Flask-RESTy

Release 4.0.2

4Catalyzer

Apr 12, 2022

CONTENTS

Flask-RESTy provides building blocks for creating REST APIs with Flask , SQLAlchemy, and marshmallow.

```
from flask_resty import Api, GenericModelView

from .models import Widget
from .schemas import WidgetSchema


class WidgetViewBase(GenericModelView):
    model = Widget
    schema = WidgetSchema()


class WidgetListView(WidgetViewBase):
    def get(self):
        return self.list()

    def post(self):
        return self.create()


class WidgetView(WidgetViewBase):
    def get(self, id):
        return self.retrieve(id)

    def patch(self, id):
        return self.update(id, partial=True)

    def delete(self, id):
        return self.destroy(id)


api = Api(app, "/api")
api.add_resource("/widgets", WidgetListView, WidgetView)
```

CHAPTER

ONE

FEATURES

Flask-RESty provides the following functionality out of the box:

- Class-based CRUD views
- Schema-based request validation and response formatting with marshmallow
- JWT and JWK authentication, with base classes for implementing your own authentication policies
- Authorization
- Sorting and pagination
- Filtering

CHAPTER

TWO

INSTALLATION

Flask-RESTy requires Python >= 3.6.

```
$ pip install flask-resty
```

For JWT support:

```
$ pip install flask-resty[jwt]
```

CHAPTER

THREE

GUIDE

3.1 Building an Application

3.1.1 Project Structure

When building applications with Flask-RESTy, we recommend starting with the following project structure.

```
example
├── __init__.py
├── settings.py  # App settings
├── models.py    # SQLAlchemy models
├── schemas.py   # marshmallow schemas
├── auth.py      # Authn and authz classes
├── views.py     # View classes
└── routes.py    # Route declarations
```

The `__init__.py` file initializes the `Flask` app and hooks up the routes.

```python
# example/__init__.py

from flask import Flask

from . import settings

app = Flask(__name__)
app.config.from_object(settings)

from . import routes  # noqa: F401 isort:skip
```

Note: `# noqa: F401 isort:skip` prevents Flake8 and isort from reporting a misplaced import.

3.1.2 Models

Models are created using Flask-SQLAlchemy .

```
# example/models.py

import datetime as dt

from flask_sqlalchemy import SQLAlchemy

from . import app

db = SQLAlchemy(app)


class Author(db.Model):
    __tablename__ = "example_authors"

    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.Text, nullable=False)
    created_at = db.Column(
        db.DateTime, default=dt.datetime.utcnow, nullable=False
    )


class Book(db.Model):
    __tablename__ = "example_books"

    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.Text, nullable=False)
    author_id = db.Column(db.Integer, db.ForeignKey(Author.id), nullable=False)
    author = db.relationship(Author, backref=db.backref("books"))
    published_at = db.Column(db.DateTime, nullable=False)
    created_at = db.Column(
        db.DateTime, default=dt.datetime.utcnow, nullable=False
    )
```

See also:

See the Flask-SQLAlchemy documentation for more information on defining models.

3.1.3 Schemas

Schemas are used to validate request input and format response outputs.

```
# example/schemas.py

from marshmallow import Schema, fields


class AuthorSchema(Schema):
    id = fields.Int(dump_only=True)
    name = fields.String(required=True)
```

(continues on next page)

(continued from previous page)

```
    created_at = fields.DateTime(dump_only=True)


class BookSchema(Schema):
    id = fields.Int(dump_only=True)
    title = fields.String(required=True)
    author_id = fields.Int(required=True)
    published_at = fields.DateTime(required=True)
    created_at = fields.DateTime(dump_only=True)
```

See also:

See the marshmallow documentation for more information on defining schemas.

3.1.4 Views

Most view classes will extend *flask_resty.GenericModelView* which provides standard CRUD behavior.

Typically, you will expose a model with a list endpoint (`/api/authors/`), and a detail endpoint (`/api/authors/<id>`). To keep your code DRY, we recommend using a common base class for both endpoints.

For example:

```
# example/views.py

from flask_resty import GenericModelView

from . import models, schemas


class AuthorViewBase(GenericModelView):
    model = models.Author
    schema = schemas.AuthorSchema()
    # authentication, authorization, pagination,
    # sorting, and filtering would also go here
```

The concrete view classes simply call the appropriate CRUD methods from *GenericModelView*.

```
class AuthorListView(AuthorViewBase):
    def get(self):
        return self.list()

    def post(self):
        return self.create()


class AuthorView(AuthorViewBase):
    def get(self, id):
        return self.retrieve(id)

    def patch(self, id):
        return self.update(id, partial=True)
```

(continues on next page)

(continued from previous page)

```
    def delete(self, id):
        return self.destroy(id)
```

Note: Unimplemented HTTP methods will return `405 Method not allowed`.

3.1.5 Pagination

Add pagination to your list endpoints by setting the `pagination` attribute on the base class.

The following will allow clients to pass a `page` parameter in the query string, e.g. `?page=2`.

```
class AuthorViewBase(GenericModelView):
    model = models.Author
    schema = schemas.AuthorSchema()

    pagination = PagePagination(page_size=10)
```

See also:

See the *Pagination* section of the API docs for a listing of available pagination classes.

3.1.6 Sorting

Add sorting to your list endpoints by setting the `sorting` attribute on the base class.

The following will allow clients to pass a `sort` parameter in the query string, e.g. `?sort=-created_at`.

```
class AuthorViewBase(GenericModelView):
    model = models.Author
    schema = schemas.AuthorSchema()

    pagination = PagePagination(page_size=10)
    sorting = Sorting("created_at", default="-created_at")
```

See also:

See the *Sorting* section of the API docs for a listing of available sorting classes.

3.1.7 Filtering

Add filtering to your list endpoints by setting the `filtering` attribute on the base class.

Filtering natively handles multiple values. Specify values in a comma separated string to the query parameter, e.g., `/books?author_id=1,2`.

```
class BookViewBase(GenericModelView):
    model = models.Book
    schema = schemas.BookSchema()
    pagination = PagePagination(page_size=10)
    sorting = Sorting("published_at", default="-published_at")
```

(continues on next page)

(continued from previous page)

```
    # An error is returned if author_id is omitted from the query string
    filtering = Filtering(author_id=ColumnFilter(operator.eq, required=True))
```

See also:

See the *Filtering* section of the API docs for a listing of available filtering classes.

3.1.8 Authentication

Add authentication by setting the `authentication` attribute on the base class. We'll use *`NoOpAuthentication`* for this example. Flask-RESTy also includes a `JwtAuthentication` class for authenticating with JSON Web Tokens .

```
class AuthorViewBase(GenericModelView):
    model = models.Author
    schema = schemas.AuthorSchema()

    authentication = NoOpAuthentication()

    pagination = PagePagination(page_size=10)
    sorting = Sorting("created_at", default="-created_at")
```

See also:

See the *Authentication* section of the API docs for a listing of available authentication classes.

3.1.9 Authorization

Add authorization by setting the `authorization` attribute on the base class. We'll use *`NoOpAuthorization`* for this example. You will likely need to implement your own subclasses of *`AuthorizationBase`* for your applications.

```
class AuthorViewBase(GenericModelView):
    model = models.Author
    schema = schemas.AuthorSchema()

    authentication = NoOpAuthentication()
    authorization = NoOpAuthorization()

    pagination = PagePagination(page_size=10)
    sorting = Sorting("created_at", default="-created_at")
```

See also:

See the *Authorization* section of the API docs for a listing of available authorization classes.

3.1.10 Routes

The `routes.py` file contains the *Api* instance with which we can connect our view classes to URL patterns.

```
# example/routes.py

from flask_resty import Api

from . import app, views

api = Api(app, prefix="/api")

api.add_resource("/authors/", views.AuthorListView, views.AuthorView)
api.add_resource("/books/", views.BookListView, views.BookView)
api.add_ping("/ping/")
```

3.1.11 Testing

Flask-RESTy includes utilities for writing integration tests for your applications. Here's how you can use them with pytest.

```
from unittest.mock import ANY

import pytest

from flask_resty.testing import ApiClient, assert_response, assert_shape

from . import app


@pytest.fixture(scope="session")
def db():
    app.config["SQLALCHEMY_DATABASE_URI"] = "sqlite:///:memory:"
    database = app.extensions["sqlalchemy"].db
    database.create_all()
    return database


@pytest.fixture(autouse=True)
def clean_tables(db):
    for table in reversed(db.metadata.sorted_tables):
        db.session.execute(table.delete())

    db.session.commit()
    yield
    db.session.rollback()


@pytest.fixture
def client(monkeypatch):
    monkeypatch.setattr(app, "testing", True)
    monkeypatch.setattr(app, "test_client_class", ApiClient)
    return app.test_client()
```

The first two fixtures ensure that we start with a clean database for each test. The third fixture constructs an *ApiClient* for sending requests within our tests.

