Flask-Rebar Documentation

Barak Alon et al.

Jan 10, 2022

Guide:

Welcome to Flask-Rebar's documentation!

Flask-Rebar combines flask, marshmallow, and swagger for robust REST services.

CHAPTER 1

Features

- **Request and Response Validation** - Flask-Rebar relies on schemas from the popular Marshmallow package to validate incoming requests and marshal outgoing responses.
- **Automatic Swagger Generation** - The same schemas used for validation and marshaling are used to automatically generate OpenAPI specifications (a.k.a. Swagger). This also means automatic documentation via Swagger UI.
- **Error Handling** - Uncaught exceptions from Flask-Rebar are converted to appropriate HTTP errors.

CHAPTER 2

Example

Here's what a basic Flask-Rebar application looks like:

```
from flask import Flask
from flask_rebar import errors, Rebar
from marshmallow import fields, Schema

from my_app import database


rebar = Rebar()

# All handler URL rules will be prefixed by '/v1'
registry = rebar.create_handler_registry(prefix='/v1')

class TodoSchema(Schema):
    id = fields.Integer()
    complete = fields.Boolean()
    description = fields.String()

# This schema will validate the incoming request's query string
class GetTodosQueryStringSchema(Schema):
    complete = fields.Boolean()

# This schema will marshal the outgoing response
class GetTodosResponseSchema(Schema):
    data = fields.Nested(TodoSchema, many=True)


@registry.handles(
    rule='/todos',
    method='GET',
    query_string_schema=GetTodosQueryStringSchema(),
    response_body_schema=GetTodosResponseSchema(), # For version <= 1.7.0 use marshal_
↪schema
```

(continues on next page)

(continued from previous page)

```
)
def get_todos():
    """
    This docstring will be rendered as the operation's description in
    the auto-generated OpenAPI specification.
    """
    # The query string has already been validated by `query_string_schema`
    complete = rebar.validated_args.get('complete')

    ...

    # Errors are converted to appropriate HTTP errors
    raise errors.Forbidden()

    ...

    # The response will be marshaled by `marshal_schema`
    return {'data': []}


def create_app(name):
    app = Flask(name)
    rebar.init_app(app)
    return app


if __name__ == '__main__':
    create_app(__name__).run()
```

2.1 Why Flask-Rebar?

There are number of packages out there that solve a similar problem. Here are just a few:

- Connexion
- Flask-RESTful
- flask-apispec
- Flasgger

These are all great projects, and one might work better for your use case. Flask-Rebar solves a similar problem with its own its own twist on the approach:

2.1.1 Marshmallow for validation *and* marshaling

Some approaches use Marshmallow only for marshaling, and provide a secondary schema module for request validation.

Flask-Rebar is Marshmallow first. Marshmallow is a well developed, well supported package, and Flask-Rebar is built on top of it from the get go.

2.1.2 Swagger as a side effect

Some approaches generate code *from* a Swagger specification, or generate Swagger from docstrings. Flask-Rebar aims to make Swagger (a.k.a. OpenAPI) a byproduct of writing application code with Marshmallow and Flask.

This is really nice if you prefer the rich validation/transformation functionality of Marshmallow over Swagger's limited.

It also alleviates the need to manually keep an API's documentation in sync with the actual application code - the schemas used by the application are the same schemas used to generate Swagger.

It's also not always practical - Flask-Rebar sometimes has to expose some Swagger specific things in its interface. C'est la vie.

And since Marshmallow can be more powerful than Swagger, it also means its possible to have validation logic that can't be represented in Swagger. Flask-Rebar assumes this is inevitable, and assumes that it's OK for an API to raise a 400 error that Swagger wasn't expecting.

2.2 Installation

Flask-Rebar can be installed with `pip`:

```
pip install flask-rebar
```

Flask-Rebar depends on Flask and Marshmallow. Flask has fantastic documentation on setting up a development environment for Python, so if you're new to this sort of thing, check that out.

2.3 Basics

2.3.1 Registering a Handler

Let's first take a look at a very basic API using Flask-Rebar. For these examples we will use basic marshmallow Schemas. As of flask-rebar 2.0, we now have support for marshmallow-objects as well, which we'll describe after the basic examples.

```
from flask import Flask
from flask_rebar import Rebar
from flask_rebar import ResponseSchema
from marshmallow import fields


rebar = Rebar()
registry = rebar.create_handler_registry()


class TodoSchema(ResponseSchema):
    id = fields.Integer()


@registry.handles(
   rule='/todos/<id>',
   method='GET',
   response_body_schema=TodoSchema()  # for versions <= 1.7.0, use marshal_schema
```

(continues on next page)

(continued from previous page)

```
)
def get_todo(id):
    ...
    return {'id': id}

app = Flask(__name__)
rebar.init_app(app)

if __name__ == '__main__':
    app.run()
```

We first create a `Rebar` instance. This is a Flask extension and takes care of attaching all the Flask-Rebar goodies to the Flask application.

We then create a handler registry that we will use to declare handlers for the service.

`ResponseSchema` is an extension of `marshmallow.Schema` that throws an error if additional fields not specified in the schema are included in the request parameters. It's usage is optional - a normal Marshmallow schema will also work.

`rule` is the same as Flask's rule, and is the URL rule for the handler as a string.

`method` is the HTTP method that the handler will accept. To register multiple methods for a single handler function, decorate the function multiple times.

`response_body_schema` is a Marshmallow schema that will be used marshal the return value of the function. marshmallow.Schema.dump will be called on the return value. `response_body_schema` can also be a dictionary mapping status codes to Marshmallow schemas - see *Marshaling*. *NOTE: In Flask-Rebar 1.0-1.7.0, this was referred to as ‘‘marshal_schema‘‘. It is being renamed and both names will function until version 2.0*

The handler function should accept any arguments specified in `rule`, just like a Flask view function.

When calling `Rebar.init_app`, all of the handlers for all the registries created by that rebar instance will be registered with the Flask application. Each registry will get its own Swagger specification and Swagger UI endpoint. This is intended as one way of doing API versioning - see *API Versioning* for more info.

2.3.2 Request Body Validation

```
from flask_rebar import RequestSchema


class CreateTodoSchema(RequestSchema):
    description = fields.String(required=True)


@registry.handles(
   rule='/todos',
   method='POST',
   request_body_schema=CreateTodoSchema(),
)
def create_todo():
    body = rebar.validated_body
    description = body['description']
    . . .
```

`RequestSchema` is an extension of `marshmallow.Schema` that throws an internal server error if an object is missing a required field. It's usage is optional - a normal Marshmallow schema will also work.

This request schema is passed to `request_body_schema`, and the handler will now call marshmallow.Schema.load on the request body decoded as JSON. A 400 error with a descriptive error will be returned if validation fails.

The validated parameters are available as a dictionary via the `rebar.validated_body` proxy.

2.3.3 Query String Validation

```
class GetTodosSchema(RequestSchema):
    exclude_completed = fields.String(missing=False)


@registry.handles(
   rule='/todos',
   method='GET',
   query_string_schema=GetTodosSchema(),
)
def get_todos():
    args = rebar.validated_args
    exclude_completed = args['exclude_completed']
    . . .
```

This request schema is passed to `query_string_schema`, and the handler will now call marshmallow.Schema.load on the query string parameters retrieved from Flask's `request.args`. A 400 error with a descriptive error will be returned if validation fails.

The validated parameters are available as a dictionary via the `rebar.validated_args` proxy.

`request_body_schema` and `query_string_schema` behave very similarly, but keep in mind that query strings can be a bit more limited in the amount of data that can be (or rather, should be) encoded in them, so the schemas for query strings should aim to be simpler.

2.3.4 Header Parameters

```
from marshmallow import Schema


class HeadersSchema(Schema):
    user_id = fields.String(required=True, load_from='X-MyApp-UserId')


@registry.handles(
   rule='/todos/<id>',
   method='PUT',
   headers_schema=HeadersSchema(),
)
def update_todo(id):
    headers = rebar.validated_headers
    user_id = headers['user_id']
    . . .
```

Note: In version 3 of Marshmallow, The *load_from* parameter of fields changes to *data_key*

In this case we use a regular Marshmallow schema, since there will almost certainly be other HTTP headers in the request that we don't want to validate against.

This schema is passed to `headers_schema`, and the handler will now call marshmallow.Schema.load on the header values retrieved from Flask's `request.headers`. A 400 error with a descriptive error will be returned if validation fails.

The validated parameters are available as a dictionary via the `rebar.validated_headers` proxy.

A schema can be added as the default headers schema for all handlers via the registry:

```
registry.set_default_headers_schema(HeadersSchema())
```

This default can be overriden in any particular handler by setting `headers_schema` to something else, including `None` to bypass any header validation.

2.3.5 Marshaling

Note: In version 2.0, we updated our supported versions of Marshmallow from 2.x to 3.x. (The Marshmallow maintainers stopped supporting 2.x on 2020-08-18.) One of the more significant changes in Marshmallow v3 is that `Schema.dump` does not trigger validation. This can result in significant performance improvements. In Flask-Rebar 2.0, we have made validation of marshalled results *opt-in*.

The `response_body_schema` (previously `marshal_schema`) argument of `HandlerRegistry.handles` can be one of three types: a `marshmallow.Schema`, a dictionary mapping integers to `marshmallow.Schema`, or `None`.

In the case of a `marshmallow.Schema`, that schema is used to `dump` the return value of the handler function.

In the case of a dictionary mapping integers to `marshmallow.Schemas`, the integers are interpreted as status codes, and the handler function must return a tuple of `(response_body, status_code)`:

```
@registry.handles(
   rule='/todos',
   method='POST',
   response_body_schema={
       201: TodoSchema()
   }
)
def create_todo():
    ...
    return {'id': id}, 201
```

The schema to use for marshaling will be retrieved based on the status code the handler function returns. This isn't the prettiest part of Flask-Rebar, but it's done this way to help with the automatic Swagger generation.

