flask-resources Documentation

Release 1.1.0

CERN

Apr 17, 2023

CONTENTS

A small library for implementing configurable REST APIs.

Further documentation is available on https://flask-resources.readthedocs.io/

CHAPTER

ONE

USER'S GUIDE

This part of the documentation will show you how to get started in using Flask-Resources.

1.1 Installation

Flask-Resources is on PyPI so all you need is:

```
$ pip install flask-resources
```

1.2 Usage

Library for implementing configurable REST APIs.

A resource is a factory for creating Flask Blueprint that's parameterized via a config. The main difference from a regular blueprint is:

- Syntactical overlay - it creates a slightly different way of writing views and wiring them up with the Flask routing system. Flask-Resources is meant for REST APIs and thus puts emphasis on the HTTP method, and as apposed to a Flask MethodView, it allows keeping all view methods together for all endpoints.
- Dependency injection - a resource enables easy dependency injection via a configuration object. The idea behind this is for instance you write a reusable application that you want to allow developers to customize. For instance you could allow a developer to accept and deserialize their custom XML instead of only JSON at a given endpoint while keeping the application view the same, or allow them to customize the URL routes via the Flask application config.

In addition, Flask-Resources provides basic utilities for developing REST APIs such as:

- Content negotiation to support multiple response serializations (e.g. serving JSON, JSON-LD, XML from the same endpoint).
- Request parsing (query string, headers, body) using Marshmallow and data deserialization.
- Resource request context to enforce paradigm of passing only validated request data to the view function.

If you don't need any of the above, you can simply use just a normal Flask Blueprint instead.

Below is small minimal example:

```
from flask import Flask
from flask_resources import Resource, ResourceConfig, route
```

(continues on next page)

(continued from previous page)

```
class Config(ResourceConfig):
    blueprint_name = "hello"

class HelloWorldResource(Resource):
    def hello_world(self):
        return "Hello, World!"

    def create_url_rules(self):
        return [
            route("GET", "/", self.hello_world),
        ]

app = Flask('test')
app.config.update({
    "RESOURCE_CONFIG": Config()
})
resource = Resource(app.config["RESOURCE_CONFIG"])
app.register_blueprint(resource.as_blueprint())
```

1.2.1 Larger example

Below is a large example that demonstrates:

- Response handling via content negotiation.
- Error handling and mapping of business-level exceptions to JSON errors.
- Request parsing from the body content, URL query string, headers and view args.
- Accessing the resource request context

```
class Config(ResourceConfig):
    # Response handlers defines possible mimetypes for content
    # negotiation
    response_handlers = {
        "application/json: ResponseHandler(JSONSerializer()),
        # ...
    }

class MyResource(Resource):

    # Error handlers maps exceptions to JSON errors.
    error_handlers = {
        ma.ValidationError: create_error_handler(
            HTTPJSONException(code=400),
        )
    }

    decorators = [
        # You can apply decorators to all views
        login_required,
    ]
```

(continues on next page)

(continued from previous page)

```
    @request_parser(
        {'q': ma.fields.String(required=True)},
        # Other locations include args, view_args, headers.
        location='args',
    )
    @response_handler(many=True)
    def search(self):
        # The validated data is available in the resource request context.
        if resource_requestctx.args['q']:
            # ...
        # From the view you can return an object which the response handler
        # will serialize.
        return [], 200

    # You can parse request body depending on the Content-Type header.
    @request_body(
        parsers={
            "application/json": RequestBodyParser(JSONDeserializer())
        }
    )
    @response_handler()
    def create(self):
        return {}, 201

    # All decorators all values to come from the conf.
    @request_parser(from_conf('update_args), location='args')
    def update(self):
        return {}, 201

    def create_url_rules(self):
        return [
            route('GET', "/", self.search),
            route('POST', "/", self.create),
            route('PUT', "/<pid_value>", self.update),
            # You can selectively disable global decorators.
            route('DELETE', "/<pid_value>", self.delete, apply_decorators=False),
        ]
```

CHAPTER

TWO

API REFERENCE

If you are looking for information on a specific function, class or method, this part of the documentation is for you.

2.1 API Docs

2.1.1 Resources

Resource view.

class flask_resources.resources.**Resource**(*config*)

Resource interface.