Let's test that we can create an author. Here we use *assert_response* to check the status code of the response and *assert_shape* to validate the shape of the response body.

```
def test_create_author(client):
    response = client.post("/authors/", data={"name": "Fred Brooks"})
    data = assert_response(response, 201)
    assert_shape(data, {"id": ANY, "name": "Fred Brooks", "created_at": ANY})
```

We can test both the response code and the data shape using a single call. The following snippet is equivalent to the above.

```
def test_create_author(client):
    response = client.post("/authors/", data={"name": "Fred Brooks"})
    data = assert_response(response, 201)
    assert_response(
        response, 201, {"id": ANY, "name": "Fred Brooks", "created_at": ANY}
    )
```

3.1.12 Running the Example Application

To run the example application, clone the Flask-RESTy repo.

```
$ git clone https://github.com/4Catalyzer/flask-resty.git
$ cd flask-resty
```

Populate the database with some dummy data.

```
$ python -m example.populate_db
```

Then serve the app on `localhost:5000`.

```
$ FLASK_APP=example FLASK_ENV=development flask run
```

You can make requests using the httpie utility.

```
$ pip install httpie
```

```
$ http ":5000/api/books/?author_id=2"
HTTP/1.0 200 OK
Content-Length: 474
Content-Type: application/json
Date: Sun, 16 Jun 2019 01:39:04 GMT
Server: Werkzeug/0.14.1 Python/3.7.3

{
    "data": [
        {
            "author_id": 2,
            "created_at": "2019-06-16T01:09:33.450768",
            "id": 2,
```

(continues on next page)

(continued from previous page)

```
            "published_at": "2013-11-05T00:00:00",
            "title": "The Design of Everyday Things"
        },
        {
            "author_id": 2,
            "created_at": "2019-06-16T01:09:33.450900",
            "id": 3,
            "published_at": "2010-10-29T00:00:00",
            "title": "Living With Complexity"
        }
    ],
    "meta": {
        "has_next_page": false
    }
}
```

The naive datetimes in the response are only because the example uses SQLite. A real application would use a timezone-aware datetime column in the database, and would have a UTC offset in the response.

3.1.13 Running the Shell

Flask-RESTy includes an enhanced `flask shell` command that automatically imports all SQLAlchemy models and marshmallow schemas. It will also automatically use IPython, BPython, or ptpython if they are installed.

```
$ FLASK_APP=example flask shell
3.8.5 (default, Jul 24 2020, 12:48:45)
[Clang 11.0.3 (clang-1103.0.32.62)]

_____ _            _          ____  _____ ____ _____
|  ___| | __ _ ___| | __     |  _ \| ____/ ___|_   _|   _
| |_  | |/ _` / __| |/ /____| |_) |  _| \___ \ | || | | |
|  _| | | (_| \__ \   <_____|  _ <| |___ ___) || || |_| |
|_|   |_|\__,_|___/_|\_\    |_| \_\_____|____/ |_| \__, |
                                                     |___/
Flask app: example, Database: sqlite:///example.db

Flask:
app, g

Schemas:
AuthorSchema, BookSchema, Schema

Models:
Author, Book, commit, db, flush, rollback, session

In [1]: Author
Out[1]: example.models.Author

In [2]: AuthorSchema
Out[2]: example.schemas.AuthorSchema
```

Note: Pass the `--sqlalchemy-echo` option to see database queries printed within your shell session.

The following app configuration options are available for customizing `flask shell`:

- `RESTY_SHELL_CONTEXT`: Dictionary of additional variables to include in the shell context.
- `RESTY_SHELL_LOGO`: Custom logo.
- `RESTY_SHELL_PROMPT`: Custom input prompt.
- `RESTY_SHELL_OUTPUT`: Custom output prompt.
- `RESTY_SHELL_SETUP`: Additional shell setup, a function.
- `RESTY_SHELL_CONTEXT_FORMAT`: Format to display shell context. May be `'full'`, `'short'`, or a function that receives the context dictionary as input and returns a string.
- `RESTY_SHELL_IPY_AUTORELOAD`: Whether to load and enable the IPython autoreload extension (must be using `ipython` shell).
- `RESTY_SHELL_IPY_EXTENSIONS`: List of IPython extension names to load (must be using `ipython` shell).
- `RESTY_SHELL_IPY_COLORS`: IPython color style.
- `RESTY_SHELL_IPY_HIGHLIGHTING_STYLE`: IPython code highlighting style.
- `RESTY_SHELL_PTPY_VI_MODE`: Enable vi mode (must be using `ptpython` shell).

3.2 Special Topics

3.2.1 Customizing CRUD behavior

You can customize how your model objects are created, updated, and deleted by overriding the `(create|update|delete)_item` and `(create|update|delete)_item_raw` hooks on *ModelView*.

For example if you want to soft-delete models by setting the `deleted_at` column, you can override `delete_item` and `delete_item_raw`.

```
class SoftDeleteMixin(ModelView):
    def delete_item(self, item):
        if item.deleted_at:
            flask.abort(404)

        super().delete_item(item)

    def delete_item_raw(self, item):
        item.deleted_at = datetime.datetime.now(timezone.utc)
```

3.2.2 Usage with marshmallow-sqlalchemy

Schemas may be generated with marshmallow-sqlalchemy . You may generate schemas from your models using `SQLAlchemySchema` or `SQLAlchemyAutoSchema`. **Do not use ModelSchema**.

```
from marshmallow_sqlalchemy import SQLAlchemySchema

from . import models


class AuthorSchema(SQLAlchemySchema):
    class Meta:
        model = models.Author
        include_fk = True
```

You can also use the `auto_field` helper.

```
from marshmallow import Schema
from marshmallow_sqlalchemy import SQLAlchemySchema

from . import models


class BookSchema(SQLAlchemySchema):
    class Meta:
        model = models.Book

    id = auto_field()
    author_id = auto_field()
    published_at = auto_field()
    created_at = auto_field(dump_only=True)
```

3.2.3 Filter-only Fields

If you have a field that should only be used for filtering, you can set both `load_only` and `dump_only` to `True` on the schema field.

```
class SoftDeletedObjectSchema(Schema):
    # Only used for filtering
    is_deleted = fields.Boolean(load_only=True, dump_only=True)
```

Filter-only fields will be validated when used as a filter but will not be returned in the response body.

3.2.4 Recommendations for Larger Applications

Todo: Subpackages

3.2.5 Loading Related Data

Todo: Document `Related`

3.2.6 Pre-fetching Data

Todo: Document `base_query_options` and `Schema.get_query_options`

CHAPTER

FOUR

API REFERENCE

4.1 API

4.1.1 Router

class flask_resty.**Api**(*app=None*, *prefix=''*)

The Api object controls the Flask-RESTy extension.

This can either be bound to an individual Flask application passed in at initialization, or to multiple applications via *init_app()*.

After initializing an application, use this object to register resources with *add_resource()*, either to the bound application by default or to an explicitly-specified application object via the `app` keyword argument. Use *add_ping()* to add a ping endpoint for health checks.

Once registered, Flask-RESTy will convert all HTTP errors thrown by the application to JSON.

By default your API will be rooted at '/'. Pass `prefix` to specify a custom root.

Parameters

- **app** (*flask.Flask*) – The Flask application object.
- **prefix** (*str*) – The API path prefix.

init_app(*app*)

Initialize an application for use with Flask-RESTy.

Parameters **app** (*flask.Flask*) – The Flask application object.

add_resource(*base_rule*, *base_view*, *alternate_view=None*, ***, *alternate_rule=None*, *id_rule=None*, *app=None*)

Add a REST resource.

Parameters

- **base_rule** (*str*) – The URL rule for the resource. This will be prefixed by the API prefix.
- **base_view** (*ApiView*) – Class-based view for the resource.
- **alternate_view** (*ApiView*) – If specified, an alternate class-based view for the resource. Usually, this will be a detail view, when the base view is a list view.
- **alternate_rule** (*str or None*) – If specified, the URL rule for the alternate view. This will be prefixed by the API prefix. This is mutually exclusive with `id_rule`, and must not be specified if `alternate_view` is not specified.

- **id_rule** (*str or None*) – If specified, a suffix to append to `base_rule` to get the alternate view URL rule. If `alternate_view` is specified, and `alternate_rule` is not, then this defaults to '<id>'. This is mutually exclusive with `alternate_rule`, and must not be specified if `alternate_view` is not specified.
- **app** (`flask.Flask`) – If specified, the application to which to add the route(s). Otherwise, this will be the bound application, if present.

Raises `AssertionError` – If no Flask application is bound or specified.

add_ping(*rule*, *, *status_code=200*, *app=None*)

Add a ping route.

Parameters

- **rule** (*str*) – The URL rule. This will not use the API prefix, as the ping endpoint is not really part of the API.
- **status_code** (*int*) – The ping response status code. The default is 200 rather than the more correct 204 because many health checks look for 200s.
- **app** (`flask.Flask`) – If specified, the application to which to add the route. Otherwise, this will be the bound application, if present.

Raises `AssertionError` – If no Flask application is bound or specified.

4.1.2 Views

class flask_resty.**ApiView**

Base class for views that expose API endpoints.

ApiView extends `flask.views.MethodView` exposes functionality to deserialize request bodies and serialize response bodies according to standard API semantics.

schema = None

The `marshmallow.Schema` for serialization and deserialization.

id_fields = ('id',)

The identifying fields for the model.

args_schema = None

The `marshmallow.Schema` for deserializing the query params in the `flask.Request.args`.

authentication = <flask_resty.authentication.NoOpAuthentication object>

The authentication component. See *AuthenticationBase*.

authorization = <flask_resty.authorization.NoOpAuthorization object>

The authorization component. See *AuthorizationBase*.

dispatch_request(**args*, ***kwargs*)

Handle an incoming request.

By default, this checks request-level authentication and authorization before calling the upstream request handler.

serialize(*item*, ***kwargs*)

Dump an item using the *serializer*.

This doesn't technically serialize the item; it instead uses marshmallow to dump the item into a native Python data type. The actual serialization is done in *make_response*.

Any provided **kwargs will be passed to marshmallow.Schema.dump().

Parameters **item** (*object*) – The object to serialize

Returns The serialized object

Return type dict

serializer

The marshmallow.Schema for serialization.

By default, this is *ApiView.schema*. This can be overridden to use a different schema for serialization.

make_items_response(*items*, **args*)

Build a response for a sequence of multiple items.

This serializes the items, then builds an response with the list of serialized items as its data.

This is useful when returning a list of items.

The response will have the items available as the **items** attribute.

Parameters **items** (*list*) – The objects to serialize into the response body.

Returns The HTTP response

Return type flask.Response

make_item_response(*item*, **args*)

Build a response for a single item.