In the case of `None` (which is also the default), no marshaling takes place, and the return value is passed directly through to Flask. This means the if `response_body_schema` is `None`, the return value must be a return value that Flask supports, e.g. a string or a `Flask.Response` object.

```
@registry.handles(
   rule='/todos',
   method='GET',
   response_body_schema=None
)
def get_todos():
```

(continues on next page)

(continued from previous page)

```
    ...
    return 'Hello World!'
```

This is a handy escape hatch when handlers don't fit the Swagger/REST mold very well, but it the swagger generation won't know how to describe this handler's response and should be avoided.

Opting In to Response Validation

There are two ways to opt-in to response validation:

1. Globally, via `validate_on_dump` attribute of your `Rebar` instance. Using this method, it is easy to turn on validation for things like test cases, while reaping performance gains by leaving it off in your production endpoints (assuming your API contract testing is sufficient to guarantee that your API can't return invalid data).
2. At schema level, via `flask_rebar.validation.RequireOnDumpMixin` (including if you use our legacy pre-canned `ResponseSchema` as the base class for your schemas). Any schema that includes that mixin is automatically opted in to response validation, regardless of global setting. Note that in Flask-Rebar 2, that mixin serves *only* as a "marker" to trigger validation; we plan to augment/replace this with ability to use *SchemaOpts* as a more logical way of accomplishing the same thing in the near future (https://github.com/plangrid/flask-rebar/issues/252).

2.3.6 Errors

Flask-Rebar includes a set of error classes that can be raised to produce HTTP errors.

```
from flask_rebar import errors

@registry.handles(
   rule='/todos/<id>',
   method='GET',
)
def get_todo(id):
    if not user_allowed_to_access_todo(
            user_id=rebar.validated_headers['user_id'],
            todo_id=id
    ):
        raise errors.Forbidden(
            msg='User not allowed to access todo object.',
            additional_data={
                'my_app_internal_error_code': 123
            }
        )
    ...
```

The `msg` parameter will override the "message" key of the JSON response. Furthermore, the JSON response will be updated with `additional_data`.

Validation errors are raised automatically, and the JSON response will include an `errors` key with more specific errors about what in the payload was invalid (this is done with the help of Marshmallow validation).

For most of our predefined errors, as of version 2.0 we include not just a message but also a "rebar-internal" error "code". By default this is included in those responses as `rebar_error_code` but you can control that by setting the `error_code_attr` attribute on your instance of `Rebar` to your preferred name, or to `None` to suppress inclusion of rebar-internal error codes entirely.

2.3.7 Support for marshmallow-objects

New and by request in version 2.0, we include some support for `marshmallow-objects`!

CAVEAT: We do not have a dependency on `marshmallow-objects` outside of `dev` extras. If you're developing a flask-rebar app that depends on `marshmallow-objects`, be sure to include it in your explicit dependencies, and be aware that `flask-rebar` is only tested with 2.3.x versions.

In many cases, you can just use a `Model` where you would use a `Schema`, but there are a couple of things to look out for:

- In many places throughout `flask-rebar`, when you need to provide a schema (for example, when registering a handler), you can pass either your `Schema` *class or an instance of it* and rebar does the rest. This is also true of `Model`; however, you can't instantiate a `Model` without providing data if there are required fields. We recommend just passing relevant `Model` subclasses consistently.
- When generating OpenAPI specification, if you use `marshmallow.Schema` classes, they are represented in OpenAPI by their class name. If you use `marshmallow_objects.Model` classes, they are represented as the class name **with a suffix** of "Schema". Note that you can use `__swagger_title__` to override this and call them whatever you want.
- `NestedModel` is supported, but there is not a good way to specify a "title" for OpenAPI generation. If you need to provide custom titles for your nested models, use `flask_rebar.utils.marshmallow_objects_helpers.NestedTitledModel`

2.4 API Versioning

2.4.1 URL Prefixing

There are many ways to do API versioning. Flask-Rebar encourages a simple and very common approach - URL prefixing.

```
from flask import Flask
from flask_rebar import Rebar


rebar = Rebar()
v1_registry = rebar.create_handler_registry(prefix='/v1')
v2_registry = rebar.create_handler_registry(prefix='/v2')

@v1_registry.handles(rule='/foos')
@v2_registry.handles(rule='/foos')
def get_foos():
    ...

@v1_registry.handles(rule='/bar')
def get_bars():
    ...

@v2_registry.handles(rule='/bar')
def get_bars():
    ...

app = Flask(__name__)
rebar.init_app(app)
```

Here we have two registries, and both get registered when calling `rebar.init_app`.

The same handler function can be used for multiple registries.

This will produce a separate Swagger specification and UI instance per API version, which Flask-Rebar encourages for better support with tools like swagger-codegen.

2.4.2 Cloning a Registry

While its recommended to start versioning an API from the get go, sometimes we don't. In that case, it's a common practice to assume that no version prefix is the same as version 1 of the API in order to maintain backwards compatibility for clients that might already be calling non-prefixed endpoints.

Flask-Rebar supports copying an entire registry and changing the URL prefix:

```
from flask import Flask
from flask_rebar import Rebar


rebar = Rebar()
registry = rebar.create_handler_registry()

@registry.handles(rule='/foos')
def get_foos():
    ...

v1_registry = registry.clone()
v1_registry.prefix = '/v1'
rebar.add_handler_registry(v1_registry)

app = Flask(__name__)
rebar.init_app(app)
```

2.5 Swagger Generation

Changed in 2.0: Deprecated functions that supported attaching a "converter function" for a custom authenticator to a generator were removed. We now only support a "registry of converters" approach (consistent with approaches used elsewhere in Flask-rebar)

2.5.1 Swagger Endpoints

Flask-Rebar aims to make Swagger generation and documentation a side effect of building the API. The same Marshmallow schemas used to actually validate and marshal in the live API are used to generate a Swagger specification, which can be used to generate API documentation and client libraries in many languages.

Flask-Rebar adds two additional endpoints for every handler registry:

- `/<registry prefix>/swagger`
- `/<registry prefix>/swagger/ui`

`/swagger` and `/swagger/ui` are configurable:

```
registry = rebar.create_handler_registry(
    spec_path='/apidocs',
    spec_ui_path='/apidocs-ui'
)
```

The HTML documentation is generated with Swagger UI.

2.5.2 Swagger Version

Flask-Rebar supports both Swagger v2 and Swagger v3 (synonymous with OpenAPI v2 and OpenAPI v3, respectively).

For backwards compatibility, handler registries will generate Swagger v2 by default. To have the registries generate Swagger v3 instead, specify an instance `SwaggerV3Generator` when instantiating the registry:

```
from flask_rebar import SwaggerV3Generator

registry = rebar.create_handler_registry(
    swagger_generator=SwaggerV3Generator()
)
```

2.5.3 Serverless Generation

It is possible to generate the Swagger specification without running the application by using `SwaggerV2Generator` or `SwaggerV3Generator` directly. This is helpful for generating static Swagger specifications.

```
from flask_rebar import SwaggerV3Generator
from flask_rebar import Rebar

rebar = Rebar()
registry = rebar.create_handler_registry()

# ...
# register handlers and what not

generator = SwaggerV3Generator()
swagger = generator.generate(registry)
```

2.5.4 Extending Swagger Generation

Flask-Rebar does its very best to free developers from having to think about how their applications map to Swagger, but sometimes it needs some hints.

`flask_rebar.swagger_generation.SwaggerV2Generator` is responsible for converting a registry to a Swagger specification.

operationId

All Swagger operations (i.e. a combination of a URL route and method) can have an "operationId", which is name that is unique to the specification. This operationId is very useful for code generation tools, e.g. swagger-codegen, that use the operationId to generate method names.

The generator first checks for the value of `endpoint` specified when declaring the handler with a handler registry. If this is not included, the generator defaults to the name of the function.

In many cases, the name of the function will be good enough. If you need more control over the operationId, specific an `endpoint` value.

description

Swagger operations can have descriptions. If a handler function has a docstring, the generator will use this as a description.

definition names

The generator makes use of Swagger "definitions" when representing schemas in the specification.

The generator first checks for a `__swagger_title__` on Marshmallow schemas when determining a name for its Swagger definition. If this is not specified, the generator defaults to the name of the schema's class.

Custom Marshmallow types

The generator knows how to convert most built in Marshmallow types to their corresponding Swagger representations, and it checks for the appropriate converter by iterating through a schema/field/validator's method resolution order, so simple extensions of Marshmallow fields should work out of the box.

If a field the extends Marshmallow's abstract field, or want to a particular Marshmallow type to have a more specific Swagger definition, you can add a customer converter.

Here's an example of a custom converter for a custom Marshmallow converter:

```
import base64

from flask_rebar.swagger_generation import swagger_words
from flask_rebar.swagger_generation.marshmallow_to_swagger import sets_swagger_attr
from flask_rebar.swagger_generation.marshmallow_to_swagger import request_body_
↪converter_registry
from flask_rebar.swagger_generation.marshmallow_to_swagger import StringConverter
from marshmallow import fields, ValidationError


class Base64EncodedString(fields.String):
     def _serialize(self, value, attr, obj):
         return base64.b64encode(value).encode('utf-8')

     def _deserialize(self, value, attr, data):
         try:
             return base64.b64decode(value.decode('utf-8'))
         except UnicodeDecodeError:
             raise ValidationError()


class Base64EncodedStringConverter(StringConverter):
    @sets_swagger_attr(swagger_words.format)
    def get_format(self, obj, context):
        return swagger_words.byte

request_body_converter_registry.register_type(Base64EncodedStringConverter())
```

First we've defined a `Base64EncodedString` that handles serializing/deserializing a string to/from base64. We want this field to be represented more specifically in our Swagger spec with a "byte" format.

We extend the `StringConverter`, which handles setting the "type".

Methods on the new converter class can be decorated with `sets_swagger_attr`, which accepts a single argument for which attribute on the JSON document to set with the result of the method.

