Rafter Documentation

Release 1.1.0

Olivier Meunier

Apr 12, 2018

Contents

1 Overview **1**

2 Not solving every problem **3**

3 Example **5**
- 3.1 Getting started . . . 5
- 3.2 Resources and Responses . . . 7
- 3.3 Filters and Error Handlers . . . 8
- 3.4 Blueprints . . . 11
- 3.5 Schema validation with Schematics . . . 12
- 3.6 API Reference . . . 16

4 All examples **23**

5 Contribute! **25**

6 Miscellaneous links **27**

Python Module Index **29**

CHAPTER 1

Overview

Rafter is a Python 3.5+ library providing building blocks for Restfull APIs. Yes, it's yet another framework trying, again, to solve the same problem!

Rafter is built on top of Sanic, an asynchronous and blazingly fast HTTP Python framework.

CHAPTER 2

Not solving every problem

A Restfull framework tries to solve specific problems with the unwritten protocol that is REST and Rafter is no exception. However, its main goals are to **provide a good user interface** (the Python API) and to **solve as few problems as possible**.

A solid API is key; it must be consistent, clear, easy to learn and use. Rafter kits you with a few classes and functions to help you create a great API but will do its best to get out of your way and let you do what needs to be done. You (the developer) should have fun coding whatever project you're coding instead of fighting and twisting your framework.

That said, Rafter provides some facilities to handle the very common use cases that come with writing a Restfull API:

- Declare your resources,
- Provide filtering and transformation routines,
- Handle errors with clear, extendible, structured data.

And that's it! The rest is up to you; bring your ideas, your favourite ORM, write your own filters. Have fun!

CHAPTER 3

Example

```
# -*- coding: utf-8 -*-
from rafter import Rafter

app = Rafter()


@app.resource('/')
async def main_view(request):
    # This simple view returns a JSON response
    # with the following content.
    return {
        'data': 'It works!'
    }


if __name__ == "__main__":
    app.run(host="127.0.0.1", port=5000)
```

Now, let's see more examples and usage instruction in *the next part*.

3.1 Getting started

3.1.1 Installation

> **Warning:** Rafter works with python 3.5 or higher.

Install Rafter with pip (in a virtualenv or not):

```
pip install rafter
```

If you'd like to test and tamper the examples, clone and install the project:

```
git clone https://github.com/olivier-m/rafter.git
pip install -e ./rafter
```

3.1.2 First basic application

Our first application is super simple and only illustrates the ability to directly return arbitrary data as a response, and raise errors.

Listing 3.1: examples/simple.py

```
# -*- coding: utf-8 -*-
from rafter import Rafter, ApiError, Response

# Our main Rafter App
app = Rafter()


@app.resource('/')
async def main_view(request):
    # Simply return arbitrary data and the response filter
    # will convert it to a sanic.response.json response.
    return {
        'data': 'Hello there!'
    }


@app.resource('/p/<param>')
async def with_params(request, param):
    # Just return the request's param in a list.
    return [param]


@app.resource('/status')
async def status(request):
    # Return a 201 response with some data
    return Response({'test': 'abc'}, 201)


@app.resource('/error')
async def error_response(request):
    # Return an error response with a status code and some extra data.
    raise ApiError('Something bad happened!', 501,
                   extra_data=':(')


if __name__ == "__main__":
    app.run(host="127.0.0.1", port=5000)
```

Tip: If you cloned the repository, you'll find the examples in the `rafter/examples` folder. The next given command will take place in your `rafter` directory.

Launch this simple app with:

```
python examples/simple.py
```

Now, in another terminal, let's call the API:

```
curl http://127.0.0.1:5000/
```

You'll receive a response:

```
{"data":"Hello there!"}
```

Then you can try the following API endpoints and see what it returns:

```
curl http://127.0.0.1:5000/p/test-param
curl -v http://127.0.0.1:5000/status
curl -v http://127.0.0.1:5000/error
```

Tip: To ease your tests, I strongly advise you to use a full-featured HTTP client. Give Insomnia a try; it's a very good client with many options.