This serializes the item, then builds an response with the serialized item as its data. If the response status code is 201, then it will also include a **Location** header with the canonical URL of the item, if available.

The response will have the item available as the **item** attribute.

Parameters **item** (*object*) – The object to serialize into the response body.

Returns The HTTP response

Return type flask.Response

set_item_response_meta(*item*)

Hook for setting additional metadata for an item.

This should call **meta.update_response_meta** to set any metadata values to add to the response.

Parameters **item** (*object*) – The object for which to generate metadata.

Returns

make_response(*data*, **args*, ***kwargs*)

Build a response for arbitrary dumped data.

This builds the response body given the data and any metadata from the request context. It then serializes the response.

Returns The HTTP response

Return type flask.Response

render_response_body(*data*, *response_meta*)

Render the response data and metadata into a body.

This is the final step of building the response payload before serialization.

By default, this builds a dictionary with a `data` item for the response data and a `meta` item for the response metadata, if any.

`make_raw_response`(**args*, ***kwargs*)

Convenience method for creating a `flask.Response`.

Any supplied keyword arguments are defined as attributes on the response object itself.

Returns The HTTP response

Return type `flask.Response`

`make_empty_response`(***kwargs*)

Build an empty response.

This response has a status code of 204 and an empty body.

Returns The HTTP response

Return type `flask.Response`

`make_created_response`(*item*)

Build a response for a newly created item.

This response will be for the item data and will have a status code of 201. It will include a `Location` header with the canonical URL of the created item, if available.

Parameters `item` (*object*) – The created item.

Returns The HTTP response

Return type `flask.Response`

`make_deleted_response`(*item*)

Build a response for a deleted item.

By default, this will be an empty response. The empty response will have the `item` attribute as with an item response.

Parameters `item` (*object*) – The deleted item.

Returns The HTTP response

Return type `flask.Response`

`get_location`(*item*)

Get the canonical URL for an item.

Override this to return `None` if no such URL is available.

Parameters `item` (*object*) – The item.

Returns The canonical URL for `item`.

Return type str

`get_request_data`(***kwargs*)

Deserialize and load data from the body of the current request.

By default, this will look for the value under the `data` key in a JSON request body.

Returns The deserialized request data

Return type dict

parse_request_data()

Deserialize the data for the current request.

This will deserialize the request data from the request body into a native Python object that can be loaded by marshmallow.

Returns The deserialized request data.

deserialize(*data_raw*, ***, *expected_id=None*, ***kwargs*)

Load data using the *deserializer*.

This doesn't technically deserialize the data; it instead uses marshmallow to load and validate the data. The actual deserialization happens in *parse_request_data*.

Any provided **kwargs will be passed to marshmallow.Schema.load().

Parameters

- **data_raw** – The request data to load.
- **expected_id** – The expected ID in the request data. See *validate_request_id*.

Returns The deserialized data

Return type dict

deserializer

The marshmallow.Schema for serialization.

By default, this is *ApiView.schema*. This can be overridden to use a different schema for deserialization.

format_validation_error(*message*, *path*)

Convert marshmallow validation error data to a serializable form.

This converts marshmallow validation error data to a standard serializable representation. By default, it converts errors into a dictionary of the form:

```
{
    "code": "invalid_data",
    "detail": "<error message>",
    "source": {
        "pointer": "/data/<field name>"
    }
}
```

Parameters

- **message** (*str*) – The marshmallow validation error message.
- **path** (*tuple*) – The path to the invalid field.

Returns The formatted validation error.

Return type dict

validate_request_id(*data*, *expected_id*)

Check that the request data has the expected ID.

This is generally used to assert that update operations include the correct item ID and that create operations do not include an ID.

This works in one of three modes:: - If `expected_id` is `None`, do no checking - If `expected_id` is `False`, check that no ID is provided - Otherwise, check that `data` has the expected ID

Parameters

- **data** – The request data.
- **expected_id** – The ID or ID tuple, or False, or None.

Raises *ApiError* – If the necessary IDs are not present and correct

get_data_id(*data*)

Get the ID as a scalar or tuple from request data.

The ID will be a scalar if *id_fields* contains a single field or a tuple if it contains multiple.

Returns The ID scalar or tuple.

property request_args

The query arguments for the current request.

This uses *args_schema* to load the current query args. This value is cached per request, and will be computed the first time it is called for any request.

Returns The query arguments.

Return type dict

deserialize_args(*data_raw*, ***kwargs*)

Load parsed query arg data using *args_schema*.

As with *deserialize*, contra the name, this handles loading with a schema rather than deserialization per se.

Parameters

- **data_raw** (*dict*) – The raw query data.
- **kwargs** (*dict*) – Additional keyword arguments for `marshmallow.Schema.load`.

Returns The deserialized data

Return type object

format_parameter_validation_error(*message*, *parameter*)

Convert a parameter validation error to a serializable form.

This closely follows *format_validation_error*, but produces error dictionaries of the form:

```
{
    "code": "invalid_parameter",
    "detail": "<error message>",
    "source": {
        "parameter": "<parameter name>"
    }
}
```

Parameters

- **message** (*str*) – The validation error message.
- **parameter** (*str*) – The query parameter name.

Returns The formatted parameter validation error

Return type dict

get_id_dict(*id*)

Convert an ID from *get_data_id* to dictionary form.

This converts an ID from *get_data_id* into a dictionary where each ID value is keyed by the corresponding ID field name.

Parameters **id** – An ID from *get_id_dict*

Type str or tuple

Returns A mapping from ID field names to ID field values

Return type dict

class flask_resty.**GenericModelView**

Base class for API views implementing CRUD methods.

GenericModelView provides basic implementations of the standard CRUD HTTP methods using the methods implemented in *ModelView*.

In simple APIs, most view classes will extend *GenericModelView*, and will declare methods that immediately call the methods here.

```
class WidgetViewBase(GenericModelView):
    model = models.Widget
    schema = models.WidgetSchema()


class WidgetListView(WidgetViewBase):
    def get(self):
        return self.list()

    def post(self):
        return self.create()


class WidgetView(WidgetViewBase):
    def get(self, id):
        return self.retrieve(id)

    def patch(self, id):
        return self.update(id, partial=True)

    def delete(self, id):
        return self.destroy(id)
```

To extend or otherwise customize the behavior of the methods here, override the methods in `MethodView`.

list()

Return a list of items.

This is the standard GET handler on a list view.

Returns An HTTP 200 response.

Return type `flask.Response`

retrieve(*id*, *, *create_transient_stub=False*)

Retrieve an item by ID.

This is the standard `GET` handler on a detail view.

Parameters

- **id** – The item ID.
- **create_transient_stub** (*bool*) – If set, create and retrieve a transient stub for the item if it is not found. This will not save the stub to the database.

Returns An HTTP 200 response.

Return type flask.Response

create(*, *allow_client_id=False*)

Create a new item using the request data.

This is the standard `POST` handler on a list view.

Parameters **allow_client_id** (*bool*) – If set, allow the client to specify ID fields for the item.

Returns An HTTP 201 response.

Return type flask.Response

update(*id*, *, *with_for_update=False*, *partial=False*)

Update the item for the specified ID with the request data.

This is the standard `PUT` handler on a detail view if `partial` is not set, or the standard `PATCH` handler if `partial` is set.

Parameters

- **id** – The item ID.
- **with_for_update** (*bool*) – If set, lock the item row while updating using `FOR UPDATE`.
- **partial** (*bool*) – If set, perform a partial update for the item, ignoring fields marked `required` on `deserializer`.

Returns An HTTP 200 response.

Return type flask.Response

upsert(*id*, *, *with_for_update=False*)

Upsert the item for the specified ID with the request data.

This will update the item for the given ID, if that item exists. Otherwise, this will create a new item with the request data.

Parameters

- **id** – The item ID.
- **with_for_update** (*bool*) – If set, lock the item row while updating using `FOR UPDATE`.

Returns An HTTP 200 or 201 response.

Return type flask.Response

destroy(*id*)

Delete the item for the specified ID.

Parameters **id** – The item ID.

Returns An HTTP 204 response.

Return type flask.Response

class flask_resty.**ModelView**

Base class for API views tied to SQLAlchemy models.

ModelView implements additional methods on top of those provided by *ApiView* to interact with SQLAlchemy models.

The functionality in this class largely ties together the authorization and the model. It provides for access to the model query as appropriately filtered for authorized rows, and provides methods to create or update model instances from request data with authorization checks.

It also provides functionality to apply filtering, sorting, and pagination when getting lists of items, and for resolving related items when deserializing request data.

model = None

A declarative SQLAlchemy model.

filtering = None

An instance of *filtering.Filtering*.

sorting = None

An instance of *sorting.SortingBase*.

pagination = None

An instance of `pagination.PaginationBase`.

related = None

An instance of *related.Related*.

session

Convenience property for the current SQLAlchemy session.

query_raw

The raw SQLAlchemy query for the view.

This is the base query, without authorization filters or query options. By default, this is the query property on the model class. This can be overridden to remove filters attached to that query.

query

The SQLAlchemy query for the view.

Override this to customize the query to fetch items in this view.

By default, this applies the filter from the view's `authorization` and the query options from *base_query_options* and *query_options*.

base_query_options = ()

Base query options to apply before *query_options*.

Set this on a base class to define base query options for its subclasses, while still allowing those subclasses to define their own additional query options via *query_options*.

For example, set this to `(raiseload('*', sql_only=True),)` to prevent all implicit SQL-emitting relationship loading, and force all relationship loading to be explicitly defined via *query_options*.

query_options

Options to apply to the query for the view.

Set this to configure relationship and column loading.

By default, this calls the `get_query_options` method on the serializer with a `Load` object bound to the model, if that serializer method exists.

Returns A sequence of query options.

Return type tuple

`get_list`()

Retrieve a list of items.

This takes the output of *get_list_query* and applies pagination.

