

Respond on GET requests using `self.list()` handler.

**class** `graceful.resources.generic.ListCreateAPI`

Bases: `graceful.resources.mixins.CreateMixin`, `graceful.resources.generic.ListAPI`

Generic List/Create API with resource serialization.

Generic resource that uses serializer for resource description, serialization and validation.

Allowed methods:

- GET: list multiple resource instances representations (handled with `.list()` method handler)
- POST: create new resource from representation provided in request body (handled with `.create()` method handler)

**on\_post** (*req, resp, \*\*kwargs*)

Respond on POST requests using `self.create()` handler.

**class** `graceful.resources.generic.ListResource`

Bases: `graceful.resources.mixins.ListMixin`, `graceful.resources.base.BaseResource`

Basic retrieval of resource instance lists without serialization.

This resource class is intended for endpoints that do not require automatic representation serialization and extensive field descriptions but still gives support for defining parameters as resource class attributes.

Example usage:

**class** `graceful.resources.generic.PaginatedListAPI`

Bases: `graceful.resources.mixins.PaginatedMixin`, `graceful.resources.generic.ListAPI`

Generic List API with resource serialization and pagination.

Generic resource that uses serializer for resource description, serialization and validation.

Adds simple pagination to list of resources.

Allowed methods:

- GET: list multiple resource instances representations (handled with `.list()` method handler)

**class** `graceful.resources.generic.PaginatedListCreateAPI`

Bases: `graceful.resources.mixins.PaginatedMixin`, `graceful.resources.generic.ListCreateAPI`

Generic List/Create API with resource serialization and pagination.

Generic resource that uses serializer for resource description, serialization and validation.

Adds simple pagination to list of resources.

Allowed methods:

- GET: list multiple resource instances representations (handled with `.list()` method handler)
- POST: create new resource from representation provided in request body (handled with `.create()` method handler)

**class** `graceful.resources.generic.Resource`

Bases: `graceful.resources.mixins.RetrieveMixin`, `graceful.resources.base.BaseResource`

Basic retrieval of resource instance lists without serialization.

This resource class is intended for endpoints that do not require automatic representation serialization and extensive field descriptions but still gives support for defining parameters as resource class attributes.

Example usage:

**class** `graceful.resources.generic.RetrieveAPI`

Bases: `graceful.resources.mixins.RetrieveMixin`, `graceful.resources.base.BaseResource`

Generic Retrieve API with resource serialization.

Generic resource that uses serializer for resource description, serialization and validation.

Allowed methods:

- GET: retrieve resource representation (handled with `.retrieve()` method handler)

**describe** (*req=None, resp=None, \*\*kwargs*)

Extend default endpoint description with serializer description.

**on\_get** (*req, resp, \*\*kwargs*)  
Respond on GET requests using `self.retrieve()` handler.

**serializer** = None

**class** `graceful.resources.generic.RetrieveUpdateAPI`

Bases: `graceful.resources.mixins.UpdateMixin`, `graceful.resources.generic.RetrieveAPI`

Generic Retrieve/Update API with resource serialization.

Generic resource that uses serializer for resource description, serialization and validation.

Allowed methods:

- GET: retrieve resource representation handled with `.retrieve()` method handler
- PUT: update resource with representation provided in request body (handled with `.update()` method handler)

**on\_put** (*req, resp, \*\*kwargs*)  
Respond on PUT requests using `self.update()` handler.

**class** `graceful.resources.generic.RetrieveUpdateDeleteAPI`

Bases: `graceful.resources.mixins.DeleteMixin`, `graceful.resources.generic.RetrieveUpdateAPI`

Generic Retrieve/Update/Delete API with resource serialization.

Generic resource that uses serializer for resource description, serialization and validation.

Allowed methods:

- GET: retrieve resource representation (handled with `.retrieve()` method handler)
- PUT: update resource with representation provided in request body (handled with `.update()` method handler)
- DELETE: delete resource (handled with `.delete()` method handler)

## graceful.resources.mixins module

**class** `graceful.resources.mixins.BaseMixin`

Bases: object

Base mixin class.