A resource is a factory for creating Flask Blueprint that's parameterized via a config.

Initialize the base resource.

as_blueprint(***options*)

Create the blueprint with all views and error handlers.

The method delegates to `create_blueprint()`, `create_url_rules()` and `create_error_handlers()` so usually you don't have to overwrite this method.

create_blueprint(***options*)

Create the blueprint.

Override this function to customize the creation of the `Blueprint` object itself.

create_error_handlers()

Create all error handlers for this resource.

This function should return a dictionary that maps an exception or HTTP response code to and error handler function. By default it merges error handlers defined on the resource itself with error handlers defined in the config. The error handlers in the config takes precedence over the resource defined error handlers.

The error handlers are registered on the blueprint using the `Blueprint.register_error_handler()`.

create_url_rules()

Create all the blueprint URL rules for this resource.

The URL rules are registered on the blueprint using the `Blueprint.add_url_rule()`.

decorators = [<function with_content_negotiation.<locals>.decorator>]

Decorators applied to all view functions.

By default, the resource request context and content negotiation is enabled. Provide an empty list to disable them.

error_handlers = {}

Mapping of exceptions or HTTP codes to error handler functions.

By default this mapping is merged with the error handlers mapping defined in the config.

class flask_resources.resources.**ResourceConfig**

Configuration for a resource.

This object is used for dependency injection in a resource.

blueprint_name = None

Name of the blueprint being created (used e.g. for prefix endpoint name).

default_accept_mimetype = 'application/json'

The default Accept MIME type if not defined by the request. Set to `None`, to require an Accept header.

default_content_type = 'application/json'

The default content type used to select the default request_body_parser. Set to `None` to require a Content-Type header.

error_handlers = {}

A mapping of exception or HTTP status code to error handler functions.

request_body_parsers = {'application/json': <flask_resources.parsers.body.RequestBodyParser object>}

Request body parser (i.e. `request.data`).

response_handlers = {'application/json': <flask_resources.responses.ResponseHandler object>}

Mapping of Accept MIME types to response handlers.

url_prefix = None

The URL prefix for the blueprint (all URL rules will be prefixed with this value)

flask_resources.resources.**route**(*method*, *rule*, *view_meth*, *endpoint=None*, *rule_options=None*, *apply_decorators=True*)

Create a route.

Use this method in `create_url_rules()` to build your list of rules.

The `view_method` parameter should be a bound method (e.g. `self.myview`).

Parameters

- **method** – The HTTP method for this URL rule.
- **rule** – A URL rule.
- **view_meth** – The view method (a bound method) for this URL rule.
- **endpoint** – The name of the endpoint. By default the name is taken from the method name.
- **rule_options** – A dictionary of extra options passed to `Blueprint.add_url_rule`.
- **apply_decorators** – Apply the decorators defined by the resource. Defaults to `True`. This allows you to selective disable decorators which are normally applied to all view methods.

2.1.2 Context

Resource request context.

The purpose of the resource request context is similar to the Flask request context. The main difference is it serves as a state object that can hold validated request data as well as the result of e.g. content negotiation.

The resource request context is used by default, and when it is used it consumes all the view arguments. These can either be retrieved via a request parser (preferably), or accessing `request.view_args`. The goal of this is to ensure that the view function access only validated data.

class flask_resources.context.**ResourceRequestCtx**(*config*)

Context manager for the resource context.

The resource request context encodes information about the currently executing request for a given resource, such as:

- The mimetype selected by the content negotiation.
- The content type of the request payload

Initialize the resource context.

update(*values*)

Update the context fields present in the received dictionary *values*.

2.1.3 Content negotiation and response handling

Response module.

class flask_resources.responses.**ResponseHandler**(*serializer*, *headers=None*)

Response handler which delegates to the a serializer.

Example usage:

```
def obj_headers(obj_or_list, code, many=False):
    return {'etag': ... }

class Config(ResourceConfig):
    response_handlers = {
        "application/json": ResponseHandler(
            JSONSerializer(), headers=obj_headers)
    }
```

Constructor.

make_headers(*obj_or_list*, *code*, *many=False*)

Builds the headers fo the response.

make_response(*obj_or_list*, *code*, *many=False*)

Builds a response for one object.

flask_resources.responses.**response_handler**(*many=False*)

Decorator for using the response handler to create the HTTP response.