Now, let's see in the next part what we can do with *routing and responses*.

3.2 Resources and Responses

3.2.1 Resource routes

Routing a resource with Rafter is the same thing as adding a route in Sanic except that we do it with the `resource` decorator of the Rafter App instance.

In this example, `app` is an instance of the `Rafter` class.

```
@app.resource('/')
async def test(request):
    return {'hello': 'world'}
```

The `resource` decorator takes the same arguments as Sanic `route` and, some extra arguments that are used by *filters*.

See also:

rafter.app.Rafter

3.2.2 rafter.http.Response

The Rafter `Response` is a specialized Sanic `HTTPResponse`. It acts almost in the same way except that it takes arbitrary data as input and serializes the response's body at the very last moment.

You can use it to return a specific status code:

```
from rafter import Response

@app.resource('/')
async def test(request):
    return Response({'hello': 'world'}, 201)
```

Note: When you return arbitrary data from a resource, Rafter will convert it to a `Response` instance.

See also:

rafter.http.Response

3.3 Filters and Error Handlers

3.3.1 Filters

Filters are like middlewares but applied to a specific resource. They have an API similar to what Django offers.

Here's a basic prototype of a filter:

```
def basic_filter(get_response, params):
    # get_response is the view function or the next filter on the chain
    # params are the resource parameters

    # This part is called during the resource initialization.
    # You can configure, for instance, things based on params values.

    async def decorated_filter(request, *args, **kwargs):
        # Pre-Response code. You can change request attributes,
        # raise exceptions, call another response function...

        # Processing resource (view)
        result = await get_response(request, *args, **kwargs)

        # Post-Response code. You can change the response attributes,
        # raise exceptions, log things...

    # Don't forget this!
    return decorated_filter
```

A filter is a decorator function. It must return an asynchronous callable that will handle the request and will return a response or the result of the `get_response` function.

On Rafter' side, you can pass the '' filters'' or the `validators` parameter (both lists) to *rafter.app.Rafter.resource()*.

Each filter will then be chained to the other, in their order of declaration.

Important: The `Rafter` class has one default validator: `filter_transform_response` that transforms the response when possible.

If you pass the `filters` argument to your resource, you'll override the default filters. If that's not what you want, you can pass the `validators` argument instead. These filters will then be chained to the default filters.

See also:

- *rafter.app.Rafter.resource()*
- *rafter.filters.filter_transform_response()*

Example

The following example demonstrates two filters. The first one changes the response according to the value of `?action` in the query string. The second serializes data to plist format when applicable.

Listing 3.2: examples/filters.py

```
# -*- coding: utf-8 -*-
import plistlib

from sanic.exceptions import abort
from sanic.response import text, HTTPResponse

from rafter import Rafter

# Our primary Rafter App
app = Rafter()


# Input filter
def basic_filter(get_response, params):
    # This filter changes the response according to the
    # request's GET parameter "action".
    async def decorated_filter(request, *args, **kwargs):
        # When ?action=abort
        if request.args.get('action') == 'abort':
            abort(500, 'Abort!')

        # When ?action=text
        if request.args.get('action') == 'text':
            return text('test response')

        # Go on with the request
        return await get_response(request, *args, **kwargs)

    return decorated_filter


# Output filter
def output_filter(get_response, params):
    # This filter is going to serialize our data into a plist value
    # and send the result as a application/plist+xml response
    # if the request's Accept header is correctly set.
    async def decorated_filter(request, *args, **kwargs):
        response = await get_response(request, *args, **kwargs)

        # Don't do it like that please!
        accept = request.headers.get('accept', '*/*')
        if accept != 'application/plist+xml':
            return response

        # Actually, here you should check if you have a Response instance
        # In this example we don't really need to.

        # You accept plist? Here you go!
        return HTTPResponse(plistlib.dumps(response.data).decode('utf-8'),
                            content_type='application/plist+xml')
```

```
    return decorated_filter


@app.resource('/input', validators=[basic_filter])
async def filtered_in(request):
    # See what happens when we add a filter in "validators"
    return request.args


@app.resource('/output', methods=['POST'], validators=[output_filter])
async def filtered_out(request):
    # Return what we received in json
    return request.json


if __name__ == "__main__":
    app.run(host="127.0.0.1", port=5000)
```

3.3.2 Error Handlers

Rafter provides 3 default error handlers and one exception class *rafter.exceptions.ApiError*.