Returns The list of items.

Return type list

`get_list_query`()

Build the query to retrieve a filtered and sorted list of items.

Returns The list query.

Return type sqlalchemy.orm.query.Query

`filter_list_query`(*query*)

Apply filtering as specified to the provided *query*.

Param A SQL query

Type sqlalchemy.orm.query.Query

Returns The filtered query

Return type sqlalchemy.orm.query.Query

`sort_list_query`(*query*)

Apply sorting as specified to the provided *query*.

Param A SQL query

Type sqlalchemy.orm.query.Query

Returns The sorted query

Return type sqlalchemy.orm.query.Query

`paginate_list_query`(*query*)

Retrieve the requested page from *query*.

If *pagination* is configured, this will retrieve the page as specified by the request and the pagination configuration. Otherwise, this will retrieve all items from the query.

Param A SQL query

Type sqlalchemy.orm.query.Query

Returns The paginated query

Return type sqlalchemy.orm.query.Query

`get_item_or_404`(*id*, ***kwargs*)

Get an item by ID; raise a 404 if it not found.

This will get an item by ID per *get_item* below. If no item is found, it will rethrow the `NoResultFound` exception as an HTTP 404.

Parameters **id** – The item ID.

Returns The item corresponding to the ID.

Return type object

get_item(*id*, *, *with_for_update=False*, *create_transient_stub=False*)

Get an item by ID.

The ID should be the scalar ID value if `id_fields` specifies a single field. Otherwise, it should be a tuple of each ID field value, corresponding to the elements of `id_fields`.

Parameters

- **id** – The item ID.
- **with_for_update** (*bool*) – If set, lock the item row for updating using `FOR UPDATE`.
- **create_transient_stub** (*bool*) – If set, create and return a transient stub for the item using *create_stub_item* if it is not found. This will not save the stub to the database.

Returns The item corresponding to the ID.

Return type object

deserialize(*data_raw*, ***kwargs*)

Load data using the `deserializer`.

In addition to the functionality of *ApiView.deserialize()*, this will resolve related items using the configured *related*.

resolve_related(*data*)

Resolve all related fields per *related*.

Parameters **data** (*object*) – A deserialized object

Returns The object with related fields resolved

Return type object

resolve_related_item(*data*, ***kwargs*)

Retrieve the related item corresponding to the provided data stub.

This is used by *Related* when this view is set for a field.

Parameters **data** (*dict*) – Stub item data with ID fields.

Returns The item corresponding to the ID in the data.

Return type object

resolve_related_id(*id*, ***kwargs*)

Retrieve the related item corresponding to the provided ID.

This is used by *Related* when a field is specified as a *RelatedId*.

Parameters **id** – The item ID.

Returns The item corresponding to the ID.

Return type object

create_stub_item(*id*)

Create a stub item that corresponds to the provided ID.

This is used by *get_item* when `create_transient_stub` is set.

Override this to configure the creation of stub items.

Parameters **id** – The item ID.

Returns A transient stub item corresponding to the ID.

Return type object

create_item(*data*)

Create an item using the provided data.

This will invoke `authorize_create_item` on the created item.

Override this to configure the creation of items, e.g. by adding additional entries to `data`.

Parameters **data** (*dict*) – The deserialized data.

Returns The newly created item.

Return type object

create_item_raw(*data*)

As with *create_item*, but without the authorization check.

This is used by *create_item*, which then applies the authorization check.

Override this instead of *create_item* when applying other modifications to the item that should take place before running the authorization check.

Parameters **data** (*dict*) – The deserialized data.

Returns The newly created item.

Return type object

add_item(*item*)

Add an item to the current session.

This will invoke `authorize_save_item` on the item to add.

Parameters **item** (*object*) – The item to add.

add_item_raw(*item*)

As with *add_item*, but without the authorization check.

This is used by *add_item*, which then applies the authorization check.

Parameters **item** (*object*) – The item to add.

create_and_add_item(*data*)

Create an item using the provided data, then add it to the session.

This uses *create_item* and *add_item*. Correspondingly, it will invoke both `authorize_create_item` and `authorize_save_item` on the item.

Parameters **data** (*dict*) – The deserialized data.

Returns The created and added item.

Return type object

update_item(*item*, *data*)

Update an existing item with the provided data.

This will invoke `authorize_update_item` using the provided item and data before updating the item, then `authorize_save_item` on the updated item afterward.

Override this to configure the updating of items, e.g. by adding additional entries to `data`.

Parameters

- **item** (*object*) – The item to update.

- **data** (*dict*) – The deserialized data.

Returns The newly updated item.

Return type object

update_item_raw(*item*, *data*)

As with *update_item*, but without the authorization checks.

Override this instead of *update_item* when applying other modifications to the item that should take place before and after the authorization checks in the above.

Parameters

- **item** (*object*) – The item to update.
- **data** (*dict*) – The deserialized data.

Returns The newly updated item.

Return type object

upsert_item(*id*, *data*, *with_for_update=False*)

Update an existing item with the matching id or if the item does not exist yet, create and insert it.

This combines `self.create_and_add_item` and `self.update_item` depending on if the item exists or not.

Parameters

- **id** – The item's identifier.
- **data** (*dict*) – The data to insert or update.
- **with_for_update** (*bool*) – If set, lock the item row for updating using `FOR UPDATE`.

Returns a tuple consisting of the newly created or the updated item and True if the item was created, False otherwise.

Return type object, bool

delete_item(*item*)

Delete an existing item.

This will run `authorize_delete_item` on the item before deleting it.

Parameters **item** (*object*) – The item to delete.

Returns The deleted item.

Return type object

delete_item_raw(*item*)

As with *delete_item*, but without the authorization check.

Override this to customize the delete behavior, e.g. by replacing the delete action with an update to mark the item deleted.

Parameters **item** (*object*) – The item to delete.

flush(*, *objects=None*)

Flush pending changes to the database.

This will check database level invariants, and will throw exceptions as with *commit* if any invariant violations are found.

It's a common pattern to call *flush*, then make external API calls, then call *commit*. The *flush* call will do a preliminary check on database-level invariants, making it less likely that the *commit* operation will fail, and reducing the risk of the external systems being left in an inconsistent state.

Parameters **objects** – If specified, the specific objects to flush. Otherwise, all pending changes will be flushed.

Returns

commit()

Commit changes to the database.

Any integrity errors that arise will be passed to *resolve_integrity_error*, which is expected to convert integrity errors corresponding to cross-row database-level invariant violations to HTTP 409 responses.

Raises *ApiError* if the commit fails with integrity errors arising from foreign key or unique constraint violations.

resolve_integrity_error(*error*)

Convert integrity errors to HTTP error responses as appropriate.

Certain kinds of database integrity errors cannot easily be caught by schema validation. These errors include violations of unique constraints and of foreign key constraints. While it's sometimes possible to check for those in application code, it's often best to let the database handle those. This will then convert those integrity errors to HTTP 409 responses.

On PostgreSQL, this uses additional integrity error details to not convert NOT NULL violations and CHECK constraint violations to HTTP 409 responses, as such checks should be done in the schema.

Returns The resolved error.

Return type Exception

set_item_response_meta(*item*)

Set the appropriate response metadata for the response item.

By default, this adds the item metadata from the pagination component.

Parameters **item** (*object*) – The item in the response.

set_item_response_meta_pagination(*item*)

Set pagination metadata for the response item.

This uses the configured pagination component to set pagination metadata for the response item.

Parameters **item** (*object*) – The item in the response.

flask_resty.**get_item_or_404**(*func=None*, ***decorator_kwargs*)

Make a view method receive an item rather than the item's ID.

This decorator takes the ID fields per `id_fields` on the view class, then uses them to fetch the corresponding item using the *get_item_or_404* on the view class.

This function can be used directly as a decorator, or it can be called with keyword arguments and then used to decorate a method to pass those arguments to *get_item_or_404*:

```
class MyView(ModelView):
    @get_item_or_404
    def get(self, item):
        pass

    @get_item_or_404(with_for_update=True)
```

(continues on next page)

(continued from previous page)

```
    def put(self, item):
        pass
```

Parameters **func** (*function or None*) – The function to decorate

Returns The decorated function or a decorator factory

Return type function

4.1.3 Authentication

class flask_resty.**AuthenticationBase**

Base class for API authentication components.

Authentication components are responsible for extracting the request credentials, if any. They should raise a 401 if the credentials are invalid, but should provide `None` for unauthenticated users.

Flask-RESTy provides an implementation using JSON Web Tokens but you can use any authentication component by extending *AuthenticationBase* and implementing *get_request_credentials()*.

authenticate_request()

Store the request credentials in the `flask.ctx.AppContext`.

> **Warning:** No validation is performed by Flask-RESTy. It is up to the implementor to validate the request in *get_request_credentials()*.

get_request_credentials()

Get the credentials for the current request.

Typically this is done by inspecting `flask.request`.

> **Warning:** Implementing classes **must** raise an exception on authentication failure. A 401 Unauthorized *ApiError* is recommended.

Returns The credentials for the current request.

class flask_resty.**NoOpAuthentication**

An authentication component that provides no credentials.

class flask_resty.**HeaderAuthenticationBase**

Base class for header authentication components.

These authentication components get their credentials from the `Authorization` request header. The Authorization header has the form:

```
Authorization: <scheme> <token>
```

This class also supports fallback to a query parameter, for cases where API clients cannot set headers.

header_scheme = 'Bearer'

Corresponds to the <scheme> in the Authorization request header.

credentials_arg = None

A fallback query parameter. The value of this query parameter will be used as credentials if the Authorization request header is missing.

get_request_credentials()

Get the credentials for the current request.

Typically this is done by inspecting `flask.request`.

> **Warning:** Implementing classes **must** raise an exception on authentication failure. A 401 Unauthorized *ApiError* is recommended.