The method should take two arguments in addition to `self`: `obj` and `context`. `obj` is the current Marshmallow object being converted. In the above case, it will be an instance of `Base64EncodedString`. `context` is a namedtuple that holds some helpful information for more complex conversions:

- `convert` - This will hold a reference to a convert method that can be used to make recursive calls
- `memo` - This holds the JSONSchema object that's been converted so far. This helps convert Validators, which might depend on the type of the object they are validating.
- `schema` - This is the full schema being converted (as opposed to `obj`, which might be a specific field in the schema).
- `openapi_version` - This is the major version of OpenAPI being converter for

We then add an instance of the new converter to the `request_body_converter_registry`, meaning this field will only be valid for request bodies. We can add it to multiple converter registries or choose to omit it from some if we don't think a particular type of field should be valid in certain situations (e.g. the query_string_converter_registry doesn't support `Nested` fields).

Default response

Another really tricky bit of the Swagger specification to automatically generate is the default response to operations. The generator needs a little hand-holding to get this right, and accepts a `default_response_schema`. By default this is set to a schema for the default error handling response.

To customize it:

```
from marshmallow import Schema, fields
from flask_rebar import SwaggerV2Generator
from flask_rebar import Rebar

class DefaultResponseSchema(Schema):
    text = fields.String()

generator = SwaggerV2Generator(
    default_response_schema=DefaultResponseSchema()
)

rebar = Rebar()
registry = rebar.create_handler_registry(swagger_generator=generator)
```

Notice that since we've started to customize the swagger generator, we should specify the generator instance when instantiating our Registry instance so our swagger endpoints get this same default response.

Authenticators

Changed in 2.0

We also need to tell the generator how to represent custom Authenticators as Swagger.

To create a proper converter:

```
from flask_rebar.swagger_generation import swagger_words as sw
from flask_rebar.swagger_generation.authenticator_to_swagger import␣
↪AuthenticatorConverter

class MyAuthConverter(AuthenticatorConverter):
    AUTHENTICATOR_TYPE=MyAuthenticator
    def get_security_schemes(self, obj, context):
        return {
            obj.name: {sw.type_: sw.api_key, sw.in_: sw.header, sw.name: obj.header}
        }
    def get_security_requirements(self, obj, context):
        return [{obj.name: []}]

auth_converter = MyAuthConverter()
```

The converter function should take an instance of the authenticator as a single positional argument and return a dictionary representing the security schema object.

To convert an old-style function into a new-style converter:

```
from flask_rebar.swagger_generation.authenticator_to_swagger import make_class_from_
↪method

from my_custom_stuff import MyAuthenticator

def my_conversion_function(authenticator):
    return {
        "name": MyAuthenticator._HEADER_NAME,
        "type": "apiKey",
        "in": "header"
    }

auth_converter = make_class_from_method(MyAuthenticator, my_conversion_function)
```

There are two supported methods of registering a custom `AuthenticatorConverter`: You can either instantiate your own registry and pass that in when instantiating the generator:

```
from flask_rebar import SwaggerV3Generator
from flask_rebar.swagger_generation.authenticator_to_swagger import␣
↪AuthenticatorConverterRegistry
from my_custom_stuff import auth_converter

my_auth_registry = AuthenticatorConverterRegistry()
my_auth_registry.register_type(auth_converter)

generator = SwaggerV3Generator(authenticator_converter_registry=my_auth_registry)
```

or, you can register your converter with the global default registry:

```
from flask_rebar.swagger_generation.authenticator_to_swagger import authenticator_
↪converter_registry as global_authenticator_converter_registry
from my_custom_stuff import auth_converter

global_authenticator_converter_registry.register_type(auth_converter)
```

Tags

Swagger supports tagging operations with arbitrary strings, and then optionally including additional metadata about those tags at the root Swagger Object.

Handlers can be tagged, which will translate to tags on the Operation Object:

```
@registry.handles(
   rule='/todos',
   method='GET',
   tags=['beta']
)
def get_todos():
    ...
```

Optionally, to include additional metadata about tags, pass the metadata directly to the swagger generator:

```
from flask_rebar import Tag

generator = SwaggerV2Generator(
    tags=[
        Tag(
            name='beta',
            description='These operations are still in beta!'
        )
    ]
)
```

Servers

OpenAPI 3+ replaces "host" with servers.

Servers can be specified by creating `Server` instances and passing them to the generator:

```
from flask_rebar import Server, ServerVariable

generator = SwaggerV3Generator(
    servers=[
        Server(
            url="https://{username}.gigantic-server.com:{port}/{basePath}",
            description="The production API server",
            variables={
                "username": ServerVariable(
                    default="demo",
                    description="this value is assigned by the service provider, in␣
↪this example `gigantic-server.com`",
                ),
                "port": ServerVariable(default="8443", enum=["8443", "443"]),
                "basePath": ServerVariable(default="v2"),
            },
        )
    ]
)
```

2.6 Authentication

2.6.1 Authenticator Interface

Flask-Rebar has very basic support for authentication - an authenticator just needs to implement `flask_rebar.authenticators.Authenticator`, which is just a class with an `authenticate` method that will be called before a handler function.

2.6.2 Header API Key Authentication

Flask-Rebar ships with a `HeaderApiKeyAuthenticator`.

```
from flask_rebar import HeaderApiKeyAuthenticator

authenticator = HeaderApiKeyAuthenticator(header='X-MyApp-ApiKey')

@registry.handles(
   rule='/todos/<id>',
   method='GET',
   authenticators=authenticator,
)
def get_todo(id):
    ...

authenticator.register_key(key='my-secret-api-key')

# Probably a good idea to include a second valid value to make key rotation
# possible without downtime
authenticator.register_key(key='my-secret-api-key-backup')
```

The `X-MyApp-ApiKey` header must now match `my-secret-api-key`, or else a `401` error will be returned.

This also supports very lightweight way to identify clients based on the value of the api key:

```
from flask import g

authenticator = HeaderApiKeyAuthenticator(header='X-MyApp-ApiKey')

@registry.handles(
   rule='/todos/<id>',
   method='GET',
   authenticators=authenticator,
)
def get_todo(id):
    app_name = authenticator.authenticated_app_name
    if app_name == 'client_1':
        raise errors.Forbidden()
    ...

authenticator.register_key(key='key1', app_name='client_1')
authenticator.register_key(key='key2', app_name='client_2')
```

This is meant to differentiate between a small set of client applications, and will not scale to a large set of keys and/or applications.

An authenticator can be added as the default headers schema for all handlers via the registry:

```
registry.set_default_authenticator(authenticator)
```

This default can be extended for any particular handler by passing flask_rebar.authenticators.USE_DEFAULT as one of the authenticators. This default can be overriden in any particular handler by setting `authenticators` to something else, including `None` to bypass any authentication.

This Header API Key authentication mechanism was designed to work for services behind some sort of reverse proxy that is handling the harder bits of client authentication.

2.6.3 Extensions for Authentication

For situations that require something more robust than the basic header-based authentication, Flask-Rebar can be extended. For example, see the following open-source Flask-Rebar extension(s):

- Flask-Rebar-Auth0 - Auth0 authenticator for Flask-Rebar

2.7 API Reference

This part of the documentation covers most of the interfaces for Flask-Rebar.

2.7.1 Rebar Extension

class flask_rebar.**Rebar**

The main entry point for the Flask-Rebar extension.

This registers handler registries with the Flask application and initializes all the Flask-Rebar goodies.

Example usage:

```
app = Flask(__name__)
rebar = Rebar()
registry = rebar.create_handler_registry()

@registry.handles()
def handler():
    ...

rebar.init_app(app)
```

create_handler_registry(*prefix=None*, *default_authenticators=None*, *default_headers_schema=None*, *default_mimetype=None*, *swagger_generator=None*, *spec_path='/swagger'*, *swagger_ui_path='/swagger/ui'*)

Create a new handler registry and add to this extension's set of registered registries.

When calling *Rebar.init_app()*, all registries created via this method will be registered with the Flask application.

Parameters are the same for the *HandlerRegistry* constructor.

Parameters

- **prefix** (*str*) – URL prefix for all handlers registered with this registry instance.

- **Authenticator, None) default_authenticators** (*Union(List(Authenticator),)*) – List of Authenticators to use for all handlers as a default.
- **default_headers_schema** (*marshmallow.Schema*) – Schema to validate the headers on all requests as a default.
- **default_mimetype** (*str*) – Default response content-type if no content and not otherwise specified by the handler.
- **swagger_generator** (*flask_rebar.swagger_generation.swagger_generator.SwaggerGeneratorI*) – Object to generate a Swagger specification from this registry. This will be the Swagger generator that is used in the endpoints swagger and swagger UI that are added to the API. If left as None, a *SwaggerV2Generator* instance will be used.
- **spec_path** (*str*) – The OpenAPI specification as a JSON document will be hosted at this URL. If set as None, no swagger specification will be hosted.
- **swagger_ui_path** (*str*) – The HTML Swagger UI will be hosted at this URL. If set as None, no Swagger UI will be hosted.

Return type *HandlerRegistry*

add_handler_registry(*registry*)
Register a handler registry with the extension.

There is no need to call this if a handler registry was created via *Rebar.create_handler_registry()*.

Parameters **registry** (HandlerRegistry) –

validated_body
Proxy to the result of validating/transforming an incoming request body with the *request_body_schema* a handler was registered with.

Return type dict

validated_args
Proxy to the result of validating/transforming an incoming request's query string with the *query_string_schema* a handler was registered with.

Return type dict

validated_headers
Proxy to the result of validating/transforming an incoming request's headers with the *headers_schema* a handler was registered with.

Return type dict

add_uncaught_exception_handler(*func*)
Add a function that will be called for uncaught exceptions, i.e. exceptions that will result in a 500 error.

This function should accept the exception instance as a single positional argument.

All handlers will be called in the order they are added.

Parameters **func** (*Callable*) –

init_app(*app*)
Register all the handler registries with a Flask application.

Parameters **app** (*flask.Flask*) –

2.7.2 Handler Registry

class flask_rebar.**HandlerRegistry**(*prefix=None*, *default_authenticators=None*, *default_headers_schema=None*, *default_mimetype=None*, *swagger_generator=None*, *spec_path='/swagger'*, *spec_ui_path='/swagger/ui'*)

Registry for request handlers.