All 3 handlers return a structured JSON response containing at least a `message` and a `status`. If the Rafter app is in debuging mode, they also return a structured stack trace.

Here are the exception classes handled by the default error handlers:

***rafter.exceptions.ApiError*:** This exception's message, status and extra argument are returned.

`sanic.exceptions.SanicException`: This exception's message and status are returned

`Exception`: For any other exception, a fixed error with status **500** and **An error occured.** message.

See also:

- *rafter.exceptions*
- *rafter.app.Rafter.default_error_handlers*

Example

Listing 3.3: examples/errors.py

```
# -*- coding: utf-8 -*-
from sanic.exceptions import abort

from rafter import Rafter, ApiError

app = Rafter()


@app.resource('/')
async def main_view(request):
    # Raising any type of exception
    raise ValueError('Something is wrong!')


@app.resource('/api')
```

```
async def api_error(request):
    # Raising an ApiError with custom code, a message
    # and extra arguments
    raise ApiError('Something went very wrong.', 599, xtra=12,
                   explanation='http://example.net/')


@app.resource('/sanic')
async def sanic_error(request):
    # Using Sanic's abort function
    abort(599, 'A bad error.')


if __name__ == "__main__":
    app.run(host="127.0.0.1", port=5000)
```

3.4 Blueprints

Rafter provides a blueprint API similar to what Sanic provides. The *rafter.blueprints.Blueprint* adds two methods to register resources:

- resource
- add_resource

You must use *rafter.blueprints.Blueprint* if you plan to add resources to your blueprint.

See also:

- *rafter.blueprints*

3.4.1 Example

Listing 3.4: examples/blueprints.py

```
# -*- coding: utf-8 -*-
from rafter import Blueprint, Rafter

bpv1 = Blueprint('v1', url_prefix='/v1')
bpv2 = Blueprint('v2')


def header_filter(get_response, params):
    async def decorated_filter(request, *args, **kwargs):
        response = await get_response(request, *args, **kwargs)
        response.headers['x-test'] = 'abc'
        return response

    return decorated_filter


@bpv1.resource('/')
async def v1_root(request):
    return {'version': 1}
```

```
@bpv1.resource('/test')
async def v1_test(request):
    return [3, 2, 1]


@bpv2.resource('/', validators=[header_filter])
async def v2_root(request):
    return {'version': 2}


app = Rafter()
app.blueprint(bpv1)
app.blueprint(bpv2, url_prefix='/v2')


if __name__ == "__main__":
    app.run(host="127.0.0.1", port=5000)
```

3.5 Schema validation with Schematics

To perform schema validation using Schematic, you need to use the `RafterSchematics` class instead of the `Rafter` one we've seen so far.

```
from rafter.contrib.schematics import RafterSchematics

app = RafterSchematics()
```

This Rafter class adds two new parameters to the **resource** decorator:

- `request_schema`: The request data validation schema
- `response_schema`: The response validation schema

See also:

For more information on the RafterSchematics class and new filters being used, see the *rafter.contrib.schematics.app* module.

3.5.1 Schemas

Request and response schemas are made using Schematic. Let's start with a simple schema:

```
from schematics import Model, types

class BodySchema(Model):
    name = types.StringType(required=True)

class InputSchema(Model):
    body = types.ModelType(BodySchema)
```

In this example, `InputSchema` declares a `body` property that is another schema. Now, let's use it in our view:

```
app = RafterSchematics()

@app.resource('/', ['POST'],
```

```
                request_schema=InputSchema)
async def test(request):
    return request.validated
```

If the input data is not valid, the request to this route will end up with an HTTP 400 error returning a structured error. If everything went well, you can access your processed data with `request.validated`.