Returns The credentials for the current request.

get_credentials_from_token(*token*)

Get the credentials from the token from the request.

Parameters **token** (*str*) – The token from the request headers or query.

Returns The credentials from the token.

class flask_resty.**HeaderAuthentication**

Header authentication component where the token is the credential.

This authentication component is useful for simple applications where the token itself is the credential, such as when it is a fixed secret shared between the client and the server that uniquely identifies the client.

4.1.4 Authorization

class flask_resty.**AuthorizationBase**

Base class for the API authorization components.

Authorization components control access to objects based on the credentials from authentication component.

Authorization components can control access in the following ways:

- Disallowing a request as a whole
- Filtering the list of visible rows in the database
- Disallowing specific modify actions

For many CRUD endpoints, *AuthorizeModifyMixin* allows consistent control of modify operations.

get_request_credentials()

Retrieve the credentials stored in the `flask.ctx.AppContext`.

authorize_request()

Authorization hook called before processing a request.

Typically this hook will inspecting `flask.request`.

filter_query(*query*, *view*)

Filter a query to hide unauthorized rows.

Parameters

- **query** (`sqlalchemy.orm.query.Query`) – The SQL construction object.

- **view** (*ModelView*) – The View instance

Returns The filtered SQL construction object.

Return type sqlalchemy.orm.query.Query

authorize_save_item(*item*)

Authorization hook called before saving a created or updated item.

This will generally be called after *authorize_create_item* or *authorize_update_item* below.

Parameters **item** (*obj*) – The model instance

authorize_create_item(*item*)

Authorization hook called before creating a new item.

Parameters **item** (*obj*) – The model instance

authorize_update_item(*item*, *data*)

Authorization hook called before updating an existing item.

Parameters

- **item** (*obj*) – The model instance
- **data** (*dict*) – A mapping from field names to updated values

authorize_delete_item(*item*)

Authorization hook called before deleting an existing item.

Parameters **item** (*obj*) – The model instance

class flask_resty.**AuthorizeModifyMixin**

An authorization component that consistently authorizes all modifies.

Child classes should implement *authorize_modify_item()*.

authorize_save_item(*item*)

Authorization hook called before saving a created or updated item.

This will generally be called after *authorize_create_item* or *authorize_update_item* below.

Parameters **item** (*obj*) – The model instance

authorize_create_item(*item*)

Authorization hook called before creating a new item.

Parameters **item** (*obj*) – The model instance

authorize_update_item(*item*, *data*)

Authorization hook called before updating an existing item.

Parameters

- **item** (*obj*) – The model instance
- **data** (*dict*) – A mapping from field names to updated values

authorize_delete_item(*item*)

Authorization hook called before deleting an existing item.

Parameters **item** (*obj*) – The model instance

authorize_modify_item(*item*, *action*)

Authorization hook for all modification actions on an item.

Parameters

- **item** (*obj*) – The model instance
- **action** (*str*) – One of `'save' | 'create' | 'update' | 'delete'`

class flask_resty.**HasAnyCredentialsAuthorization**

An authorization component that allows any action when authenticated.

This doesn't check the credentials; it just checks that some valid credentials were provided.

class flask_resty.**HasCredentialsAuthorizationBase**

A base authorization component that requires some authentication.

This authorization component doesn't check the credentials, but will block all requests that do not provide some credentials.

authorize_request()

Authorization hook called before processing a request.

Typically this hook will inspecting `flask.request`.

class flask_resty.**NoOpAuthorization**

An authorization component that allows any action.

authorize_request()

Authorization hook called before processing a request.

Typically this hook will inspecting `flask.request`.

filter_query(*query*, *view*)

Filter a query to hide unauthorized rows.

Parameters

- **query** (`sqlalchemy.orm.query.Query`) – The SQL construction object.
- **view** (*ModelView*) – The View instance

Returns The filtered SQL construction object.

Return type `sqlalchemy.orm.query.Query`

authorize_save_item(*item*)

Authorization hook called before saving a created or updated item.

This will generally be called after *authorize_create_item* or *authorize_update_item* below.

Parameters **item** (*obj*) – The model instance

authorize_create_item(*item*)

Authorization hook called before creating a new item.

Parameters **item** (*obj*) – The model instance

authorize_update_item(*item*, *data*)

Authorization hook called before updating an existing item.

Parameters

- **item** (*obj*) – The model instance

- **data** (*dict*) – A mapping from field names to updated values

authorize_delete_item(*item*)

Authorization hook called before deleting an existing item.

Parameters **item** (*obj*) – The model instance

4.1.5 Relationships

class flask_resty.**Related**(*item_class=None*, ***kwargs*)

A component for resolving deserialized data fields to model instances.

The *Related* component is responsible for resolving related model instances by ID and for constructing nested model instances. It supports multiple related types of functionality. For a view with:

```
related = Related(
    foo=RelatedId(FooView, "foo_id"),
    bar=BarView,
    baz=Related(models.Baz, qux=RelatedId(QuxView, "qux_id"),
)
```

Given deserialized input data like:

```
{
    "foo_id": "3",
    "bar": {"id": "4"},
    "baz": {name: "Bob", "qux_id": "5"},
    "other_field": "value",
}
```

This component will resolve these data into something like:

```
{
    "foo": <Foo(id=3)>,
    "bar": <Bar(id=4)>,
    "baz": <Baz(name="Bob", qux=<Qux(id=5)>>,
    "other_field": "value",
}
```

In this case, the Foo, Bar, and Qux instances are fetched from the database, while the Baz instance is freshly constructed. If any of the Foo, Bar, or Qux instances do not exist, then the component will fail the request with a 422.

Formally, in this specification:

- A *RelatedId* item will retrieve the existing object in the database with the ID from the specified scalar ID field using the specified view.
- A view class will retrieve the existing object in the database using the object stub containing the ID fields from the data field of the same name, using the specified view. This is generally used with the *RelatedItem* field class, and unlike *RelatedId*, supports composite IDs.
- Another *Related* item will apply the same resolution to a nested dictionary. Additionally, if the *Related* item is given a callable as its positional argument, it will construct a new instance given that callable, which can often be a model class.

Related depends on the deserializer schema to function accordingly, and delegates validation beyond the database fetch to the schema. *Related* also automatically supports cases where the fields are list fields or are configured with `many=True`. In those cases, *Related* will iterate through the sequence and resolve each item in turn, using the rules as above.

Parameters

- **item_class** – The SQLAlchemy mapper corresponding to the related item.
- **kwargs** (*dict*) – A mapping from related fields to a callable resolver.

resolve_related(*data*)

Resolve the related values in the request data.

This method will replace values in `data` with resolved model instances as described above. This operates in place and will mutate `data`.

Parameters **object** (*data*) – The deserialized request data.

Returns The deserialized data with related fields resolved.

Return type object

resolve_field(*value*, *resolver*)

Resolve a single field value.

Parameters

- **value** – The value corresponding to the field we are resolving.
- **resolver** (*Related* | *RelatedId* | func) – A callable capable of resolving the given `value`.

class flask_resty.**RelatedId**(*view_class*, *field_name*)

Resolve a related item by a scalar ID.

Parameters

- **view_class** – The *ModelView* corresponding to the related model.
- **field_name** (*str*) – The field name on request data.

class flask_resty.**RelatedItem**(*nested: SchemaABC | type | str | dict[str, Field | type] | typing.Callable[[], SchemaABC | dict[str, Field | type]], *, dump_default: typing.Any = <marshmallow.missing>, default: typing.Any = <marshmallow.missing>, only: types.StrSequenceOrSet | None = None, exclude: types.StrSequenceOrSet = (), many: bool = False, unknown: str | None = None, **kwargs*)

A nested object field that only requires the ID on load.

This class is a wrapper around `marshmallow.fields.Nested` that provides simplified semantics in the context of a normalized REST API.

When dumping, this field will dump the nested object as normal. When loading, this field will do a partial load to retrieve just the ID. This is because, when interacting with a resource that has a relationship to existing instances of another resource, the ID is sufficient to uniquely identify instances of the other resource.

4.1.6 Filtering

class flask_resty.**ArgFilterBase**

An abstract specification of a filter from a query argument.

Implementing classes must provide *maybe_set_arg_name()* and *filter_query()*.

maybe_set_arg_name(*arg_name*)

Set the name of the argument to which this filter is bound.

Parameters **arg_name** (*str*) – The name of the field to filter against.

Raises NotImplementedError if no implementation is provided.

filter_query(*query*, *view*, *arg_value*)

Filter the query.

Parameters

- **query** (sqlalchemy.orm.query.Query) – The query to filter.
- **view** (*ModelView*) – The view with the model we wish to filter for.
- **arg_value** (*str*) – The filter specification

Returns The filtered query

Return type sqlalchemy.orm.query.Query

Raises NotImplementedError if no implementation is provided.

class flask_resty.**ColumnFilter**(*column_name=None*, *operator=None*, ***, *required=False*, *missing=<marshmallow.missing>*, *validate=True*, ***kwargs*)

A filter that operates on the value of a database column.

This filter relies on the schema to deserialize the query argument values. *ColumnFilter* cannot normally be used for columns that do not appear on the schema, but such columns can be added to the schema with fields that have both `load_only` and `dump_only` set.

Parameters

- **column_name** (*str*) – The name of the column to filter against.
- **operator** (*func*) – A callable that returns the filter expression given the column and the filter value.
- **required** (*bool*) – If set, fail if this filter is not specified.
- **validate** (*bool*) – If unset, bypass validation on the field. This is useful if the field specifies validation rule for inputs that are not relevant for filters.

maybe_set_arg_name(*arg_name*)

Set `arg_name` as the column name if no explicit value is available.

Parameters **arg_name** (*str*) – The name of the column to filter against.

get_field(*view*)

Construct the marshmallow field for deserializing filter values.

This takes the field from the deserializer, then creates a copy with the desired semantics around missing values.