This should typically be instantiated via a *Rebar* instance:

```
rebar = Rebar()
registry = rebar.create_handler_registry()
```

Although it can be instantiated independently and added to the registry:

```
rebar = Rebar()
registry = HandlerRegistry()
rebar.add_handler_registry(registry)
```

Parameters **prefix** (*str*) – URL prefix for all handlers registered with this registry instance.

:param Union(flask_rebar.authenticators.Authenticator, List(flask_rebar.authenticators.Authenticator), None)
default_authenticators: List of Authenticators to use for all handlers as a default.

Parameters

- **default_headers_schema** (*marshmallow.Schema*) – Schema to validate the headers on all requests as a default.
- **default_mimetype** (*str*) – Default response content-type if no content and not otherwise specified by the handler.
- **swagger_generator** (*flask_rebar.swagger_generation.swagger_generator.SwaggerGeneratorI*) – Object to generate a Swagger specification from this registry. This will be the Swagger generator that is used in the endpoints swagger and swagger UI that are added to the API. If left as None, a *SwaggerV2Generator* instance will be used.
- **spec_path** (*str*) – The Swagger specification as a JSON document will be hosted at this URL. If set as None, no swagger specification will be hosted.
- **spec_ui_path** (*str*) – The HTML Swagger UI will be hosted at this URL. If set as None, no Swagger UI will be hosted.

set_default_authenticator(*authenticator*)
Sets a handler authenticator to be used by default.

Parameters **flask_rebar.authenticators.Authenticator) authenticator** (*Union(None,)*) –

set_default_authenticators(*authenticators*)
Sets the handler authenticators to be used by default.

Parameters **authenticators** (*Union(List(flask_rebar.authenticators.Authenticator))*) –

set_default_headers_schema(*headers_schema*)
Sets the schema to be used by default to validate incoming headers.

Parameters **headers_schema** (*marshmallow.Schema*) –

clone()
Returns a new, shallow-copied instance of *HandlerRegistry*.

Return type *HandlerRegistry*

add_handler(*func*, *rule*, *method='GET'*, *endpoint=None*, *response_body_schema=None*, *query_string_schema=None*, *request_body_schema=None*, *headers_schema=<class 'flask_rebar.utils.defaults.USE_DEFAULT'>*, *authenticators=<class 'flask_rebar.utils.defaults.USE_DEFAULT'>*, *tags=None*, *mimetype=<class 'flask_rebar.utils.defaults.USE_DEFAULT'>*, *hidden=False*)
Registers a function as a request handler.

Parameters

- **func** – The Flask "view_func"
- **rule** (*str*) – The Flask "rule"
- **method** (*str*) – The HTTP method this handler accepts
- **endpoint** (*str*) –
- **marshmallow.Schema] response_body_schema** (*dict[int,*) – Dictionary mapping response codes to schemas to use to marshal the response. For now this assumes everything is JSON.
- **query_string_schema** (*marshmallow.Schema*) – Schema to use to deserialize query string arguments.
- **request_body_schema** (*marshmallow.Schema*) – Schema to use to deserialize the request body. For now this assumes everything is JSON.
- **headers_schema** (*Type[USE_DEFAULT]|None|marshmallow.Schema*) – Schema to use to grab and validate headers.
- **authenticators** (*Type[USE_DEFAULT]|None|List(Authenticator)|Authenticator*) – A list of authenticator objects to authenticate incoming requests. If left as USE_DEFAULT, the Rebar's default will be used. Set to None to make this an unauthenticated handler.
- **tags** (*Sequence[str]*) – Arbitrary strings to tag the handler with. These will translate to Swagger operation tags.
- **mimetype** (*Type[USE_DEFAULT]|None|str*) – Content-Type header to add to the response schema
- **hidden** (*bool*) – if hidden, documentation is not created for this request handler by default

handles(*rule*, *method='GET'*, *endpoint=None*, *response_body_schema=None*, *query_string_schema=None*, *request_body_schema=None*, *headers_schema=<class 'flask_rebar.utils.defaults.USE_DEFAULT'>*, *authenticators=<class 'flask_rebar.utils.defaults.USE_DEFAULT'>*, *tags=None*, *mimetype=<class 'flask_rebar.utils.defaults.USE_DEFAULT'>*, *hidden=False*)
Same arguments as *HandlerRegistry.add_handler()*, except this can be used as a decorator.

2.7.3 Authenticator Objects

class flask_rebar.authenticators.**Authenticator**
Abstract authenticator class. Custom authentication methods should extend this class.

authenticate()
Implementations of *Authenticator* should override this method.

This will be called before a request handler is called, and should raise an *flask_rebar.errors.HttpJsonError* is authentication fails.

Otherwise the return value is ignored.

Raises *flask_rebar.errors.Unauthorized*

class flask_rebar.**HeaderApiKeyAuthenticator**(*header*, *name='sharedSecret'*)
Authenticates based on a small set of shared secrets, passed via a header.

This allows multiple client applications to be registered with their own keys. This also allows multiple keys to be registered for a single client application.

Parameters

- **header** (*str*) – The header where clients where client applications must include their secret.
- **name** (*str*) – A name for this authenticator. This should be unique across authenticators.

register_key(*key*, *app_name='default'*)
Register a client application's shared secret.

Parameters

- **app_name** (*str*) – Name for the application. Since an application can have multiple shared secrets, this does not need to be unique.
- **key** (*str*) – The shared secret.

authenticate()
Implementations of `Authenticator` should override this method.

This will be called before a request handler is called, and should raise an *flask_rebar.errors.HttpJsonError* is authentication fails.

Otherwise the return value is ignored.

Raises *flask_rebar.errors.Unauthorized*

2.7.4 Swagger V2 Generation

class flask_rebar.**SwaggerV2Generator**(*host='localhost'*, *schemes=()*, *consumes=('application/json',)*, *produces=('application/json',)*, *version='1.0.0'*, *title='My API'*, *description=''*, *query_string_converter_registry=None*, *request_body_converter_registry=None*, *headers_converter_registry=None*, *response_converter_registry=None*, *tags=None*, *default_response_schema=<Error(many=False)>*, *authenticator_converter_registry=None*)
Generates a v2.0 Swagger specification from a Rebar object.

Not all things are retrievable from the Rebar object, so this guy also needs some additional information to complete the job.

Parameters

- **host** (*str*) – Host name or ip of the API. This is not that useful for generating a static specification that will be used across multiple hosts (i.e. PlanGrid folks, don't worry about this guy. We have to override it manually when initializing a client anyways.
- **schemes** (*Sequence[str]*) – "http", "https", "ws", or "wss". Defaults to empty. If left empty, the Swagger UI will infer the scheme from the document URL, ensuring that the "Try it out" buttons work.
- **consumes** (*Sequence[str]*) – Mime Types the API accepts
- **produces** (*Sequence[str]*) – Mime Types the API returns
- **query_string_converter_registry** (ConverterRegistry) –
- **request_body_converter_registry** (ConverterRegistry) –
- **headers_converter_registry** (ConverterRegistry) –
- **response_converter_registry** (ConverterRegistry) – ConverterRegistrys that will be used to convert Marshmallow schemas to the corresponding types of swagger objects. These default to the global registries.
- **tags** (*Sequence[Tag]*) – A list of tags used by the specification with additional metadata.

generate_swagger(*registry*, *host=None*)
Generate a swagger definition json object. :param registry: :param host: :return:

generate(*registry*, *host=None*, *schemes=None*, *consumes=None*, *produces=None*, *sort_keys=True*)
Generate a swagger specification from the provided *registry*

generate_swagger implements the SwaggerGeneratorI interface. But for backwards compatibility, we are keeping the similarly named *generate* signature.

Parameters

- **registry** (*flask_rebar.rebar.HandlerRegistry*) –
- **host** (*str*) – Overrides the initialized host
- **schemes** (*Sequence[str]*) – Overrides the initialized schemas
- **consumes** (*Sequence[str]*) – Overrides the initialized consumes
- **produces** (*Sequence[str]*) – Overrides the initialized produces
- **sort_keys** (*bool*) – Use OrderedDicts sorted by keys instead of dicts

Return type dict

class flask_rebar.**Tag**(*name*, *description=None*, *external_docs=None*)
Represents a Swagger "Tag Object"

Parameters

- **name** (*str*) – The name of the tag
- **description** (*str*) – A short description for the tag
- **external_docs** (ExternalDocumentation) – Additional external documentation for this tag

as_swagger()
Create a Swagger representation of this object

Return type dict

class flask_rebar.**ExternalDocumentation**(*url*, *description=None*)
Represents a Swagger "External Documentation Object"

Parameters

- **url** (*str*) – The URL for the target documentation. Value MUST be in the format of a URL
- **description** (*str*) – A short description of the target documentation

as_swagger()
Create a Swagger representation of this object

Return type dict

flask_rebar.swagger_generation.**sets_swagger_attr**(*attr*)
Decorates a *MarshmallowConverter* method, marking it as an JSONSchema attribute setter.

Example usage:

```
class Converter(MarshmallowConverter):
    MARSHMALLOW_TYPE = String()

    @sets_swagger_attr('type')
    def get_type(obj, context):
        return 'string'
```

This converter receives instances of *String* and converts it to a JSONSchema object that looks like *{'type': 'string'}*.

Parameters **attr** (*str*) – The attribute to set

class flask_rebar.swagger_generation.**ConverterRegistry**
Registry for MarshmallowConverters.

Schemas for responses, query strings, request bodies, etc. need to be converted differently. For example, response schemas as "dump"ed and request body schemas are "loaded". For another example, query strings don't support nesting.

This registry also allows for additional converters to be added for custom Marshmallow types.

register_type(*converter*)
Registers a converter.

Parameters **converter** (*MarshmallowConverter*) –

register_types(*converters*)
Registers multiple converters.

Parameters **converters** (*iterable[MarshmallowConverter]*) –

convert(*obj*, *openapi_version=2*)
Converts a Marshmallow object to a JSONSchema dictionary.