Let's say now that we want to return the input body that we received and use a schema to validate the output data. Here's how to do it:

```
app = RafterSchematics()

@app.resource('/', ['POST'],
                request_schema=InputSchema,
                response_schema=InputSchema)
async def test(request):
    return {}
```

In that case, the `response_schema` will fail because `name` is a required field. It will end up with an HTTP 500 error.

3.5.2 The model_node decorator

Having to create many classes and use the `types.ModelType` could be annoying, although convenient at time. Rafter offers a decorator to directly instantiate a sub-node in your schema. Here's how it applies to our `InputSchema`:

```
from schematics import Model, types
from rafter.contrib.schematics import model_node

class InputSchema(Model):
    @model_node()
    class body(Model):
        name = types.StringType(required=True)
```

See also:

rafter.contrib.schematics.helpers.model_node()

3.5.3 Request (input) schema

An request schema is set with the `request_schema` parameter of your resource. It must be a Schematics Model instance with the following, optional, sub schemas:

- `body`: Used to validate your request's body data (form url-encoded or json)
- `params`: To validate the query string parameters
- `path`: To validate data in the path parameters
- `headers`: To validate the request headers

3.5.4 Response (output) schema

A response schema is set with the `response_schema` parameter of your resource. It must be a Schematics Model instance with the following, optional, sub schemas:

- `body`: Used to validate your response body data
- `headers`: To validate the response headers

Important: The response validation is only effective when:

- A `response_schema` has been provided by the resource definition
- The resource returns a *rafter.http.Response* instance or arbitrary data.

3.5.5 Example

Listing 3.5: examples/contrib_schematics.py

```
# -*- coding: utf-8 -*-
from sanic.response import text
from rafter import Response
from rafter.contrib.schematics import RafterSchematics, model_node
from schematics import Model, types

# Let's create our app
app = RafterSchematics()


# -- Schemas
#
class InputSchema(Model):
    @model_node()
    class body(Model):
        # This schema is for our input data (json or form url encoded body)
        # - The name takes a default value
        # - The id has a different name than what will be return in the
        #   resulting validated data
        name = types.StringType(default='')  # Change the default value
        id_ = types.IntType(required=True, serialized_name='id')

    @model_node()
    class headers(Model):
        # This schema defines the request's headers.
        # It this case, we ensure x-test is a positive integer
        # and we provide a default value.
        x_test = types.IntType(serialized_name='x-test', min_value=0,
                               default=0)


class TagSchema(Model):
    @model_node()
    class path(Model):
        # For the sake of the demonstration, because it would be easier
        # to do that in the route definition.
        tag = types.StringType(regex=r'^[a-z]+$')

    @model_node()
    class params(Model):
        # Request's GET parameters validation
        sort = types.StringType(default='asc', choices=('asc', 'desc'))
```

```
        page = types.IntType(default=1, min_value=1)


class ReturnSchema(Model):
    @model_node()
    class body(Model):
        # This schema defines the response data format
        # for the return_schema resource.
        name = types.StringType(required=True, min_length=1)

        @model_node(serialized_name='options')  # Let's change the name!
        class params(Model):
            xray = types.BooleanType(default=False)

    @model_node()
    class headers(Model):
        # Validate and set a default returned header
        x_response = types.IntType(serialized_name='x-response', default=5)


# -- API Endpoints
#
@app.route('/')
async def main(request):
    # Classic Sanic route returning a text/plain response
    return text('Hi mate!')


@app.resource('/post', ['POST'],
              request_schema=InputSchema)
async def post(request):
    # A resource which data are going to be validated before processing
    # Then, we'll return the raw body and the validated data
    # We'll return a response with a specific status code
    return Response({
        'raw': request.form or request.json,
        'validated': request.validated
    }, 201)


@app.resource('/tags/<tag>', ['GET'],
              request_schema=TagSchema)
async def tag(request, tag):
    # Validation and returning data directly
    return {
        'args': request.args,
        'tag': tag,
        'validated': request.validated
    }


@app.resource('/return', ['POST'],
              response_schema=ReturnSchema)
async def return_schema(request):
    # Returns the provided data, so you can see what's going on
    # with the response_schema and data transformation
    return request.json
```

```
if __name__ == "__main__":
    app.run(host="127.0.0.1", port=5000)
```