**handle** (*handler, req, resp, \*\*kwargs*)  
Handle given resource manipulation flow in consistent manner.

This mixin is intended to be used only as a base class in new flow mixin classes. It ensures that regardless of resource manipulation semantics (retrieve, get, delete etc.) the flow is always the same:

- 1.Decode and validate all request parameters from the query string using `self.require_params()` method.
- 2.Use `self.require_meta_and_content()` method to construct meta and content dictionaries that will be later used to create serialized response body.
- 3.Construct serialized response body using `self.body()` method.

### Parameters

- **handler** (*method*) – resource manipulation method handler.

- **req** (*falcon.Request*) – request object instance.
- **resp** (*falcon.Response*) – response object instance to be modified.
- **\*\*kwargs** – additional keyword arguments retrieved from url template.

**Returns** Content dictionary (preferably resource representation).

**class** `graceful.resources.mixins.CreateMixin`

Bases: `graceful.resources.mixins.BaseMixin`

Add default “creation flow on POST” to any resource class.

**create** (*params, meta, \*\*kwargs*)

Create new resource instance and return its representation.

This is default resource instance creation method. Value returned is the representation of single resource instance. It will be included in the ‘content’ section of response body.

#### Parameters

- **params** (*dict*) – dictionary of parsed parameters accordingly to definitions provided as resource class attributes.
- **meta** (*dict*) – dictionary of meta parameters anything added to this dict will be later included in response ‘meta’ section. This can already be prepopulated by method that calls this handler.
- **kwargs** (*dict*) – dictionary of values retrieved from route url template by falcon. This is suggested way for providing resource identifiers.

**Returns** value to be included in response ‘content’ section

**get\_object\_location** (*obj*)

Return location URI associated with given resource representation.

This handler is optional. Returned URI will be included as the value of `Location` header on POST responses.

**on\_post** (*req, resp, handler=None, \*\*kwargs*)

Respond on POST HTTP request assuming resource creation flow.

This request handler assumes that POST requests are associated with resource creation. Thus default flow for such requests is:

- Create new resource instance and prepare its representation by calling its creation method handler.
- Try to retrieve URI of newly created object using `self.get_object_location()`. If it succeeds use that URI as the value of `Location` header in response object instance.
- Set response status code to 201 `Created`.

#### Parameters

- **req** (*falcon.Request*) – request object instance.
- **resp** (*falcon.Response*) – response object instance to be modified
- **handler** (*method*) – creation method handler to be called. Defaults to `self.create`.
- **\*\*kwargs** – additional keyword arguments retrieved from url template.

**class** `graceful.resources.mixins.DeleteMixin`

Bases: `graceful.resources.mixins.BaseMixin`



Add default “delete flow on DELETE” to any resource class.

**delete** (*params*, *meta*, *\*\*kwargs*)

Delete existing resource instance.

#### Parameters

- **params** (*dict*) – dictionary of parsed parameters accordingly to definitions provided as resource class attributes.
- **meta** (*dict*) – dictionary of meta parameters anything added to this dict will be later included in response ‘meta’ section. This can already be prepopulated by method that calls this handler.
- **\*\*kwargs** – dictionary of values retrieved from route url template by falcon. This is suggested way for providing resource identifiers.

**Returns** value to be included in response ‘content’ section

**on\_delete** (*req*, *resp*, *handler=None*, *\*\*kwargs*)

Respond on DELETE HTTP request assuming resource deletion flow.

This request handler assumes that DELETE requests are associated with resource deletion. Thus default flow for such requests is:

- Delete existing resource instance.
- Set response status code to 202 Accepted.

#### Parameters

- **req** (*falcon.Request*) – request object instance.
- **resp** (*falcon.Response*) – response object instance to be modified
- **handler** (*method*) – deletion method handler to be called. Defaults to *self.delete*.
- **\*\*kwargs** – additional keyword arguments retrieved from url template.