Parameters

- **obj** (*m.Schema|m.fields.Field|Validator*) – The Marshmallow object to be converted
- **openapi_version** (*int*) – major version of OpenAPI to convert obj for

Return type dict

2.7.5 Helpers

class flask_rebar.**ResponseSchema**(*only: Union[Sequence[str], Set[str], NoneType] = None, exclude: Union[Sequence[str], Set[str]] = (), many: bool = False, context: Union[Dict[KT, VT], NoneType] = None, load_only: Union[Sequence[str], Set[str]] = (), dump_only: Union[Sequence[str], Set[str]] = (), partial: Union[bool, Sequence[str], Set[str]] = False, unknown: Union[str, NoneType] = None*)

flask_rebar.**RequestSchema**
alias of marshmallow.schema.Schema

flask_rebar.**get_validated_args**()
Retrieve the result of validating/transforming an incoming request's query string with the *query_string_schema* a handler was registered with.

Return type dict

flask_rebar.**get_validated_body**()
Retrieve the result of validating/transforming an incoming request body with the *request_body_schema* a handler was registered with.

Return type dict

flask_rebar.**marshal**(*data*, *schema*)
Dumps an object with the given marshmallow.Schema.

Raises marshmallow.ValidationError if the given data fails validation of the schema.

flask_rebar.**response**(*data*, *status_code=200*, *headers=None*, *mimetype=None*)
Constructs a flask.jsonify response.

Parameters

- **data** (*dict*) – The JSON body of the response
- **status_code** (*int*) – HTTP status code to use in the response
- **headers** (*dict*) – Additional headers to attach to the response
- **mimetype** (*str*) – Default Content-Type response header

Return type flask.Response

2.7.6 Exceptions

class flask_rebar.errors.**HttpJsonError**(*msg=None*, *additional_data=None*)
Abstract base class for exceptions that will be cause and transformed into an appropriate HTTP error response with a JSON body.

These can be raised at any time during the handling of a request, and the Rebar extension will handling catching it and transforming it.

This class itself shouldn't be used. Instead, use one of the subclasses.

Parameters

- **msg** (*str*) – A human readable message to be included in the JSON error response
- **additional_data** (*dict*) – Dictionary of additional keys and values to be set in the JSON body. Note that these keys and values are added to the root object of the response, not nested under "additional_data".

class flask_rebar.errors.**BadRequest**(*msg=None*, *additional_data=None*)

http_status_code = 400

default_message = 'Bad Request'

class flask_rebar.errors.**Unauthorized**(*msg=None*, *additional_data=None*)

http_status_code = 401

default_message = 'Unauthorized'

class flask_rebar.errors.**PaymentRequired**(*msg=None*, *additional_data=None*)

http_status_code = 402

default_message = 'Payment Required'

class flask_rebar.errors.**Forbidden**(*msg=None*, *additional_data=None*)

http_status_code = 403

default_message = 'Forbidden'

class flask_rebar.errors.**NotFound**(*msg=None*, *additional_data=None*)

http_status_code = 404

default_message = 'Not Found'

class flask_rebar.errors.**MethodNotAllowed**(*msg=None*, *additional_data=None*)

http_status_code = 405

default_message = 'Method Not Allowed'

class flask_rebar.errors.**NotAcceptable**(*msg=None*, *additional_data=None*)

http_status_code = 406

default_message = 'Not Acceptable'

class flask_rebar.errors.**ProxyAuthenticationRequired**(*msg=None*, *additional_data=None*)

http_status_code = 407

default_message = 'Proxy Authentication Required'

class flask_rebar.errors.**RequestTimeout**(*msg=None*, *additional_data=None*)

http_status_code = 408

default_message = 'Request Timeout'

class flask_rebar.errors.**Conflict**(*msg=None*, *additional_data=None*)

http_status_code = 409

```
    default_message = 'Conflict'
class flask_rebar.errors.Gone(msg=None, additional_data=None)

    http_status_code = 410
    default_message = 'Gone'
class flask_rebar.errors.LengthRequired(msg=None, additional_data=None)

    http_status_code = 411
    default_message = 'Length Required'
class flask_rebar.errors.PreconditionFailed(msg=None, additional_data=None)

    http_status_code = 412
    default_message = 'Precondition Failed'
class flask_rebar.errors.RequestEntityTooLarge(msg=None, additional_data=None)

    http_status_code = 413
    default_message = 'Request Entity Too Large'
class flask_rebar.errors.RequestUriTooLong(msg=None, additional_data=None)

    http_status_code = 414
    default_message = 'Request URI Too Long'
class flask_rebar.errors.UnsupportedMediaType(msg=None, additional_data=None)

    http_status_code = 415
    default_message = 'Unsupported Media Type'
class flask_rebar.errors.RequestedRangeNotSatisfiable(msg=None, addi-
                                                      tional_data=None)
    http_status_code = 416
    default_message = 'Requested Range Not Satisfiable'
class flask_rebar.errors.ExpectationFailed(msg=None, additional_data=None)

    http_status_code = 417
    default_message = 'Expectation Failed'
class flask_rebar.errors.UnprocessableEntity(msg=None, additional_data=None)

    http_status_code = 422
    default_message = 'Unprocessable Entity'
class flask_rebar.errors.InternalError(msg=None, additional_data=None)
```

```
    http_status_code = 500
    default_message = 'Internal Server Error'
class flask_rebar.errors.NotImplemented(msg=None, additional_data=None)
    http_status_code = 501
    default_message = 'Not Implemented'
class flask_rebar.errors.BadGateway(msg=None, additional_data=None)
    http_status_code = 502
    default_message = 'Bad Gateway'
class flask_rebar.errors.ServiceUnavailable(msg=None, additional_data=None)
    http_status_code = 503
    default_message = 'Service Unavailable'
class flask_rebar.errors.GatewayTimeout(msg=None, additional_data=None)
    http_status_code = 504
    default_message = 'Gateway Timeout'
```

2.8 Tutorials

Our amazing community has some fantastic write ups on using Flask-Rebar. If you're new to Rebar and looking for some more concrete examples of how to use it check out some of these tutorials

2.8.1 Creating a Production Ready Python REST Backend with Flask-Rebar

- Part 1
- Part 2

Note: If you have a Flask-Rebar tutorial consider opening a PR to add it!

2.9 Recipes

2.9.1 Class Based Views

Some people prefer basing Flask view functions on classes rather than functions, and other REST frameworks for Flask base themselves on classes.

First, an opinion: people often prefer classes simply because they are used to them. If you're looking for classes because functions make you uncomfortable, I encourage you to take a moment to reconsider your feelings. Embracing functions, thread locals, and all of Flask's little quirks can feel oh so good.

With that, there are perfectly valid use cases for class based views, like creating abstract views that can be inherited and customized. This is the main intent of Flask's built-in pluggable views.

Here is a simple recipe for using Flask-Rebar with these pluggable views:

```
from flask import Flask
from flask import request
from flask.views import MethodView
from flask_rebar import Rebar


rebar = Rebar()
registry = rebar.create_handler_registry()


class AbstractResource(MethodView):
    def __init__(self, database):
        self.database = database

    def get_resource(self, id):
        raise NotImplemented

    def get(self, id):
        return self.get_resource(id)

    def put(self, id):
        resource = self.get_resource(id)
        resource.update(rebar.validated_body)
        return resource


class Todo(AbstractResource):
    def get_resource(self, id):
        return get_todo(database, id)


for method, request_body_schema in [
    ("get", None),
    ("put", UpdateTodoSchema()),
]:
    registry.add_handler(
        func=Todo.as_view(method + "_todo", database=database),
        rule="/todos/<id>",
        response_body_schema=TodoSchema(),  # for versions <= 1.7.0, use marshal_
↪schema
        method=method,
        request_body_schema=request_body_schema,
    )
```

This isn't a super slick, classed based interface for Flask-Rebar, but it *is* a way to use unadulterated Flask views to their full intent with minimal DRY violations.

2.9.2 Combining Security/Authentication

Authentication is hard, and complicated. Flask-Rebar supports custom Authenticator classes so that you can make your authentication as complicated as your heart desires.

Sometime though you want to combine security requirements. Maybe an endpoint should allow either an admin user

or a user with an "edit" permission, maybe you want to allow requests to use Auth0 or an Api Key, maybe you want to only authenticate if it's Sunday and Jupiter is in retrograde?

Here are some simple recipes for what Flask-Rebar currently supports:

Allow a user with either scope "A" OR scope "B"

```
from flask import g
from my_app import authenticator, registry
from my_app.scheme import EditStuffSchema, StuffSchema


# Allow a user with the "admin" scope OR the "edit:stuff" scope
@registry.handles(
        rule="/stuff/<uid:thing>/",
        method="POST",e
        request_body_schema=EditStuffSchema(),
        response_body_schema=StuffSchema(),
        authenticators=[authenticator.with_scope("admin"), authenticator.with_scope(
↪"edit:stuff")]
)
def edit_stuff(thing):
        update_stuff(thing, g.validated_body)
        return thing
```

Allow a request with either valid Auth0 OR an API-Key

```
from flask import g
from flask_rebar.authenticators import HeaderApiKeyAuthenticator
from flask_rebar_auth0 import get_authenticated_user
from my_app import authenticator, registry


# Allow Auth0 or API Key
@registry.handles(
    rule="/rate_limit/",
    method="GET",
    response_body_schema=RateLimitSchema(),
    authenticators=[authenticator, HeaderApiKeyAuthenticator("X-API-KEY")]
)
def get_limits():
        requester = g.authenticated_app_name or get_authenticated_user()
        rate_limit = get_limits_for_app_or_user(requester)
        return rate_limit
```

Allow a request with Auth0 AND an API-Key

Note: This currently requires some workarounds. Better support is planned.

```
from flask_rebar.authenticators import HeaderApiKeyAuthenticator
from flask_rebar_auth0 import Auth0Authenticator
from flask_rebar.swagger_generation.authenticator_to_swagger import (
        AuthenticatorConverter, authenticator_converter_registry
)
from my_app import app
```