Parameters **view** (*ModelView*) – The view with the model we wish to filter for.

get_filter_clause(*view*, *value*)

Build the filter clause for the deserialized value.

Parameters

- **view** (*ModelView*) – The view with the model we wish to filter for.
- **value** (*str*) – The right-hand side of the WHERE clause.

Raises NotImplementedError if no implementation is provided.

deserialize(*field*, *value_raw*)

Deserialize `value_raw`, optionally skipping validation.

Parameters

- **field** (marshmallow.fields.Field) – The marshmallow field.
- **value_raw** – The value to deserialize.

Returns The deserialized value.

filter_query(*query*, *view*, *arg_value*)

Filter the query.

Parameters

- **query** (sqlalchemy.orm.query.Query) – The query to filter.
- **view** (*ModelView*) – The view with the model we wish to filter for.
- **arg_value** (*str*) – The filter specification

Returns The filtered query

Return type sqlalchemy.orm.query.Query

Raises NotImplementedError if no implementation is provided.

class flask_resty.**FieldFilterBase**(*, *separator=','*, *allow_empty=False*, *skip_invalid=False*)

A filter that uses a marshmallow field to deserialize its value.

Implementing classes must provide `get_filter_field()` and *get_filter_clause()*.

Parameters

- **separator** (*str*) – Character that separates individual elements in the query value.
- **allow_empty** (*bool*) – If set, allow filtering for empty values; otherwise, filter out all items on an empty value.
- **skip_invalid** (*bool*) – If set, ignore invalid filter values instead of throwing an API error.

maybe_set_arg_name(*arg_name*)

Set the name of the argument to which this filter is bound.

Parameters **arg_name** (*str*) – The name of the field to filter against.

Raises NotImplementedError if no implementation is provided.

filter_query(*query*, *view*, *arg_value*)

Filter the query.

Parameters

- **query** (sqlalchemy.orm.query.Query) – The query to filter.
- **view** (*ModelView*) – The view with the model we wish to filter for.

- **arg_value** (*str*) – The filter specification

Returns The filtered query

Return type sqlalchemy.orm.query.Query

Raises NotImplementedError if no implementation is provided.

deserialize(*field*, *value_raw*)

Overridable hook for deserializing a value.

Parameters

- **field** (marshmallow.fields.Field) – The marshmallow field.
- **value_raw** – The value to deserialize.

Returns The deserialized value.

get_field(*view*)

Get the marshmallow field for deserializing filter values.

Parameters **view** (*ModelView*) – The view with the model we wish to filter for.

Raises NotImplementedError if no implementation is provided.

get_filter_clause(*view*, *value*)

Build the filter clause for the deserialized value.

Parameters

- **view** (*ModelView*) – The view with the model we wish to filter for.
- **value** (*str*) – The right-hand side of the WHERE clause.

Raises NotImplementedError if no implementation is provided.

class flask_resty.**Filtering**(***kwargs*)

Container for the arg filters on a *ModelView*.

Parameters **kwargs** (*dict*) – A mapping from filter field names to filters.

filter_query(*query*, *view*)

Filter a query using the configured filters and the request args.

Parameters

- **query** (sqlalchemy.orm.query.Query) – The query to filter.
- **view** (*ModelView*) – The view with the model we wish to filter for.

Returns The filtered query

Return type sqlalchemy.orm.query.Query

flask_resty.**model_filter**(*field*, ***kwargs*)

A convenience decorator for building a *ModelFilter*.

This decorator allows building a *ModelFilter* around a named function:

```
@model_filter(fields.String(required=True))
def filter_color(model, value):
    return model.color == value
```

Parameters

- **field** (marshmallow.fields.Field) – A marshmallow field for deserializing filter values.
- **kwargs** (*dict*) – Passed to *ModelFilter*.

class flask_resty.**ModelFilter**(*field*, *filter*, ***kwargs*)

An arbitrary filter against the model.

Parameters

- **field** (marshmallow.fields.Field) – A marshmallow field for deserializing filter values.
- **filter** – A callable that returns the filter expression given the model and the filter value.
- **kwargs** (*dict*) – Passed to *FieldFilterBase*.

get_field(*view*)

Get the marshmallow field for deserializing filter values.

Parameters **view** (*ModelView*) – The view with the model we wish to filter for.

Raises NotImplementedError if no implementation is provided.

get_filter_clause(*view*, *value*)

Build the filter clause for the deserialized value.

Parameters

- **view** (*ModelView*) – The view with the model we wish to filter for.
- **value** (*str*) – The right-hand side of the WHERE clause.

Raises NotImplementedError if no implementation is provided.

deserialize(*field*, *value_raw*)

Overridable hook for deserializing a value.

Parameters

- **field** (marshmallow.fields.Field) – The marshmallow field.
- **value_raw** – The value to deserialize.

Returns The deserialized value.

filter_query(*query*, *view*, *arg_value*)

Filter the query.

Parameters

- **query** (sqlalchemy.orm.query.Query) – The query to filter.
- **view** (*ModelView*) – The view with the model we wish to filter for.
- **arg_value** (*str*) – The filter specification

Returns The filtered query

Return type sqlalchemy.orm.query.Query

Raises NotImplementedError if no implementation is provided.

maybe_set_arg_name(*arg_name*)

Set the name of the argument to which this filter is bound.

Parameters **arg_name** (*str*) – The name of the field to filter against.

Raises NotImplementedError if no implementation is provided.

4.1.7 Pagination

class flask_resty.**CursorPaginationBase**(**args*, *validate_values=True*, ***kwargs*)

The base class for pagination schemes that use cursors.

Unlike with offsets that identify items by relative position, cursors identify position by content. This allows continuous pagination without concerns about page shear on dynamic collections. This makes cursor-based pagination especially effective for lists with infinite scroll dynamics, as offset-based pagination can miss or duplicate items due to inserts or deletes.

It's also more efficient against the database, as the cursor condition can be cheaply evaluated as a filter against an index.

Parameters **validate** (*bool*) – If unset, bypass validation on cursor values. This is useful if the deserializer field imposes validation that will fail for on cursor values for items actually present.

cursor_arg = 'cursor'

The name of the query parameter to inspect for the cursor value.

after_arg = 'after'

the name of the query parameter to inspect for explicit forward pagination

before_arg = 'before'

the name of the query parameter to inspect for explicit backward pagination

get_limit()

Override this method to return the maximum number of returned items.

Return type int

adjust_sort_ordering(*view:* flask_resty.view.ModelView, *field_orderings*) → Tuple[Tuple[str, bool], ...]

Ensure the query is sorted correctly and get the field orderings.

The implementation of cursor-based pagination in Flask-RESTy requires that the query be sorted in a fully deterministic manner. The timestamp columns usually used in sorting do not quite qualify, as two different rows can have the same timestamp. This method adds the ID fields to the sorting criterion, then returns the field orderings for use in the other methods, as in `get_field_orderings` below.

Parameters **view** (*ModelView*) – The view with the model we wish to paginate.

Returns The field orderings necessary to do cursor pagination deterministically

Return type FieldOrderings

get_request_cursor(*view*, *field_orderings*)

Get the cursor value specified in the request.

Given the view and the field_orderings as above, this method will read the encoded cursor from the query, then return the cursor as a tuple of the field values in the cursor.

This parsed cursor can then be used in *get_filter*.

Parameters

- **view** (*ModelView*) – The view with the model we wish to paginate.
- **field_orderings** (*seq*) – A sequence of field_ordering tuples

Returns A cursor value

Return type str

Raises *ApiError* if an invalid cursor is provided in *cursor_arg*.

get_filter(*view*, *field_orderings: Tuple[Tuple[str, bool], ...]*, *cursor: Tuple[Any, ...]*)

Build the filter clause corresponding to a cursor.

Given the field orderings and the cursor as above, this will construct a filter clause that can be used to filter a query to return only items after the specified cursor, per the specified field orderings. Use this to apply the equivalent of the offset specified by the cursor.

Parameters

- **view** (*ModelView*) – The view with the model we wish to paginate.
- **field_orderings** (*seq*) – A sequence of field_ordering tuples derived from the view's Sorting with explicit id ordering
- **cursor** (*seq*) – A set of values corresponding to the fields in `field_orderings`

Returns A filter clause

make_cursors(*items*, *view*, *field_orderings*)

Build a cursor for each of many items.

This method creates a cursor for each item in `items`. It produces the same cursors as *make_cursor()*, but is slightly more efficient in cases where cursors for multiple items are required.

Parameters

- **items** (*seq*) – A sequence of instances of `ApiView.model`
- **view** (*ModelView*) – The view we wish to paginate.
- **field_orderings** (*seq*) – A sequence of (field, asc?).

Returns A sequence of `marshmallow.Field`.

Return type seq

make_cursor(*item*, *view*, *field_orderings*)

Build a cursor for a given item.

Given an item and the field orderings as above, this builds a cursor for the item. This cursor encodes the value for each field on the item per the specified field orderings.

This cursor should be returned in page or item metadata to allow pagination continuing after the cursor for the item.

Parameters

- **item** (*obj*) – An instance `ApiView.model`
- **view** (*ModelView*) – The view we wish to paginate.
- **field_orderings** (*seq*) – A sequence of (field, asc?).

Returns A sequence of `marshmallow.Field`.

Return type seq

get_item_meta(*item*, *view*)

Build pagination metadata for a single item.

Parameters

- **item** (*obj*) – An instance of the *ModelView.model*.
- **view** (*ModelView*) – The view with the *ModelView.model*.

get_page(*query*, *view*) → List

Restrict the specified query to a single page.

Parameters

- **query** (sqlalchemy.orm.query.Query) – The query to paginate.
- **view** (*ModelView*) – The view with the model we wish to paginate.