3.6 API Reference

3.6.1 Rafter API

rafter.app

class Rafter(***kwargs*)

Bases: `sanic.app.Sanic`

This class inherits Sanic's default `Sanic` class. It provides an instance of your app, to which you can add resources or regular Sanic routes.

Note: Please refer to Sanic API reference for init arguments.

---

default_filters = [<function filter_transform_response>]

Default filters called on every resource route.

default_error_handlers = ((<class 'rafter.exceptions.ApiError'>, <rafter.exceptions.Ap

Default error handlers. It must be a list of tuples containing the exception type and a callable.

default_request_class = <class 'rafter.http.Request'>

The default request class. If changed, it must inherit from *rafter.http.Request*.

add_resource(*handler*, *uri*, *methods=frozenset({'GET'})*, ***kwargs*)

Register a resource route.

Parameters

- **handler** – function or class instance
- **uri** – path of the URL
- **methods** – list or tuple of methods allowed
- **host** –
- **strict_slashes** –
- **version** –
- **name** – user defined route name for url_for
- **filters** – List of callable that will filter request and response data
- **validators** – List of callable added to the filter list.

Returns function or class instance

resource(*uri*, *methods=frozenset({'GET'})*, ***kwargs*)

Decorates a function to be registered as a resource route.

Parameters

- **uri** – path of the URL

- **methods** – list or tuple of methods allowed
- **host** –
- **strict_slashes** –
- **stream** –
- **version** –
- **name** – user defined route name for url_for
- **filters** – List of callable that will filter request and response data
- **validators** – List of callable added to the filter list.

Returns A decorated function

rafter.blueprints

class Blueprint(**args*, ***kwargs*)

Bases: `sanic.blueprints.Blueprint`

Create a new blueprint.

Parameters

- **name** – unique name of the blueprint
- **url_prefix** – URL to be prefixed before all route URLs
- **strict_slashes** – strict to trailing slash

add_resource(*handler*, *uri*, *methods=frozenset({'GET'})*, *host=None*, *strict_slashes=None*, *version=None*, *name=None*, ***kwargs*)

Create a blueprint resource route from a function.

Parameters

- **uri** – endpoint at which the route will be accessible.
- **methods** – list of acceptable HTTP methods.
- **host** –
- **strict_slashes** –
- **version** –
- **name** – user defined route name for url_for

Returns function or class instance

Accepts any keyword argument that will be passed to the app resource.

resource(*uri*, *methods=frozenset({'GET'})*, *host=None*, *strict_slashes=None*, *stream=False*, *version=None*, *name=None*, ***kwargs*)

Create a blueprint resource route from a decorated function.

Parameters

- **uri** – endpoint at which the route will be accessible.
- **methods** – list of acceptable HTTP methods.
- **host** –
- **strict_slashes** –

- **version** –
- **name** – user defined route name for url_for

Returns function or class instance

Accepts any keyword argument that will be passed to the app resource.

rafter.exceptions

Exceptions

exception ApiError(*message: str*, *status_code: int = 500*, ***kwargs*)
Bases: `sanic.exceptions.SanicException`

data
Returns the internal property `_data`. It can be overriden by specialized inherited exceptions.

to_primitive() → dict
This methods is called by the error handler *ApiErrorHandler* and returns a dict of error data.

Error Handlers

class ExceptionHandler
Bases: `object`

A generic `Exception` handler.

The callable returns a JSON response with structured error data. The original error message is never returned. Use any type of SanicException if you need to do so.