(continues on next page)

(continued from previous page)

```
class CombinedAuthenticator(Auth0Authenticator, HeaderApiKeyAuthenticator):

        def __init__(self, app, header):
                Auth0Authenticator.__init__(self, app)
                HeaderApiKeyAuthenticator.__init__(self, header)

        def authenticate(self):
                Auth0Authenticator.authenticate(self)
                HeaderApiKeyAuthenticator.authenticate(self)


# You need to make sure that the converters already exist before trying to access them
# Create mock/stub authenticators that are used only for lookup
auth0_converter = authenticator_converter_registry._get_converter_for_
↪type(Auth0Authenticator(app))
header_api_converter = authenticator_converter_registry._get_converter_for_
↪type(HeaderApiKeyAuthenticator("header"))

class CombinedAuthenticatorConverter(AuthenticatorConverter):

        AUTHENTICATOR_TYPE = CombinedAuthenticator

        def get_security_schemes(self, obj, context):
                definition = dict()
                definition.update(auth0_converter.get_security_schemes(obj, context))
                definition.update(header_api_converter.get_security_schemes(obj,␣
↪context))
                return definition

        def get_security_requirements(self, obj, context):
                auth_requirement = auth0_converter.get_security_requirements(obj,␣
↪context)[0]
                header_requirement = header_api_converter.get_security_
↪requirements(obj, context)[0]
                combined_requirement = dict()
                combined_requirement.update(auth_requirement)
                combined_requirement.update(header_requirement)

                return [
                        combined_requirement
                ]


authenticator_converter_registry.register_type(CombinedAuthenticatorConverter())


@registry.handles(
    rule="/user/me/api_token",
    method="GET",
    authenticators=CombinedAuthenticator(app, "X-API-Key")
)
def check_token():
        return 200
```

2.9.3 Marshmallow Partial Schemas

Beginning with version 1.12, Flask-Rebar includes support for Marshmallow "partial" loading of schemas. This is particularly useful if you have a complicated schema with lots of required fields for creating an item (e.g., via a POST endpoint) and want to reuse the schema with some or all fields as optional for an update operation (e.g., via PATCH).

While you can accomplish this by simply adding a `partial` keyword argument when instantiating an existing schema, to avoid confusion in the generated OpenAPI model, we strongly recommend creating a derived schema class as illustrated in the following example:

```
class CreateTodoSchema(RequestSchema):
        complete = fields.Boolean(required=True)
        description = fields.String(required=True)
        created_by = fields.String(required=True)


class UpdateTodoSchema(CreateTodoSchema):
        def __init__(self, **kwargs):
                super_kwargs = dict(kwargs)
                partial_arg = super_kwargs.pop('partial', True)
                super(UpdateTodoSchema, self).__init__(partial=partial_arg, **super_
↪kwargs)
```

The preceeding example makes *all* fields from `CreateTodoSchema` optional in the derived `UpdateTodoSchema` class by injecting `partial=True` as a keyword argument. Marshmallow also supports specifying only some fields as "partial" so if, for example, you wanted to use this approach but make only the `description` and `created_by` fields optional, you could use something like:

```
class UpdateTodoSchema(CreateTodoSchema):
        def __init__(self, **kwargs):
                super_kwargs = dict(kwargs)
                partial_arg = super_kwargs.pop('partial', ['description', 'created_by
↪'])
                super(UpdateTodoSchema, self).__init__(partial=partial_arg, **super_
↪kwargs)
```

2.10 Contributing

We're excited about new contributors and want to make it easy for you to help improve this project. If you run into problems, please open a GitHub issue.

If you want to contribute a fix, or a minor change that doesn't change the API, go ahead and open a pull request; see details on pull requests below.

If you're interested in making a larger change, we recommend you to open an issue for discussion first. That way we can ensure that the change is within the scope of this project before you put a lot of effort into it.

2.10.1 Issues

We use GitHub issues to track public bugs and feature requests. Please ensure your description is clear and, if reporting a bug, include sufficient instructions to be able to reproduce the issue.

2.10.2 Our Commitment to You

Our commitment is to review new items promptly, within 3-5 business days as a general goal. Of course, this may vary with factors such as individual workloads, complexity of the issue or pull request, etc. Issues that have been reviewed will have a "triaged" label applied by the reviewer if they are to be kept open.

If you feel that an issue or pull request may have fallen through the cracks, tag an admin in a comment to bring it to our attention. (You can start with @RookieRick, and/or look up who else has recently merged PRs).

2.10.3 Process

Flask-Rebar is developed both internally within PlanGrid and via open-source contributions. To coordinate and avoid duplication of effort, we use two mechanisms:

1. We use the "triaged" label to mark issues as having been reviewed. Unless there are outstanding questions that need to be ironed out, you can assume that if an issue is marked as "triaged," we have generated an internal ticket, meaning someone will *eventually* address it. Timing of this will largely depend on whether there's a driving need within our own codebases that relies on Flask-Rebar.
2. Because internal ticketing is a black-box to our open source contributors, we will also make use of the "assignee" feature. If someone has picked up an internal ticket, there will be an assignee on the issue. If you see an open issue that doesn't have an assignee and that you would like to tackle please tag a maintainer in a comment requesting assignment, and/or open an early "WIP" pull request so we'll know the issue is already being worked, and can coordinate development efforts as needed.

2.10.4 Developing

We recommend using a virtual environment for development. Once within a virtual environment install the `flask_rebar` package:

```
pip install .[dev]
```

For *zsh* shell users, use

```
pip install '.[dev]'
```

We use *black* to format code and keep it all consistent within the repo. With that in mind, you'll also want to install the precommit hooks because your build will fail if your code isn't black:

```
pre-commit install
```

To run the test suite with the current version of Python/virtual environment, use pytest:

```
pytest
```

Flask-Rebar supports multiple versions of Python, Flask, and Marshmallow and uses Travis CI to run the test suite with different combinations of dependency versions. These tests are required before a PR is merged.

2.10.5 Pull Requests

1. Fork the repo and create your branch from `master`.
2. If you've added code that should be tested, add tests.
3. If you've changed APIs, update the documentation.

4. Make sure you commit message matches something like *(chg|fix|new): COMMIT_MSG* so *gitchangelog* can correctly generate the entry for your commit.

2.10.6 Meeting Notes

Links to notes from team meetings:

meeting_notes/roadmap_2020Jan29

2.10.7 Releasing to PyPI

We use GitHub Actions to automate releasing package versions to PyPI.

> **Warning:** These steps must be completed by an administrator. We generally do at least patch releases fairly frequently, but if you have a feature that urgently requires release, feel free to reach out and request one and we'll do our best to accommodate.

Flask-Rebar uses semantic versions. Once you know the appropriate version part to bump, use the `bumpversion` tool which will bump the package version, add a commit, and tag the commit appropriately. Note, it's not a bad idea to do a manual inspection and any cleanup you deem necessary after running `gitchangelog` to ensure it looks good before then committing a "@cosmetic" update.

Note: Before completing the following steps, you will need to temporarily change settings on GitHub under branch protection rules to NOT include administrators. This is required to allow you to push the changelog update.

```
git checkout master
git pull # just to play it safe and make sure you're up to date
bumpversion patch # or major or minor if applicable
gitchangelog
# STOP HERE:  inspect CHANGELOG.rst and clean up as needed before continuing
git commit -a -m "@cosmetic - changelog"
```

Then push the new commits and tags:

```
git push && git push --tags
```

Finally, while you're waiting for GitHub to pick up the tagged version, build it, and deploy it to PyPi, don't forget to reset branch protection settings (for normal operation, administrators should be subject to these restrictions to enforce PR code review requirements).

2.11 Version History

This Version History provides a high-level overview of changes in major versions. It is intended as a supplement for *Changelog*. In this document we highlight major changes, especially breaking changes. If you notice a breaking change that we neglected to note here, please let us know (or open a PR to add it to this doc)!

2.11.1 Version 2.0 (2021-07-26)

Errata

Version 2.0.0 included a couple of bugs related to the upgrade from Marshmallow 2 to 3. While the fix for one of those (removal of `DisallowExtraFieldsMixin`) might technically be considered a "breaking change" requiring a new major version, we deemed it acceptable to bend the rules of semantic versioning since that mixin **never actually functioned** in 2.0.0.

- Removed support for versions < 3.6 of Python
- Removed support for versions < 1.0 of Flask, and added support for Flask 2.x; we now support only Flask 1.x and 2.x.
- Removed support for versions < 3.0 of Marshmallow; we now support only Marshmallow 3.x
- (2.0.1) Removed `flask_rebar.validation.DisallowExtraFieldsMixin` - with Marshmallow 3, this is now default behavior and this mixin was broken in 2.0.0
- We now generate appropriate OpenAPI spec based on `Schema`'s `Meta` (ref https://marshmallow.readthedocs.io/en/stable/quickstart.html#handling-unknown-fields)
- Removed support for previously deprecated parameter names (https://github.com/plangrid/flask-rebar/pull/246/files)
- In methods that register handlers, `marshal_schema` is now `response_body_schema` and the former name is no longer supported
- `AuthenticatorConverterRegistry` no longer accepts a `converters` parameter when instantiating. Use `register_type` on an instance to add a converter
- Standardized registration of custom swagger authenticator converters (https://github.com/plangrid/flask-rebar/pull/216)
- Use of "converter functions" is no longer supported; use a class that derives from `AuthenticatorConverter` instead.
- Added "rebar-internal" error codes (https://github.com/plangrid/flask-rebar/pull/245)
- Can be used programmatically to differentiate between different "flavors" of errors (for example, the specific reason behind a `400 Bad Request`)
- This gets added to the JSON we return for errors
- Added support for marshmallow-objects >= 2.3, < 3.0 (https://github.com/plangrid/flask-rebar/pull/243)
- You can now use a marshmallow-objects `Model` instead of a marshmallow `Schema` when registering your endpoints.
- Add support for "hidden API" endpoints that are by default not exposed in generated OpenAPI spec (https://github.com/plangrid/flask-rebar/pull/191/files)

2.11.2 Version 1.0 (2018-03-04)

The first official release of Flask-Rebar!