**class** `graceful.resources.mixins.ListMixin`

Bases: `graceful.resources.mixins.BaseMixin`

Add default “list flow on GET” to any resource class.

**list** (*params*, *meta*, *\*\*kwargs*)

List existing resource instances and return their representations.

Value returned by this handler will be included in response ‘content’ section.

#### Parameters

- **params** (*dict*) – dictionary of parsed parameters accordingly to definitions provided as resource class attributes.
- **meta** (*dict*) – dictionary of meta parameters anything added to this dict will be later included in response ‘meta’ section. This can already be prepopulated by method that calls this handler.
- **\*\*kwargs** – dictionary of values retrieved from route url template by falcon. This is suggested way for providing resource identifiers.

**Returns** value to be included in response ‘content’ section

**on\_get** (*req, resp, handler=None, \*\*kwargs*)

Respond on GET HTTP request assuming resource list retrieval flow.

This request handler assumes that GET requests are associated with resource list retrieval. Thus default flow for such requests is:

- Retrieve list of existing resource instances and prepare their representations by calling list retrieval method handler.

#### Parameters

- **req** (*falcon.Request*) – request object instance.
- **resp** (*falcon.Response*) – response object instance to be modified
- **handler** (*method*) – list method handler to be called. Defaults to `self.list`.
- **\*\*kwargs** – additional keyword arguments retrieved from url template.

**class** `graceful.resources.mixins.PaginatedMixin`

Bases: `graceful.resources.base.BaseResource`

Add simple pagination capabilities to resource.

This class provides two additional parameters with some default descriptions and `add_pagination_meta` method that can update meta with more useful pagination information.

Example usage:

```
from graceful.resources.mixins import PaginatedMixin
from graceful.resources.generic import ListResource

class SomeResource(PaginatedMixin, ListResource):

    def list(self, params, meta):
        # params has now 'page' and 'page_size' params that
        # can be used for offset&limit-like operations
        self.add_pagination_meta(params, meta)

        # ...
```

**add\_pagination\_meta** (*params, meta*)

Extend default meta dictionary value with pagination hints.

---

**Note:** This method handler attaches values to `meta` dictionary without changing it's reference. This means that you should never replace `meta` dictionary with any other dict instance but simply modify its content.

---

#### Parameters

- **params** (*dict*) – dictionary of decoded parameter values
- **meta** (*dict*) – dictionary of meta values attached to response

**class** `graceful.resources.mixins.RetrieveMixin`

Bases: `graceful.resources.mixins.BaseMixin`

Add default “retrieve flow on GET” to any resource class.

**on\_get** (*req, resp, handler=None, \*\*kwargs*)

Respond on GET HTTP request assuming resource retrieval flow.

This request handler assumes that GET requests are associated with single resource instance retrieval. Thus default flow for such requests is:

- Retrieve single resource instance of prepare its representation by calling retrieve method handler.

#### Parameters

- **req** (*falcon.Request*) – request object instance.
- **resp** (*falcon.Response*) – response object instance to be modified
- **handler** (*method*) – list method handler to be called. Defaults to `self.list`.
- **\*\*kwargs** – additional keyword arguments retrieved from url template.

**retrieve** (*params, meta, \*\*kwargs*)

Retrieve existing resource instance and return its representation.

Value returned by this handler will be included in response ‘content’ section.

#### Parameters

- **params** (*dict*) – dictionary of parsed parameters accordingly to definitions provided as resource class attributes.
- **meta** (*dict*) – dictionary of meta parameters anything added to this dict will be later included in response ‘meta’ section. This can already prepopulated by method that calls this handler.
- **\*\*kwargs** – dictionary of values retrieved from route url template by falcon. This is suggested way for providing resource identifiers.

**Returns** value to be included in response ‘content’ section

**class** `graceful.resources.mixins.UpdateMixin`

Bases: `graceful.resources.mixins.BaseMixin`

Add default “update flow on PUT” to any resource class.

**on\_put** (*req, resp, handler=None, \*\*kwargs*)

Respond on PUT HTTP request assuming resource update flow.