__call__(*request*, *exception*)
If the data's status is superior or equal to 500, the exception is logged, and if the Rafter app runs in debug mode, the statck trace is also returned in the response.

class SanicExceptionHandler
Bases: *rafter.exceptions.ExceptionHandler*

`SanicException` handler.

This handler returns the original error message in its data.

class ApiErrorHandler
Bases: *rafter.exceptions.SanicExceptionHandler*

ApiError handler.

This handler returns all error data returned by *ApiError.to_primitive()*.

rafter.filters

filter_transform_response(*get_response*, *params*)
This filter process the returned response. It does 3 things:

- If the response is a `sanic.response.HTTPResponse` and not a *rafter.http.Response*, return it immediately.
- If the response is not a *rafter.http.Response* instance, turn it to a *rafter.http.Response* instance with the response as data.

- Then, return the Response instance.

As the Response instance is not immediately serialized, you can still validate its data without any serialization / de-serialization penalty.

rafter.http

class Request(**args*, ***kwargs*)
Bases: `sanic.request.Request`

This class is the default *`rafter.app.Rafter`*'s request object that will be transmitted to every route. It adds a `validated` attribute that will contains all of the validated values, if the route uses schemas.

validated
This property can contain the request data after validation and conversion by the filter.

class Response(*body=None*, *status=200*, *headers=None*, *content_type='application/json'*)
Bases: `sanic.response.HTTPResponse`

A response object that you can return in any route. It looks a lot like `sanic.response.json` function except that instead of immediately serialize the data, it just keeps the value. Serialization (to JSON) will only happen when the response's body is retrieved.

Example:

```
@app.resource('/')
def main_route(request):
    return Response({'data': 'some data'})
```

3.6.2 Contrib / Schematics API

rafter.contrib.schematics.app

class RafterSchematics(***kwargs*)
Bases: *`rafter.app.Rafter`*

default_filters = [<function filter_validate_schemas>, <function filter_transform_resp

- Validate request data
- Pass Rafter's default filters
- Validate output data

resource(*uri*, *methods=frozenset({'GET'})*, ***kwargs*)
Decorates a function to be registered as a resource route.

Parameters

- **uri** – path of the URL
- **methods** – list or tuple of methods allowed
- **host** –
- **strict_slashes** –
- **stream** –
- **version** –
- **name** – user defined route name for url_for

- **filters** – List of callable that will filter request and response data
- **validators** – List of callable added to the filter list.
- **request_schema** – Schema for request data
- **response_schema** – Schema for response data

Returns A decorated function

add_resource(*handler*, *uri*, *methods=frozenset({'GET'})*, ***kwargs*)
Register a resource route.

Parameters

- **handler** – function or class instance
- **uri** – path of the URL
- **methods** – list or tuple of methods allowed
- **host** –
- **strict_slashes** –
- **version** –
- **name** – user defined route name for url_for
- **filters** – List of callable that will filter request and response data
- **validators** – List of callable added to the filter list.
- **request_schema** – Schema for request data
- **response_schema** – Schema for response data

Returns function or class instance

rafter.contrib.schematics.exceptions

exception ValidationErrors(*errors: dict*, ***kwargs*)
Bases: *rafter.exceptions.ApiError*

data
Returns a dictionnary containing all the passed data and an item `error_list` which holds the result of *error_list*.

error_list
Returns an error list based on the internal error dict values. Each item contains a dict with `messages` and `path` keys.

Example:

```
>>> errors = {
>>>     'body': {
>>>         'age': ['invalid age'],
>>>         'options': {
>>>             'extra': {
>>>                 'ex1': ['invalid ex1'],
>>>             }
>>>         }
>>>     }
>>> }
>>> e = ValidationErrors(errors)
```

```
>>> e.error_list

[
    {'messages': ['invalid age'],
     'location': ['body', 'age']},
    {'messages': ['invalid ex1'],
     'location': ['body', 'options', 'extra', 'ex1']},
]
```

rafter.contrib.schematics.filters

See also:

- *Request (input) schema*
- *Response (output) schema*

filter_validate_schemas(*get_response*, *params*)

This filter validates input data against the resource's `request_schema` and fill the request's `validated` dict.