The response handler works in conjunction with `with_content_negotiation()` which is responsible for selecting the correct response handler based on the content negotiation.

```
@response_handler()
def read(self):
    return obj, 200

@response_handler(many=True)
def search(self):
    return [obj], 200
```

Content negotiation API.

class flask_resources.content_negotiation.**ContentNegotiator**

Content negotiation API.

Implements a procedure for selecting a mimetype best matching what the client is requesting.

classmethod match(*mimetypes*, *accept_mimetypes*, *formats_map*, *fmt*, *default=None*)

Select the MIME type which best matches the client request.

Parameters

- **mimetypes** – Iterable of available MIME types.
- **accept_mimetypes** – The client's "Accept" header as MIMEAccept object.
- **formats_map** – Map of format values to MIME type.
- **format** – The client's selected format.
- **default** – Default MIMEtype if a wildcard was received.

classmethod match_by_accept(*mimetypes*, *accept_mimetypes*, *default=None*)

Select the MIME type which best matches Accept header.

NOTE: Our match policy differs from Werkzeug's best_match policy:
If the client accepts a specific mimetype and wildcards, and the server serves that specific mimetype, then favour that mimetype no matter its quality over the wildcard. This is as opposed to Werkzeug which only cares about quality.

Parameters

- **mimetypes** – Iterable of available MIME types.
- **accept_mimetypes** – The client's "Accept" header as MIMEAccept object.
- **default** – Default MIMEtype if wildcard received.

classmethod match_by_format(*formats_map*, *fmt*)

Select the MIME type based on a query parameters.

flask_resources.content_negotiation.**with_content_negotiation**(*response_handlers=None*, *default_accept_mimetype=None*)

Decorator to perform content negotiation.

The result of the content negotiation is stored in the resources request context.

2.1.4 Request body parsing

Request parser for the body, headers, query string and view args.

class flask_resources.parsers.**RequestBodyParser**(*deserializer*)

Parse the request body.

Constructor.

parse()

Parse the request body.

flask_resources.parsers.**request_body_parser**(*parsers={'application/json': <flask_resources.parsers.body.RequestBodyParser object>}*, *default_content_type='application/json'*)

Create decorator for parsing the request body.

Both decorator parameters can be resolved from the resource configuration.

Parameters

- **parsers** – A mapping of content types to parsers.
- **default_content_type_name** – The default content type used to select a parser if no content type was provided.

2.1.5 Request parsing

Request parser for extracting URL args, headers and view args.

The request parser uses a declarative way to extract and validate request parameters. The parser can parse data in three different locations:

- `args`: URL query string (i.e. `request.args`)
- `headers`: Request headers (i.e. `request.headers`)
- `view_args`: Request view args (i.e. `request.view_args`)

The parser is not meant to parse the request body. For that you should use the `RequestBodyParser`.

The request parser can accept both a schema or a dictionary. Using the schema enables you to do further pre/post-processing of values, while the dict version can be more compact.

Example with schema:

```
class MyHeaders(ma.Schema):
    content_type = ma.fields.String()

parser = RequestParser(MyHeaders, location='headers')
parser.parse()
```

Same example with dict:

```
parser = RequestParser({
    'content_type': ma.fields.String()
}, location='headers')
parser.parse()
```

URL args parsing

If you are parsing URL args, be aware that a query string can have repeated variables (e.g. in `?type=a&type=b` the value `type` is repeated).

Thus if you build your own schema for URL args, you should inherit from `MultiDictSchema`. If you don't have repeated keys you can use a normal Marshmallow schema.

Unknown values

If you pass a dict for the schema, you can control what to do with unknown values:

```
parser = RequestParser({
    'id': ma.fields.String()
}, location='args', unknown=ma.RAISE)
parser.parse()
```

If you build your own schema, the same can be achieved with by providing the meta class:

```
class MyArgs(ma.Schema):
    id = ma.fields.String()

    class Meta:
        unknown = ma.INCLUDE
```

class flask_resources.parsers.base.**RequestParser**(*schema_or_dict*, *location*, *unknown='exclude'*)

Request parser.

Constructor.