2.12 Changelog

2.12.1 v2.1.0 (2021-10-19)

Fix

Add Python 3.10 support:

- Use Mapping from abstract classes (#259) [Daniel Wallace]
  - Mapping was removed from collections in 3.10 and is only available in collections.abc now.
  - update pytest for py3.10 support
  - add py3.10 to pr workflow
  - use setupactions v2

2.12.2 v2.0.2 (2021-08-09)

Fix

- Bumpversion bumped mock :([Rick Riensche]

2.12.3 v2.0.1 (2021-08-09)

Fix

- Remove now-pointless (and worse, broken) DisallowExtraFieldsMixin (#253) [Rick Riensche]
- Handle dump_only fields properly (#251) [Rick Riensche]

2.12.4 v2.0.0 (2021-07-27)

Changes

- Add internal error codes (#245) [Rick Riensche]
  - add error codes to ride alongside human-readable messages
  - Allow rename or suppression of flask_error_code in error responses
- Rename swagger_path to spec_path (#214) [Matthew Janes, Rick Riensche]
  - https://github.com/plangrid/flask-rebar/issues/98
  - Continuation of work started here: https://github.com/plangrid/flask-rebar/pull/209
  - Use deprecation util to make this more forgiving
- Remove previously deprecated things that have passed EOL (#246) [Rick Riensche]
  - remove deprecated 'marshal_schema' alias for 'response_body_schema'
  - remove deprecated (and already broken) converters parameter to `AuthenticatorConverterRegistry`
- Add support for marshmallow-objects (#243) [Rick Riensche]

- Add support for flask 2.0 (#239) [Eugene Kim]
- Store "rebar" in app.extensions on rebar.init_app (#230) [twosigmajab]
  - Include handler_registries so that it's possible to enumerate all the Rebar APIs a Flask app provides given only a reference to the app object.

Fix

- Update list of versions for automated tests + fix broken test (#238) [Eugene Kim]
- Remove deprecated method and update docs (#216) [Rick Riensche]
  - fix: remove deprecated authenticator converter methods and update docs
- Issue 90: Fix for self-referential nested fields (#226) [Pratik Pramanik, Rick Riensche]
- Fix bug that prevents returning ‘Flask.Response‘s. (#153) [twosigmajab]

Other

- Remove unused imports via Deepsource (#222) [Matthew Janes, deepsource-autofix[bot]]
- Remove old ref to TravisCI. [Rick Riensche]
- Drop Python 3.5 (fixes #220) (#221) [twosigmajab]
- Bump version to RC2. [Brock Haywood]
- Dropping Marshallow_v2 support (#195) [Ali Scott, Matthew Janes, Rick Riensche]
- Enabling hidden APIs in flask-rebar (#191) [Ali Scott]
- Rename create_handler_registry's "swagger_path" param (#209) [Ali Scott]
- Removing Actually from class names (#207) [Ali Scott]
- Fix trigger for auto-release on tag (#203) [Rick Riensche]
- Remove pyyaml pin since it is not needed anymore (#192) [Daniel Wallace]
  - We have dropped support for python3.4, which is why pyyaml was pinned.
- Doc: Fix code typos in recipes examples (#190) [Krismix1]
- Fix minor typos in swagger_generation.rst (#188) [Matthew Janes]

2.12.5 v1.12.2 (2020-08-04)

- Change trigger. [Rick Riensche]

  Auto-release on tag beginning with "v"

- Troubleshooting build-and-publish. [Rick Riensche]

  Add explicit mention of ref to checkout per https://github.com/actions/checkout/issues/20#issuecomment-524521113 (from the comment that follows this one though I'm not sure why this didn't work before if this DOES fix it.. :P)

- Authenticator_converter_registry is missing register_type. [Brock Haywood]
- Doc: Add meeting notes from Jan 29 Roadmap Call (#165) [Rick Riensche]

- Fix name and tweak “types” [Rick Riensche]

  More closely align with what is included if you use GitHub’s built-in package example (in attempt to make this actually trigger which it doesn’t seem to have done when I created 1.12.1)

2.12.6 v1.12.1 (2020-03-26)

- Fixes for oneof on _flatten (#182) [Francisco Castro]
- Add github actions to workflow (#177) [Daniel Wallace]
  - add workflows for github actions
  - removed python3.4 because it is end of lifed and doesn’t exist on

  github. + removed marshmallow 3.0.0rc5 since newer 3 versions have changed the api. Also, marshmallow 3.0 seems to have dropped support for python2. + remove .travis.yml
- !chg: Remove Flask-Testing dependency and drop Python 3.4 support. (#173) [Andrew Standley]
  - Remove Flask-Testing dependency.
  - Added JsonResponseMixin from Flask-Testing which we need as long as we continue to test flask<1.0
  - Pinned Werkzueg in travis. Should be able to drop if we drop flask<0.12 support.
  - Dropped support for python 3.4 in order to test support for Werkzeug 1.0.0
- FIX: Fix OpenApi v3 generation for nullable fields (#154) [mbierma]
  - Use ‘nullable’ to specify that the value may be null when generating v3 schema.
- Change host default to localhost (#157) [Daniel Wallace]

2.12.7 v1.12.0 (2020-01-08)

Changes

- Added support for marshmallow partial schema (#146) [Tuan Anh Hoang- Vu]
- Pin to PyYAML to avoid breaking change (Python 3.4) until we release our 2.0 and cut those old cords [Rick Riensche]

Other

- Doc: Added tutorial section for linking blogs and other external resources. (#143) [Andrew Standley]

2.12.8 v1.11.0 (2019-10-28)

- Improve swagger support for authenticators (#130) (BP-778. [Andrew Standley]
  - Added a get_open_api_version method to the swagger generator interface to help with refactoring the swagger tests so that we can use generators that have customer converters registered.
  - Updated jsonschema library for tests.
  - Added failing tests for swagger generation from Authenticators.
  - Added tests for the interface of AuthenticatorConverter to make sure I don’t accidentally change it.

  - Added authenticator to swagger conversion framework.
  - Updated the multiple_authenticators test to use the new auth converter framework.
  - Fixed eol_version for a deprecation message, and caught warnings on the legacy AuthenticatorConverter test.
  - Fix typos and imports.
  - Added documentation to AuthenticatorConverter. Also noted potential issue with conflicting scheme names in generators, going to push addressing that to later.
  - Added combined authentication examples to the recipes doc.

2.12.9 v1.10.2 (2019-09-19)

Fix

- Update authenticators to catch Forbidden exception (#133) [Marc-Éric]

2.12.10 v1.10.1 (2019-09-19)

Changes

- Tweaking build rules, updating docs, and prepping for bumpversion do- over. [Rick Riensche]

Fix

- Treat "description" key the same way as "explode" key for query and h... (#129) [Artem Revenko]

Other

- Accept bare class for schema arguments (#126) [Rick Riensche]
- Fix marshmallow test helpers so that they work will all unittest compatible frameworks and not just pytest. 'python setup.py test' works again. (#127) [Andrew Standley]

2.12.11 v1.10.0 (2019-09-11)

- BP-763: Add support for multiple authenticators (#122) [Andrew Standley]
  - Added the ability to specify a conversion function for deprecated params.
  - Added support for defining authentication with a list of Authenticators; None, a single Authenticator, and USE_DEFAULT(where applicable) are still valid values. The authenticator parameter becomes authenticators; authenticator is still usable until 3.0 via the deprecation wrappers. The default_authenticator parameter becomes default_authenticators; default_authenticator is still usable until 3.0 via the deprecation wrappers. This change affects PathDefinition, HandlerRegistry, Rebar, SwaggerGeneratorI, SwaggerV2Generator, and SwaggerV3Generator. Note: It's an open question how best to handle returning the errors when all authenticators fail. For now we are returning the first error with the assumption that the first authenticator is the 'preferred' one; this also preserves the previous behaviour.
  - Updated docs.
- [FEATURE] adding too many requests error (#120) [Fabian]

2.12.12 v1.9.1 (2019-08-20)

Fix

- 118 - pinned to an incompatible version of Marshmallow (3.0.0) [Rick Riensche]
  - Changes between 3.0.0rc5 and the actual release of 3.0.0 made our presumptive compatibility changes no longer sufficient
- Relax overly-sensitive test (#117) [Rick Riensche]
  - Deals with a subtle change in returned data on "Invalid input type" schema validation error between marshmallow 2.19 and 2.20. In return from Schema.load, "data" changed from empty dictionary to None, and we had an overzealous test that was expecting empty dictionary; whereas the value of "data" in this scenario appears to be undefined.