This request handler assumes that PUT requests are associated with resource update/modification. Thus default flow for such requests is:

- Modify existing resource instance and prepare its representation by calling its update method handler.
- Set response status code to 202 Accepted.

#### Parameters

- **req** (*falcon.Request*) – request object instance.
- **resp** (*falcon.Response*) – response object instance to be modified
- **handler** (*method*) – update method handler to be called. Defaults to `self.update`.
- **\*\*kwargs** – additional keyword arguments retrieved from url template.

**update** (*params*, *meta*, *\*\*kwargs*)

Update existing resource instance and return its representation.

Value returned by this handler will be included in response 'content' section.

#### Parameters

- **params** (*dict*) – dictionary of parsed parameters accordingly to definitions provided as resource class attributes.
- **meta** (*dict*) – dictionary of meta parameters anything added to this dict will be later included in response 'meta' section. This can already be prepopulated by method that calls this handler.
- **\*\*kwargs** – dictionary of values retrieved from route url template by falcon. This is suggested way for providing resource identifiers.

**Returns** value to be included in response 'content' section

## CHAPTER 2

---

### graceful

---

`graceful` is elegant Python REST toolkit built on top of `falcon`. It is highly inspired by `Django REST framework` - mostly by how object serialization is done but more emphasis here is put on API to be self-descriptive.

Features:

- generic classes for list and single object resources
- simple but extendable pagination
- structured responses with content/meta separation
- declarative fields and parameters
- self-descriptive-everything: API description accessible both in python and through `OPTIONS` requests
- painless validation
- 100% tests coverage
- `falcon`  $\geq 0.3.0$  (tested up to 1.0.x)
- python3 exclusive (tested from 3.3 to 3.5)

There is no community behind `graceful` yet but I hope we will build one someday with your help. Anyway there is a mailing list on `Librelist`. Just send an email to `graceful@librelist.com` and you're subscribed.



## CHAPTER 3

---

python3 only

---

**Important:** `graceful` is python3 exclusive because **right now** should be a good time to forget about python2. There are no plans for making `graceful` python2 compatible although it would be pretty straightforward to do so with existing tools (like `six`).





## CHAPTER 4

---

### usage

---

For extended tutorial and more information please refer to [guide](#) included in documentation.

Anyway here is simple example of working API made made with graceful:

```
import falcon

from graceful.serializers import BaseSerializer
from graceful.fields import IntField, RawField
from graceful.parameters import StringParam
from graceful.resources.generic import (
    RetrieveAPI,
    PaginatedListAPI,
)

api = application = falcon.API()

# lets pretend that this is our backend storage
CATS_STORAGE = [
    {"id": 0, "name": "kitty", "breed": "saimese"},
    {"id": 1, "name": "lucie", "breed": "maine coon"},
    {"id": 2, "name": "molly", "breed": "sphynx"},
]

# this is how we represent cats in our API
class CatSerializer(BaseSerializer):
    id = IntField("cat identification number", read_only=True)
    name = RawField("cat name")
    breed = RawField("official breed name")

class Cat(RetrieveAPI):
    """
    Single cat identified by its id
    """
```

```
serializer = CatSerializer()

def get_cat(self, cat_id):
    try:
        return [
            cat for cat in CATS_STORAGE if cat['id'] == int(cat_id)
        ][0]
    except IndexError:
        raise falcon.HTTPNotFound

def retrieve(self, params, meta, **kwargs):
    cat_id = kwargs['cat_id']
    return self.get_cat(cat_id)

class CatList(PaginatedListAPI):
    """
    List of all cats in our API
    """
    serializer = CatSerializer()

    breed = StringParam("set this param to filter cats by breed")

    def list(self, params, meta, **kwargs):
        if 'breed' in params:
            filtered = [
                cat for cat in CATS_STORAGE
                if cat['breed'] == params['breed']
            ]
            return filtered
        else:
            return CATS_STORAGE

api.add_route("/v1/cats/{cat_id}", Cat())
api.add_route("/v1/cats/", CatList())
```