Returns The paginated query

Return type sqlalchemy.orm.query.Query

Raises A NotImplementedError if no implementation is provided.

class flask_resty.**LimitOffsetPagination**(*default_limit=None*, *max_limit=None*)

A pagination scheme that takes a user-specified limit and offset.

This pagination scheme takes a user-specified limit and offset. It will retrieve up to the specified number of items, beginning at the specified offset.

offset_arg = 'offset'

The name of the query parameter to inspect for the OFFSET value.

get_page(*query*, *view*)

Restrict the specified query to a single page.

Parameters

- **query** (sqlalchemy.orm.query.Query) – The query to paginate.
- **view** (*ModelView*) – The view with the model we wish to paginate.

Returns The paginated query

Return type sqlalchemy.orm.query.Query

Raises A NotImplementedError if no implementation is provided.

get_item_meta(*item*, *view*)

Build pagination metadata for a single item.

Parameters

- **item** (*obj*) – An instance of the *ModelView.model*.
- **view** (*ModelView*) – The view with the *ModelView.model*.

get_limit()

Override this method to return the maximum number of returned items.

Return type int

class flask_resty.**LimitPagination**(*default_limit=None*, *max_limit=None*)

A pagination scheme that takes a user-specified limit.

This is not especially useful and is included only for completeness.

This pagination scheme uses the *limit_arg* query parameter to limit the number of items returned by the query.

If no such limit is explicitly specified, this uses `default_limit`. If `max_limit` is specified, then the user-specified limit may not exceed `max_limit`.

Parameters

- **default_limit** (*int*) – The default maximum number of items to retrieve, if the user does not specify an explicit value.
- **max_limit** (*int*) – The maximum number of items the user is allowed to request.

limit_arg = 'limit'

The name of the query parameter to inspect for the LIMIT value.

get_limit()

Override this method to return the maximum number of returned items.

Return type int

get_item_meta(*item*, *view*)

Build pagination metadata for a single item.

Parameters

- **item** (*obj*) – An instance of the *ModelView.model*.
- **view** (*ModelView*) – The view with the *ModelView.model*.

get_page(*query*, *view*) → List

Restrict the specified query to a single page.

Parameters

- **query** (sqlalchemy.orm.query.Query) – The query to paginate.
- **view** (*ModelView*) – The view with the model we wish to paginate.

Returns The paginated query

Return type sqlalchemy.orm.query.Query

Raises A NotImplementedError if no implementation is provided.

class flask_resty.**MaxLimitPagination**(*max_limit*)

Return up to a fixed maximum number of items.

This is not especially useful and is included only for completeness.

Parameters **max_limit** (*int*) – The maximum number of items to retrieve.

get_limit()

Override this method to return the maximum number of returned items.

Return type int

get_item_meta(*item*, *view*)

Build pagination metadata for a single item.

Parameters

- **item** (*obj*) – An instance of the *ModelView.model*.
- **view** (*ModelView*) – The view with the *ModelView.model*.

get_page(*query*, *view*) → List

Restrict the specified query to a single page.

Parameters

- **query** (sqlalchemy.orm.query.Query) – The query to paginate.
- **view** (*ModelView*) – The view with the model we wish to paginate.

Returns The paginated query

Return type sqlalchemy.orm.query.Query

Raises A NotImplementedError if no implementation is provided.

class flask_resty.**PagePagination**(*page_size*)

A pagination scheme that fetches a particular fixed-size page.

This works similar to *LimitOffsetPagination*. The limit used will always be the fixed page size. The offset will be page * page_size.

Parameters **page_size** (*int*) – The fixed number of items per page.

page_arg = 'page'

The name of the query parameter to inspect for the page value.

get_limit()

Override this method to return the maximum number of returned items.

Return type int

get_item_meta(*item*, *view*)

Build pagination metadata for a single item.

Parameters

- **item** (*obj*) – An instance of the *ModelView.model*.
- **view** (*ModelView*) – The view with the *ModelView.model*.

get_page(*query*, *view*)

Restrict the specified query to a single page.

Parameters

- **query** (sqlalchemy.orm.query.Query) – The query to paginate.
- **view** (*ModelView*) – The view with the model we wish to paginate.

Returns The paginated query

Return type sqlalchemy.orm.query.Query

Raises A NotImplementedError if no implementation is provided.

class flask_resty.**RelayCursorPagination**(**args*, *page_info_arg=None*, *default_include_page_info=False*, ***kwargs*)

A pagination scheme that works with the Relay specification.

This pagination scheme assigns a cursor to each retrieved item. The page metadata will contain an array of cursors, one per item. The item metadata will include the cursor for the fetched item.

For Relay Cursor Connections Specification, see https://facebook.github.io/relay/graphql/connections.htm.

get_page(*query*, *view*)

Restrict the specified query to a single page.

Parameters

- **query** (sqlalchemy.orm.query.Query) – The query to paginate.
- **view** (*ModelView*) – The view with the model we wish to paginate.

Returns The paginated query

Return type sqlalchemy.orm.query.Query

Raises A NotImplementedError if no implementation is provided.

get_item_meta(*item*, *view*)

Build pagination metadata for a single item.

Parameters

- **item** (*obj*) – An instance of the *ModelView.model*.
- **view** (*ModelView*) – The view with the *ModelView.model*.

adjust_sort_ordering(*view: flask_resty.view.ModelView*, *field_orderings*) → Tuple[Tuple[str, bool], ...]

Ensure the query is sorted correctly and get the field orderings.

The implementation of cursor-based pagination in Flask-RESTy requires that the query be sorted in a fully deterministic manner. The timestamp columns usually used in sorting do not quite qualify, as two different rows can have the same timestamp. This method adds the ID fields to the sorting criterion, then returns the field orderings for use in the other methods, as in `get_field_orderings` below.

Parameters **view** (*ModelView*) – The view with the model we wish to paginate.

Returns The field orderings necessary to do cursor pagination deterministically

Return type FieldOrderings

get_filter(*view*, *field_orderings: Tuple[Tuple[str, bool], ...]*, *cursor: Tuple[Any, ...]*)

Build the filter clause corresponding to a cursor.

Given the field orderings and the cursor as above, this will construct a filter clause that can be used to filter a query to return only items after the specified cursor, per the specified field orderings. Use this to apply the equivalent of the offset specified by the cursor.

Parameters

- **view** (*ModelView*) – The view with the model we wish to paginate.
- **field_orderings** (*seq*) – A sequence of field_ordering tuples derived from the view's Sorting with explicit id ordering
- **cursor** (*seq*) – A set of values corresponding to the fields in `field_orderings`

Returns A filter clause

get_limit()

Override this method to return the maximum number of returned items.

Return type int

get_request_cursor(*view*, *field_orderings*)

Get the cursor value specified in the request.

Given the view and the field_orderings as above, this method will read the encoded cursor from the query, then return the cursor as a tuple of the field values in the cursor.

This parsed cursor can then be used in *get_filter*.

Parameters

- **view** (*ModelView*) – The view with the model we wish to paginate.
- **field_orderings** (*seq*) – A sequence of field_ordering tuples

Returns A cursor value

Return type str

Raises *ApiError* if an invalid cursor is provided in `cursor_arg`.

make_cursor(*item*, *view*, *field_orderings*)

Build a cursor for a given item.

Given an item and the field orderings as above, this builds a cursor for the item. This cursor encodes the value for each field on the item per the specified field orderings.

This cursor should be returned in page or item metadata to allow pagination continuing after the cursor for the item.

Parameters

- **item** (*obj*) – An instance `ApiView.model`
- **view** (*ModelView*) – The view we wish to paginate.
- **field_orderings** (*seq*) – A sequence of (field, asc?).

Returns A sequence of `marshmallow.Field`.

Return type seq

make_cursors(*items*, *view*, *field_orderings*)

Build a cursor for each of many items.

This method creates a cursor for each item in `items`. It produces the same cursors as *make_cursor()*, but is slightly more efficient in cases where cursors for multiple items are required.

Parameters

- **items** (*seq*) – A sequence of instances of `ApiView.model`
- **view** (*ModelView*) – The view we wish to paginate.
- **field_orderings** (*seq*) – A sequence of (field, asc?).

Returns A sequence of `marshmallow.Field`.

Return type seq

4.1.8 Sorting

class flask_resty.**FieldSortingBase**

The base class for sorting components that sort on model fields.

These sorting components work on JSON-API style sort field strings, which consist of a list of comma-separated field names, optionally prepended by `-` to indicate descending sort order.

For example, the sort field string of `name,-date` will sort by `name` ascending and `date` descending.

Subclasses must implement *get_request_field_orderings()* to specify the actual field orderings to use.

sort_query(*query*, *view*)

Sort the provided `query`.

Parameters

- **query** (sqlalchemy.orm.query.Query) – The query to sort.
- **view** (*ModelView*) – The view with the model we wish to sort.

Returns The sorted query

Return type sqlalchemy.orm.query.Query

Raises A `NotImplementedError` if no implementation is provided.

get_request_field_orderings(*view*) → Tuple[Tuple[str, bool], ...]

Get the field orderings to use for the current request.

These should be created from a sort field string with *get_field_orderings* below.

Parameters **view** (*ModelView*) – The view with the model containing the target fields

Returns A sequence of field orderings. See `get_criterion()`.

Return type tuple

Raises A `NotImplementedError` if no implementation is provided.

get_field_orderings(*fields*)

Given a sort field string, build the field orderings.

These field orders can then be used with *get_request_field_orderings* above. See the class documentation for details on sort field strings.

Parameters **fields** (*str*) – The sort field string.

Returns A sequence of field orderings. See `get_criterion()`.

Return type tuple

class flask_resty.**FixedSorting**(*fields*)

A sorting component that applies a fixed sort order.

For example, to sort queries by `name` ascending and `date` descending, specify the following in your view:

```
sorting = FixedSorting('name,-date')
```

Parameters **fields** (*str*) – The formatted fields.

get_request_field_orderings(*view*)

Get the field orderings to use for the current request.