Data from `request.params` and `request.body` (when the request body is of a form type) will be converted using the schema in order to get proper lists or unique values.

Important: The request validation is only effective when a `request_schema` has been provided by the resource definition.

---

filter_validate_response(*get_response*, *params*)

This filter process the returned response. It does 2 things:

- If the response is a `sanic.response.HTTPResponse` and not a *`rafter.http.Response`*, return it immediately.
- It processes, validates and serializes this response when a schema is provided.

That means that you can always return a normal Sanic's HTTPResponse and thus, bypass the validation process when you need to do so.

Important: The response validation is only effective when:

- A `response_schema` has been provided by the resource definition
- The resource returns a *`rafter.http.Response`* instance or arbitrary data.

---

rafter.contrib.schematics.helpers

model_node(***kwargs*)

Decorates a `schematics.Model` class to add it as a field of type `schematic.types.ModelType`.

Keyword arguments are passed to `schematic.types.ModelType`.

Example:

```python
from schematics import Model, types
from rafter.contrib.schematics.helpers import model_node


class MyModel(Model):
    name = types.StringType()

    @model_node()
    class options(Model):
        status = types.IntType()

    # With arguments and another name
    @model_node(serialized_name='extra', required=True)
    class _extra(Model):
        test = types.StringType()
```

CHAPTER 4

All examples

- *Simple first API*
- *Error handlers*
- *Request and response filters*
- *Blueprints*
- *Schema validation with Schematics*

CHAPTER 5

Contribute!

- Source Code
- Bug reports

CHAPTER 6

Miscellaneous links

- genindex
- modindex

Python Module Index

r

rafter.app, 16
rafter.blueprints, 17
rafter.contrib.schematics.app, 19
rafter.contrib.schematics.exceptions, 20
rafter.contrib.schematics.filters, 21
rafter.contrib.schematics.helpers, 21
rafter.exceptions, 18
rafter.filters, 18
rafter.http, 19

Index

Symbols

__call__() (ExceptionHandler method), 18

A

add_resource() (Blueprint method), 17
add_resource() (Rafter method), 16
add_resource() (RafterSchematics method), 20
ApiError, 18
ApiErrorHandler (class in rafter.exceptions), 18

B

Blueprint (class in rafter.blueprints), 17

D

data (ApiError attribute), 18
data (ValidationErrors attribute), 20
default_error_handlers (Rafter attribute), 16
default_filters (Rafter attribute), 16
default_filters (RafterSchematics attribute), 19
default_request_class (Rafter attribute), 16

E

error_list (ValidationErrors attribute), 20
ExceptionHandler (class in rafter.exceptions), 18

F

filter_transform_response() (in module rafter.filters), 18
filter_validate_response() (in module rafter.contrib.schematics.filters), 21
filter_validate_schemas() (in module rafter.contrib.schematics.filters), 21

M

model_node() (in module rafter.contrib.schematics.helpers), 21

R

Rafter (class in rafter.app), 16
rafter.app (module), 16
rafter.blueprints (module), 17
rafter.contrib.schematics.app (module), 19
rafter.contrib.schematics.exceptions (module), 20
rafter.contrib.schematics.filters (module), 21
rafter.contrib.schematics.helpers (module), 21
rafter.exceptions (module), 18
rafter.filters (module), 18
rafter.http (module), 19
RafterSchematics (class in rafter.contrib.schematics.app), 19
Request (class in rafter.http), 19
resource() (Blueprint method), 17
resource() (Rafter method), 16
resource() (RafterSchematics method), 19
Response (class in rafter.http), 19

S

SanicExceptionHandler (class in rafter.exceptions), 18

T

to_primitive() (ApiError method), 18

V

validated (Request attribute), 19
ValidationErrors, 20