Assume this code is in python module named `example.py`. Now run it with `gunicorn`:

```
gunicorn -b localhost:8888 example
```

And you're ready to query it (here with awesome `httpie` tool):

```
$ http localhost:8888/v0/cats/?breed=saimese
HTTP/1.1 200 OK
Connection: close
Date: Tue, 16 Jun 2015 08:43:05 GMT
Server: gunicorn/19.3.0
content-length: 116
content-type: application/json

{
  "content": [
    {
      "breed": "saimese",
      "id": 0,
      "name": "kitty"
    }
  ],
}
```

```

    "meta": {
      "params": {
        "breed": "saimese",
        "indent": 0
      }
    }
  }
}

```

Or access API description issuing OPTIONS request:

```

$ http OPTIONS localhost:8888/v0/cats
HTTP/1.1 200 OK
Connection: close
Date: Tue, 16 Jun 2015 08:40:00 GMT
Server: gunicorn/19.3.0
allow: GET, OPTIONS
content-length: 740
content-type: application/json

{
  "details": "List of all cats in our API",
  "fields": {
    "breed": {
      "details": "official breed name",
      "label": null,
      "spec": null,
      "type": "string"
    },
    "id": {
      "details": "cat identification number",
      "label": null,
      "spec": null,
      "type": "int"
    },
    "name": {
      "details": "cat name",
      "label": null,
      "spec": null,
      "type": "string"
    }
  },
  "methods": [
    "GET",
    "OPTIONS"
  ],
  "name": "CatList",
  "params": {
    "breed": {
      "default": null,
      "details": "set this param to filter cats by breed",
      "label": null,
      "required": false,
      "spec": null,
      "type": "string"
    },
    "indent": {
      "default": "0",
      "details": "JSON output indentation. Set to 0 if output should not be
→formatted."

```

```
        "label": null,  
        "required": false,  
        "spec": null,  
        "type": "integer"  
    },  
    },  
    "path": "/v0/cats",  
    "type": "list"  
}
```

## CHAPTER 5

---

### contributing

---

Any contribution is welcome. Issues, suggestions, pull requests - whatever. There is only short set of rules that guide this project development you should be aware of before submitting a pull request:

- Only requests that have passing CI builds (Travis) will be merged.
- Code is checked with `flake8` and `pydocstyle` during build so this implicitly means that compliance with PEP-8 and PEP-257 are mandatory.
- No changes that decrease coverage will be merged.

One thing: if you submit a PR please do not rebase it later unless you are asked for that explicitly. Reviewing pull requests that suddenly had their history rewritten just drives me crazy.



## CHAPTER 6

---

license

---

See LICENSE file.





## CHAPTER 7

---

### Indices and tables

---

- `genindex`
- `modindex`
- `search`



### g

- `graceful.errors`, [30](#)
- `graceful.fields`, [21](#)
- `graceful.parameters`, [25](#)
- `graceful.resources.base`, [31](#)
- `graceful.resources.generic`, [33](#)
- `graceful.resources.mixins`, [35](#)
- `graceful.serializers`, [28](#)
- `graceful.validators`, [29](#)



## Symbols

`__new__()` (`graceful.resources.base.MetaResource` static method), 33  
`__new__()` (`graceful.serializers.MetaSerializer` static method), 29  
`__prepare__()` (`graceful.resources.base.MetaResource` class method), 33  
`__prepare__()` (`graceful.serializers.MetaSerializer` class method), 29

## A

`add_pagination_meta()` (`graceful.resources.mixins.PaginatedMixin` method), 38  
`allowed_methods()` (`graceful.resources.base.BaseResource` method), 31  
`as_bad_request()` (`graceful.errors.DeserializationError` method), 30  
`as_bad_request()` (`graceful.errors.ValidationError` method), 30  
`as_invalid_param()` (`graceful.errors.ValidationError` method), 30