Parameters

- **schema_or_dict** – A marshmallow schema class or a mapping from keys to fields.
- **location** – Location where to load data from. Possible values: (`args`, `headers`, or `view_args`).
- **unknown** – Determines how to handle unknown values. Possible values: `ma.EXCLUDE`, `ma.INCLUDE`, `ma.RAISE`. Only used if the schema is a dict.

property default_schema_cls

Get the base schema class when dynamically creating the schema.

By default, `request.args` is a MultiDict which a normal Marshmallow schema does not know how to handle, we therefore change the schema only for request args parsing.

load_data()

Load data from request.

property location

The request location for this request parser.

parse()

Parse the request data.

property schema

Build the schema class.

schema_from_dict(*schema_dict*)

Construct a schema from a dict.

Decorator for invoking the request parser.

flask_resources.parsers.decorators.**request_parser**(*schema_or_parser*, *location=None*, ***options*)

Create decorator for parsing the request.

Both decorator parameters can be resolved from the resource configuration.

Parameters

- **schema_or_parser** – A mapping of content types to parsers.
- **default_content_type_name** – The default content type used to select a parser if no content type was provided.

2.1.6 Errors

Exceptions used in Flask Resources module.

exception flask_resources.errors.**HTTPJSONException**(*code=None*, *errors=None*, ***kwargs*)

HTTP Exception delivering JSON error responses.

Initialize HTTPJSONException.

get_body(*environ=None*, *scope=None*)

Get the request body.

get_description(*environ=None*, *scope=None*)

Returns an unescaped description.

get_errors()

Get errors.

Returns
A list containing the errors.

get_headers(*environ=None*, *scope=None*)

Get a list of headers.

property name

The status name.

exception flask_resources.errors.**InvalidContentType**(*allowed_mimetypes=None*, ***kwargs*)

Error for when an invalid *Content-Type* header is provided.

Initialize exception.

exception flask_resources.errors.**MIMETypeException**(*allowed_mimetypes=None*, ***kwargs*)

Error for when an invalid Content-Type is provided.

Initialize exception.

exception flask_resources.errors.**MIMETypeNotAccepted**(*allowed_mimetypes=None*, ***kwargs*)

Error for when an invalid *Accept* header is provided.

Initialize exception.

flask_resources.errors.**create_error_handler**(*map_func_or_exception*)

Creates a resource error handler.

The handler is used to map business logic exceptions to REST exceptions. The original exceptions is being stored in the *__original_exc__* attribute of the mapped exception.

Parameters

map_func_or_exception – Function or exception to map originally raised exception to a *flask_resources.errors.HTTPJSONException*.

2.1.7 Serializers/deserializers

Serializers.

class flask_resources.serializers.**BaseSerializer**

Serializer Interface.

abstract serialize_object(*obj*)

Serialize a single object according to the response ctx.

serialize_object_list(*obj_list*)

Serialize a list of objects according to the response ctx.

class flask_resources.serializers.**BaseSerializerSchema**(*dumpers=None*, ***kwargs*)

Enables the extension of Marshmallow schemas serialization.

Constructor.

post_dump_pipeline(*data*, *original*, *many*, ***kwargs*)

Applies a sequence of post-dump steps to the serialized data.

Parameters

- **data** – The result of serialization.
- **original** – The original object that was serialized.
- **many** – Whether the serialization was done on a collection of objects.

Returns

The result of the pipeline processing on the serialized data.

pre_dump_pipeline(*data*, *many*, ***kwargs*)

Applies a sequence of pre-dump steps to the input data.

Parameters

- **data** – The result of serialization.
- **many** – Whether the serialization was done on a collection of objects.

Returns

The result of the pipeline processing on the serialized data.

class flask_resources.serializers.**DumperMixin**

Abstract class that defines an interface for pre_dump and post_dump methods.

It allows to extend records serialization.

post_dump(*data*, *original=None*, ***kwargs*)

Hook called after the marshmallow serialization of the record.

Parameters

- **data** – The dumped record data.
- **original** – The original record data.
- **kwargs** – Additional keyword arguments.

Returns
The serialized record data.

pre_dump(*data*, *original=None*, ***kwargs*)

Hook called before the marshmallow serialization of the record.

Parameters

- **data** – The record data to dump.
- **original** – The original record data.
- **kwargs** – Additional keyword arguments.