These should be created from a sort field string with *get_field_orderings* below.

Parameters **view** (*ModelView*) – The view with the model containing the target fields

Returns A sequence of field orderings. See `get_criterion()`.

Return type tuple

Raises A `NotImplementedError` if no implementation is provided.

get_field_orderings(*fields*)

Given a sort field string, build the field orderings.

These field orders can then be used with *get_request_field_orderings* above. See the class documentation for details on sort field strings.

Parameters **fields** (*str*) – The sort field string.

Returns A sequence of field orderings. See `get_criterion()`.

Return type tuple

sort_query(*query*, *view*)

Sort the provided `query`.

Parameters

- **query** (`sqlalchemy.orm.query.Query`) – The query to sort.
- **view** (*ModelView*) – The view with the model we wish to sort.

Returns The sorted query

Return type `sqlalchemy.orm.query.Query`

Raises A `NotImplementedError` if no implementation is provided.

class flask_resty.**Sorting**(**field_names*, *default=None*, ***kwargs*)

A sorting component that allows the user to specify sort fields.

For example, to allow users to sort by `title` and/or `content_length`, specify the following in your view:

```
sorting = Sorting(
    'title',
    content_length=sql.func.length(Post.content)
)
```

One or both of `title` or `content_length` can be formatted in the *sort_arg* request parameter to determine the sort order. For example, users can sort requests by `name` ascending and `date` descending by making a `GET` request to:

```
/api/comments/?sort=title,-content_length
```

Parameters

- **field_names** (*str*) – The fields available for sorting. Names should match a column on your View's `model`.
- **default** (*str*) – If provided, specifies a default sort order when the request does not specify an explicit sort order.

- **kwargs** (*dict*) – Provide custom sort behavior by mapping a sort argument name to a model order_by expression.

sort_arg = 'sort'

The request parameter from which the formatted sorting fields will be retrieved.

get_request_field_orderings(*view*)

Get the field orderings to use for the current request.

These should be created from a sort field string with *get_field_orderings* below.

Parameters **view** (*ModelView*) – The view with the model containing the target fields

Returns A sequence of field orderings. See `get_criterion()`.

Return type tuple

Raises A `NotImplementedError` if no implementation is provided.

get_field_orderings(*fields*)

Given a sort field string, build the field orderings.

These field orders can then be used with *get_request_field_orderings* above. See the class documentation for details on sort field strings.

Parameters **fields** (*str*) – The sort field string.

Returns A sequence of field orderings. See `get_criterion()`.

Return type tuple

sort_query(*query*, *view*)

Sort the provided `query`.

Parameters

- **query** (`sqlalchemy.orm.query.Query`) – The query to sort.
- **view** (*ModelView*) – The view with the model we wish to sort.

Returns The sorted query

Return type `sqlalchemy.orm.query.Query`

Raises A `NotImplementedError` if no implementation is provided.

class flask_resty.**SortingBase**

The base class for sorting components.

Sorting components control how list queries are sorted.

They also expose an API for cursor pagination components to get the sort fields, which are required to build cursors.

Subclasses must implement *sort_query()* to provide the sorting logic.

sort_query(*query*, *view*)

Sort the provided `query`.

Parameters

- **query** (`sqlalchemy.orm.query.Query`) – The query to sort.
- **view** (*ModelView*) – The view with the model we wish to sort.

Returns The sorted query

Return type sqlalchemy.orm.query.Query

Raises A NotImplementedError if no implementation is provided.

4.1.9 Exceptions

class flask_resty.**ApiError**(*status_code*, **errors*)

An API exception.

When raised, Flask-RESTy will send an HTTP response with the provided `status_code` and the provided `errors` under the `errors` property as JSON.

If flask.Flask.debug or flask.Flask.testing is True, the body will also contain the full traceback under the `debug` property.

Parameters

- **status_code** (*int*) – The HTTP status code for the error response.
- **errors** (*dict*) – A list of dict with error data.

update(*additional*)

Add additional metadata to the error.

Can be chained with further updates.

Parameters **additional** (*dict*) – The additional metadata

Returns The *ApiError* that *update()* was called on

Return type *ApiError*

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

4.1.10 Routing

class flask_resty.**StrictRule**(*string: str*, *defaults: Optional[Mapping[str, Any]] = None*, *subdomain: Optional[str] = None*, *methods: Optional[Iterable[str]] = None*, *build_only: bool = False*, *endpoint: Optional[str] = None*, *strict_slashes: Optional[bool] = None*, *merge_slashes: Optional[bool] = None*, *redirect_to: Optional[Union[str, Callable[[...], str]]] = None*, *alias: bool = False*, *host: Optional[str] = None*, *websocket: bool = False*)

A Werkzeug rule that does not append missing slashes to paths.

match(*path*, *method=None*)

Check if the rule matches a given path. Path is a string in the form `"subdomain|/path"` and is assembled by the map. If the map is doing host matching the subdomain part will be the host instead.

If the rule matches a dict with the converted values is returned, otherwise the return value is None.

Internal

bind(*map: werkzeug.routing.Map*, *rebind: bool = False*) → None

Bind the url to a map and create a regular expression based on the information from the rule itself and the defaults from the map.

Internal

build(*values: Mapping[str, Any]*, *append_unknown: bool = True*) → Optional[Tuple[str, str]]

Assembles the relative url for that rule and the subdomain. If building doesn't work for some reasons `None` is returned.

Internal

build_compare_key() → Tuple[int, int, int]

The build compare key for sorting.

Internal

compile() → None

Compiles the regular expression and stores it.

empty() → werkzeug.routing.Rule

Return an unbound copy of this rule.

This can be useful if want to reuse an already bound URL for another map. See `get_empty_kwargs` to override what keyword arguments are provided to the new copy.

get_converter(*variable_name: str*, *converter_name: str*, *args: Tuple*, *kwargs: Mapping[str, Any]*) → werkzeug.routing.BaseConverter

Looks up the converter for the given parameter.

New in version 0.9.

get_empty_kwargs() → Mapping[str, Any]

Provides kwargs for instantiating empty copy with empty()

Use this method to provide custom keyword arguments to the subclass of `Rule` when calling `some_rule.empty()`. Helpful when the subclass has custom keyword arguments that are needed at instantiation.

Must return a `dict` that will be provided as kwargs to the new instance of `Rule`, following the initial `self.rule` value which is always provided as the first, required positional argument.

get_rules(*map: werkzeug.routing.Map*) → Iterator[werkzeug.routing.Rule]

Subclasses of `RuleFactory` have to override this method and return an iterable of rules.

match_compare_key() → Tuple[bool, int, Iterable[Tuple[int, int]], int, Iterable[int]]

The match compare key for sorting.

Current implementation:

1. rules without any arguments come first for performance reasons only as we expect them to match faster and some common ones usually don't have any arguments (index pages etc.)
2. rules with more static parts come first so the second argument is the negative length of the number of the static weights.
3. we order by static weights, which is a combination of index and length
4. The more complex rules come first so the next argument is the negative length of the number of argument weights.
5. lastly we order by the actual argument weights.

Internal

provides_defaults_for(*rule: werkzeug.routing.Rule*) → bool

Check if this rule has defaults for a given rule.

Internal

refresh() → None

Rebinds and refreshes the URL. Call this if you modified the rule in place.

Internal

suitable_for(*values: Mapping[str, Any]*, *method: Optional[str] = None*) → bool

Check if the dict of values has enough data for url generation.

Internal

4.1.11 Fields

class flask_resty.**RelatedItem**(*nested: SchemaABC | type | str | dict[str, Field | type] | typing.Callable[[], SchemaABC | dict[str, Field | type]], *, dump_default: typing.Any = <marshmallow.missing>, default: typing.Any = <marshmallow.missing>, only: types.StrSequenceOrSet | None = None, exclude: types.StrSequenceOrSet = (), many: bool = False, unknown: str | None = None, **kwargs*)

A nested object field that only requires the ID on load.

This class is a wrapper around `marshmallow.fields.Nested` that provides simplified semantics in the context of a normalized REST API.

When dumping, this field will dump the nested object as normal. When loading, this field will do a partial load to retrieve just the ID. This is because, when interacting with a resource that has a relationship to existing instances of another resource, the ID is sufficient to uniquely identify instances of the other resource.

class flask_resty.**DelimitedList**(*cls_or_instance*, *delimiter=None*, *as_string=False*, ***kwargs*)

List represented as a comma-delimited string, for use with args_schema.

Same as `marshmallow.fields.List`, except can load from either a list or a delimited string (e.g. "foo,bar,baz"). Directly taken from webargs: https://github.com/marshmallow-code/webargs/blob/de061e037285fd08a42d73be95bc779f2a4e3c47/src/webargs/fields.py#L47

Parameters

- **cls_or_instance** (*Field*) – A field class or instance.
- **delimiter** (*str*) – Delimiter between values.
- **as_string** (*bool*) – Dump values to string.

4.1.12 Testing

class flask_resty.testing.**ApiClient**(**args: Any*, ***kwargs: Any*)

A `flask.testing.FlaskClient` with a few conveniences:

- Prefixes paths
- Sets `Content-Type` to "application/json"
- Envelopes `data` within a "data" key in the request payload

flask_resty.testing.**assert_shape**(*actual*, *expected*, *key=None*)

Assert that `actual` and `expected` have the same data shape.

flask_resty.testing.**assert_response**(*response*, *expected_status_code*, *expected_data=<UNDEFINED>*, ***, *get_data=<function get_data>*, *get_errors=<function get_errors>*)

Assert on the status and contents of a response.

If specified, expected_data is checked against either the data or the errors in the response body, depending on the response status. This check ignores extra dictionary items in the response contents.

PYTHON MODULE INDEX

f

INDEX

D

E

F

G

H

I

L

M

N