## B

`Base64EncodedParam` (class in `graceful.parameters`), 25  
`BaseField` (class in `graceful.fields`), 21  
`BaseMixin` (class in `graceful.resources.mixins`), 35  
`BaseParam` (class in `graceful.parameters`), 25  
`BaseResource` (class in `graceful.resources.base`), 31  
`BaseSerializer` (class in `graceful.serializers`), 28  
`BoolField` (class in `graceful.fields`), 23  
`BoolParam` (class in `graceful.parameters`), 26

## C

`choices_validator()` (in module `graceful.validators`), 30  
`container` (`graceful.parameters.BaseParam` attribute), 26  
`create()` (`graceful.resources.mixins.CreateMixin` method), 36

`CreateMixin` (class in `graceful.resources.mixins`), 36

## D

`DecimalParam` (class in `graceful.parameters`), 27  
`delete()` (`graceful.resources.mixins.DeleteMixin` method), 37  
`DeleteMixin` (class in `graceful.resources.mixins`), 36  
`describe()` (`graceful.fields.BaseField` method), 22  
`describe()` (`graceful.parameters.BaseParam` method), 26  
`describe()` (`graceful.resources.base.BaseResource` method), 31  
`describe()` (`graceful.resources.generic.ListAPI` method), 33  
`describe()` (`graceful.resources.generic.RetrieveAPI` method), 34  
`describe()` (`graceful.serializers.BaseSerializer` method), 28  
`DeserializationError`, 30

## F

`fields` (`graceful.serializers.BaseSerializer` attribute), 28  
`FloatField` (class in `graceful.fields`), 23  
`FloatParam` (class in `graceful.parameters`), 27  
`from_representation()` (`graceful.fields.BaseField` method), 22  
`from_representation()` (`graceful.fields.BoolField` method), 23  
`from_representation()` (`graceful.fields.FloatField` method), 24  
`from_representation()` (`graceful.fields.IntField` method), 24  
`from_representation()` (`graceful.fields.RawField` method), 24  
`from_representation()` (`graceful.fields.StringField` method), 24  
`from_representation()` (`graceful.serializers.BaseSerializer` method), 28

## G

`get_attribute()` (`graceful.serializers.BaseSerializer`

method), 28  
get\_object\_location() (graceful.resources.mixins.CreateMixin method), 36  
graceful.errors (module), 30  
graceful.fields (module), 21  
graceful.parameters (module), 25  
graceful.resources.base (module), 31  
graceful.resources.generic (module), 33  
graceful.resources.mixins (module), 35  
graceful.serializers (module), 28  
graceful.validators (module), 29

## H

handle() (graceful.resources.mixins.BaseMixin method), 35

## I

IntField (class in graceful.fields), 24  
IntParam (class in graceful.parameters), 27

## L

list() (graceful.resources.mixins.ListMixin method), 37  
ListAPI (class in graceful.resources.generic), 33  
ListCreateAPI (class in graceful.resources.generic), 33  
ListMixin (class in graceful.resources.mixins), 37  
ListResource (class in graceful.resources.generic), 34

## M

make\_body() (graceful.resources.base.BaseResource method), 31  
match\_validator() (in module graceful.validators), 30  
max\_validator() (in module graceful.validators), 30  
MetaResource (class in graceful.resources.base), 33  
MetaSerializer (class in graceful.serializers), 29  
min\_validator() (in module graceful.validators), 29

## O

on\_delete() (graceful.resources.mixins.DeleteMixin method), 37  
on\_get() (graceful.resources.generic.ListAPI method), 33  
on\_get() (graceful.resources.generic.RetrieveAPI method), 34  
on\_get() (graceful.resources.mixins.ListMixin method), 37  
on\_get() (graceful.resources.mixins.RetrieveMixin method), 38  
on\_options() (graceful.resources.base.BaseResource method), 31  
on\_post() (graceful.resources.generic.ListCreateAPI method), 33  
on\_post() (graceful.resources.mixins.CreateMixin method), 36

on\_put() (graceful.resources.generic.RetrieveUpdateAPI method), 35  
on\_put() (graceful.resources.mixins.UpdateMixin method), 39