Returns
The data to dump.

class flask_resources.serializers.**JSONSerializer**(*encoder=None*, *options=None*)

JSON serializer implementation.

Initialize the JSONSerializer.

property dumps_options

Support adding options for the dumps() method.

property encoder

Support overriding the JSONEncoder used for serialization.

serialize_object(*obj*)

Dump the object into a json string.

serialize_object_list(*obj_list*)

Dump the object list into a json string.

class flask_resources.serializers.**MarshmallowSerializer**(*format_serializer_cls*, *object_schema_cls*, *list_schema_cls=None*, *schema_context=None*, *schema_kwargs=None*, ***serializer_options*)

Marshmallow serializer that serializes an obj into defined schema.

Parameters

- **format_serializer_cls** – Serializer in charge of converting the data object into the desired format.
- **object_schema_cls** – Marshmallow Schema of the object.
- **list_schema_cls** – Marshmallow Schema of the object list.
- **schema_context** – Context of the Marshmallow Schema.
- **schema_kwargs** – Additional arguments to be passed to marshmallow schema.

Initialize the serializer.

dump_list(*obj_list*)

Dump the list of objects.

dump_obj(*obj*)

Dump the object using object schema class.

serialize_object(*obj*)

Dump the object using the serializer.

serialize_object_list(*obj_list*)

Dump the object list using the serializer.

class flask_resources.serializers.**SimpleSerializer**(*encoder*)

Simple serializer implementation.

Initialize the SimpleSerializer.

serialize_object(*obj*, ***kwargs*)

Dump the object into a string using the encoder function.

serialize_object_list(*obj_list*, ***kwargs*)

Dump the object list into a string separated by new lines.

Deserializers.

class flask_resources.deserializers.**DeserializerMixin**

Deserializer Interface.

deserialize(*data*)

Deserializes the data into an object.

class flask_resources.deserializers.**JSONDeserializer**

JSON Deserializer.

deserialize(*data*)

Deserializes JSON into a Python dictionary.

CHAPTER

THREE

ADDITIONAL NOTES

Notes on how to contribute, legal information and changes are here for the interested.

3.1 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

3.1.1 Types of Contributions

Report Bugs

Report bugs at https://github.com/inveniosoftware/flask-resources/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with "feature" is open to whoever wants to implement it.

Write Documentation

Flask-Resources could always use more documentation, whether as part of the official Flask-Resources docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/inveniosoftware/flask-resources/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

3.1.2 Get Started!

Ready to contribute? Here's how to set up *flask-resources* for local development.

1. Fork the *inveniosoftware/flask-resources* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/flask-resources.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv flask-resources
$ cd flask-resources/
$ pip install -e .[all]
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass tests:

```
$ ./run-tests.sh
```

   The tests will provide you with test coverage and also check PEP8 (code style), PEP257 (documentation), flake8 as well as build the Sphinx documentation and run doctests.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -s
    -m "component: title without verbs"
    -m "* NEW Adds your new feature."
    -m "* FIX Fixes an existing issue."
    -m "* BETTER Improves and existing feature."
    -m "* Changes something that should not be visible in release notes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

3.1.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests and must not decrease test coverage.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring.
3. The pull request should work for Python 2.7, 3.5 and 3.6. Check https://travis-ci.org/inveniosoftware/flask-resources/pull_requests and make sure that the tests pass for all supported Python versions.

3.2 Changes

Version 1.1.0 (released 2023-04-17)

- Serializers: add marshmallow schema processors

Version 1.0.0 (released 2023-03-09)

- Remove MarshmallowJSONSerializer (deprecated).
- Remove XMLSerializer in favor of SimpleSerializer with encoder function.
- Remove SerializerMixin in favor of BaseSerializer interface.
- Replace flask.JSONEncoder by json.JSONEncoder.

Version 0.9.1 (released 2023-02-24)

- Fix bug on XML object and object list serialization formatting.

Version 0.9.0 (released 2023-02-24)

- Add deprecation warning to MarshmallowJSONSerializer.
- Add support for XML serialization formatting.

Version 0.1.0 (released TBD)

- Initial public release.

3.3 License

3.4 Authors

Flask Resources module to create REST APIs

- CERN <info@inveniosoftware.org>

PYTHON MODULE INDEX

f

INDEX

U

W