## P

PaginatedListAPI (class in graceful.resources.generic), 34  
PaginatedListCreateAPI (class in graceful.resources.generic), 34  
PaginatedMixin (class in graceful.resources.mixins), 38  
params (graceful.resources.base.BaseResource attribute), 32

## R

RawField (class in graceful.fields), 24  
require\_meta\_and\_content() (graceful.resources.base.BaseResource method), 32  
require\_params() (graceful.resources.base.BaseResource method), 32  
require\_representation() (graceful.resources.base.BaseResource method), 32  
require\_validated() (graceful.resources.base.BaseResource method), 32  
Resource (class in graceful.resources.generic), 34  
retrieve() (graceful.resources.mixins.RetrieveMixin method), 39  
RetrieveAPI (class in graceful.resources.generic), 34  
RetrieveMixin (class in graceful.resources.mixins), 38  
RetrieveUpdateAPI (class in graceful.resources.generic), 35  
RetrieveUpdateDeleteAPI (class in graceful.resources.generic), 35

## S

serializer (graceful.resources.base.BaseResource attribute), 33  
serializer (graceful.resources.generic.RetrieveAPI attribute), 35  
set\_attribute() (graceful.serializers.BaseSerializer method), 28  
spec (graceful.fields.BaseField attribute), 22  
spec (graceful.parameters.Base64EncodedParam attribute), 25  
spec (graceful.parameters.BaseParam attribute), 26  
StringField (class in graceful.fields), 24  
StringParam (class in graceful.parameters), 27

## T

to\_representation() (graceful.fields.BaseField method), 22

[to\\_representation\(\) \(graceful.fields.BoolField method\), 23](#)  
[to\\_representation\(\) \(graceful.fields.FloatField method\), 24](#)  
[to\\_representation\(\) \(graceful.fields.IntField method\), 24](#)  
[to\\_representation\(\) \(graceful.fields.RawField method\), 24](#)  
[to\\_representation\(\) \(graceful.fields.StringField method\), 25](#)  
[to\\_representation\(\) \(graceful.serializers.BaseSerializer method\), 29](#)  
[type \(graceful.fields.BaseField attribute\), 23](#)  
[type \(graceful.fields.BoolField attribute\), 23](#)  
[type \(graceful.fields.FloatField attribute\), 24](#)  
[type \(graceful.fields.IntField attribute\), 24](#)  
[type \(graceful.fields.RawField attribute\), 24](#)  
[type \(graceful.fields.StringField attribute\), 25](#)  
[type \(graceful.parameters.BaseParam attribute\), 26](#)  
[type \(graceful.parameters.BoolParam attribute\), 27](#)  
[type \(graceful.parameters.DecimalParam attribute\), 27](#)  
[type \(graceful.parameters.FloatParam attribute\), 27](#)  
[type \(graceful.parameters.IntParam attribute\), 27](#)  
[type \(graceful.parameters.StringParam attribute\), 27](#)

## U

[update\(\) \(graceful.resources.mixins.UpdateMixin method\), 39](#)  
[UpdateMixin \(class in graceful.resources.mixins\), 39](#)

## V

[validate\(\) \(graceful.fields.BaseField method\), 23](#)  
[validate\(\) \(graceful.serializers.BaseSerializer method\), 29](#)  
[validated\\_value\(\) \(graceful.parameters.BaseParam method\), 26](#)  
[ValidationError, 30](#)  
[value\(\) \(graceful.parameters.Base64EncodedParam method\), 25](#)  
[value\(\) \(graceful.parameters.BaseParam method\), 26](#)  
[value\(\) \(graceful.parameters.BoolParam method\), 27](#)  
[value\(\) \(graceful.parameters.DecimalParam method\), 27](#)  
[value\(\) \(graceful.parameters.FloatParam method\), 27](#)  
[value\(\) \(graceful.parameters.IntParam method\), 27](#)  
[value\(\) \(graceful.parameters.StringParam method\), 27](#)