2.12.13 v1.9.0 (2019-07-24)

New

- Graceful deprecation rename of marshal_schema to response_body_schema (#101) [Rick Riensche]
  - chg: Refactor utilities into a separate utils package

Changes

- Move USE_DEFAULT to utils (#107) [retornam]
- Use extras_require for dev requirements (#106) [retornam]
- Allow /swagger/ui to resolve to swagger UI without redirect (#102) [Michael Bryant]

Fix

- Revert the red-herring sphinx conf change, add readthedocs yaml config. [Rick Riensche]
- Broke sphinx when we removed requirements.txt (#111) [Rick Riensche]

Other

- Run exception handlers on sys exit. [Brock Haywood]
- Doc: add code of conduct, based on https://www.contributor- covenant.org/ (#108) [Fabian]
- Fix(pypi): update pypi password (#105) [Sonny Van]
- Updated changelog. [Brock Haywood]

2.12.14 v1.8.1 (2019-06-14)

Changes

- Deprecation util cleaned up and expanded a bit. More forgiving of unexpected inputs. [Rick Riensche]

Fix

- Bug in v1.8.0 deprecation util - deepcopy inadvertently replacing things like default_authenticator

2.12.15 v1.8.0 (2019-06-12)

New

- Graceful deprecation rename of marshal_schema to response_body_schema (#101) [Rick Riensche]
- Refactor utilities into a separate utils package including new deprecation utility

Changes

- Allow /swagger/ui to resolve to swagger UI without redirect (#102) [Michael Bryant]

2.12.16 v1.7.0 (2019-06-05)

- Fixes a bug where http 400s are returned as http 500s (#99) [Brock Haywood]

  this is for a case where a werkzeug badrequest exception is raised before the rebar handlers get invoked. this was causing the default rebar exception handler to run, thus returning a 500
- Updating Contributing page to reflect revised issue review process (#95) [Rick Riensche]
- Fix #96 - Flask no longer treats redirects as errors (#97) [Rick Riensche]

2.12.17 v1.6.3 (2019-05-10)

- Respect user-provided content type in all cases. [Joe Bryan]
- Add default_mimetype to registry. [Joe Bryan]
- Return empty object not empty string, if an empty non-null object response is specified. [Joe Bryan]

2.12.18 v1.6.2 (2019-05-08)

Fix

- DELETE requests should return specified Content-Type (#85) [Joe Bryan]

2.12.19 v1.6.1 (2019-05-03)

Fix

- Quick rehacktor to unbreak import statements like "from flask_rebar.swagger_generation.swagger_generator import SwaggerV2Generator" (#86) [Rick Riensche]

2.12.20 v1.6.0 (2019-05-02)

- Add OpenAPI 3 Support (#80) [barak]
- Sort required array (#81) [Brandon Weng]
- Doc: List Flask-Rebar-Auth0 as an extension (#76) [barak]
- Minor changelog manual cleanup. [Rick Riensche]
- Doc: update changelog. [Rick Riensche]

2.12.21 v1.5.1 (2019-03-22)

Fix

- Werkzeug 0.14->0.15 introduced some breaking changes in redirects (#73) [Rick Riensche]

2.12.22 v1.5.0 (2019-03-22)

Changes

- Enforce black on PR's (#68) [Julius Alexander IV, Fabian]
- Updated todo example to show tag usage (#59) [Fabian]

Fix

- Do not rethrow redirect errors (#65) [Julius Alexander IV]

Other

- Doc: one more minor tweak to our "SLA" (#71) [Rick Riensche]
- Doc: minor doc cleanup, addition of "SLA-esque" statement to Contributing (#70) [Rick Riensche]
- Fix minor formatting issue in docs. [Rick Riensche]
- Add recipe for class based views (#63) [barak]
- Adds a codeowners file (#66) [Brock Haywood]
- Update changelog. [Julius Alexander]

2.12.23 v1.4.1 (2019-02-19)

Fix

- Change schemes=() default so Swagger UI infers scheme from document URL (#61) [twosigmajab]

Other

- Update changelog. [Julius Alexander]

2.12.24 v1.4.0 (2019-01-31)

New

- Add gitchangelog (#56) [Julius Alexander IV]

Other

- Support for tags (#55) [barak]
- Add 'https' to default schemes (#53) [twosigmajab]

2.12.25 v1.3.0 (2018-12-04)

- Prepare for Marshmallow version 3 (#43) [barak]

2.12.26 v1.2.0 (2018-11-29)

- Dump_only=True -> readOnly (#42) [twosigmajab]

  Fixes #39.
- Fix "passowrd" typo in swagger_words (#40) [twosigmajab]
- Rm superfluous logic in swagger_ui.blueprint.show (#38) [twosigmajab]
- Respect many=True in swagger_generator. (#45) [twosigmajab]

  Fixes #41.

2.12.27 v1.1.0 (2018-11-13)

- Allow disabling OrderedDicts in generated swagger (#32) [twosigmajab]
- Improve marshal_schema and response header handling (#28) [barak]
- Update release docs. (#31) [Julius Alexander IV]
- Merge pull request #34 from plangrid/required-field-enforce- validation. [Joe Bryan]

  Enforce field validators when using ActuallyRequireOnDumpMixin
- Merge branch 'master' into required-field-enforce-validation. [Joe Bryan]
- Merge pull request #35 from plangrid/sort-query-params. [Joe Bryan]

  Sort query params for consistent output
- Sort query params for consistent output. [Joe Bryan]
- Use marshmallow built in validation. [Joe Bryan]
- Enforce field validators when using ActuallyRequireOnDumpMixin. [Joe Bryan]

2.12.28 v1.0.8 (2018-10-30)

- Use built in library for version comparison (#29) [barak]

2.12.29 v1.0.7 (2018-10-29)

- Handle RequestRedirect errors properly (#25) [barak]
- Fix docs about specifying custom swagger generator (#23) [barak]

2.12.30 v1.0.6 (2018-10-11)

- Changed default 'produces' of swagger generation to 'application/json' (#19) [barak]

2.12.31 v1.0.4 (2018-04-05)

- Feat(type): added path. [Anthony Martinet]

2.12.32 v1.0.3 (2018-03-27)

- Re-raise uncaught errors in debug mode (#14) [barak]
- Add Swagger UI data files to MANIFEST.in. [barakalon]

2.12.33 v1.0.2 (2018-03-07)

- Get Travis to deploy again. [barakalon]

2.12.34 v1.0.1 (2018-03-07)

- Use find_packages in setup.py. [barakalon]
- Fix README example. [barakalon]
- Break pypi release into its own job. [barakalon]
- Prevent double travis builds for PRs. [barakalon]
- Clarify PyPI release instructions. [barakalon]

2.12.35 v1.0.0 (2018-03-04)

- Rename marshal_schemas to marshal_schema. [barakalon]
- Add badge and some documentation for releasing. [barakalon]

2.12.36 v0.1.0 (2018-03-03)

- Add deployment to PyPI. [barakalon]
- Remove client_test since its not working for python2.7 and needs more testing/documentation. [barakalon]
- Adding travis yaml file. [barakalon]
- Move why flask-rebar documetnation to sphinx only. [barakalon]
- Adding ReadTheDocs. [barakalon]

- Add lots of documentation. [barakalon]
- Split registry out and add prefixing. [barakalon]
- Remove flask_swagger_ui dependency. [barakalon]
- Example app and pytest. [barakalon]
- Refactoring to a smaller package. [barakalon]
- Moving tests directories around. [barakalon]
- Move authenticators to package root. [barakalon]
- Rename framing to swagger_generation. [barakalon]
- Move registry to package root. [barakalon]
- Rename extension to registry. [barakalon]
- Packaging boilerplate. [barakalon]
- Some packaging updates. [barakalon]
- Flask_toolbox -> flask_rebar. [barakalon]
- Get rid of plangrid namespace. [barakalon]
- Cleanup some files. [barakalon]
- Sort generated swagger alphabetically (#46) [colinhostetter]
- Don't ship tests or examples in installed package. [Tom Lippman]
- Add framer env variables to readme. [barakalon]
- Support configuring Framer auth without app. [Nathan Yergler]
- Fixes UUID and ObjectId fields: - honor the allow_none keyword - but don't pass validation for an empty string. [Tom Lippman]

  Also adds a function to dynamically subclass any Field or Schema to add checking validation logic on serialization.
- Update bugsnag to 3.4.0. [Nathan Yergler]
- Add PaginatedListOf and SkipLimitSchema helpers (#41) [colinhostetter]
- Add configuration for bumpversion utility. [Nathan Yergler]
- Add utility for testing with swagger generated client libraries. [Nathan Yergler]
- Fix converter handling in swagger generator. [colinhostetter]
- Bump version to 2.3.0. [barakalon]
- Allow for paginated data. [barakalon]
- Bump version to 2.2.0. [barakalon]
- Add default headers to bootstrapping. [barakalon]
- Fix up the README a little bit. [barakalon]
- Bump version to 2.1.1. [barakalon]
- Fix up some of the package interface. [barakalon]
- Bump major version. [barakalon]
- Some more marshmallow to jsonschema fields. [barakalon]

- Default headers. [barakalon]
- Example app. [barakalon]
- Refactor tests a bit. [barakalon]
- CACA-468 Fix DisallowExtraFields erroring for bad input. [Julius Alexander]
- Bump version 1.7.1. [barak-plangrid]
- Gracefully handle missing marshmallow validators in swagger generator. [barak-plangrid]
- Publicize marshmallow formatting. [barak-plangrid]
- Move swagger ui to flask toolbox. [barak-plangrid]
- Add back some commits lost in rebase. [barak-plangrid]
- Explicitly import bugsnag.flask. [Nathan Yergler]
- Allow apps to pass in their swagger generator. [Nathan Yergler]
- Allow specification of API description. [Nathan Yergler]
- Swagger endpoint. [barak-plangrid]
- Add check the the swagger we're producing is valid. [barak-plangrid]
- Added default authenticators. [barak-plangrid]
- Dont marsh my mellow. [barak-plangrid]
- Fix the error raised by UUIDStringConverter. [Colin Hostetter]
- Add custom UUID string converter. [Colin Hostetter]
- Fix comma splice in healthcheck response message (#20) [dblackdblack]
- Start recording userId in new relic. [barak-plangrid]
- Test improvements. [Colin Hostetter]
- Fix null values in ObjectId/UUID marshmallow fields. [Colin Hostetter]
- Fix UUID field type to work with None values. [Colin Hostetter]
- Use route:method for new relic transaction name. [Colin Hostetter]
- Correctly set New Relic transaction name in restful adapter. [Colin Hostetter]
- Support multiple routes in RestfulApiAdapter.add_resource. [Colin Hostetter]
- Bump version to 1.2.0. [barak-plangrid]
- CACA-84 support capi in flask toolbox. [barak-plangrid]
- CACA-97 add scope helper functions (#13) [barak]
- Expand abbreviation. [Colin Hostetter]
- Add get_user_id_from_header_or_400 function to toolbox. [Colin Hostetter]
- Add docstring to QueryParamList. [Colin Hostetter]
- Add a Marshmallow list type for repeated query params. [Colin Hostetter]
- Version bump. [Colin Hostetter]
- Break response messages into separate file. [Colin Hostetter]
- Use keyword args for building response. [Colin Hostetter]

- Fix non-tuple returns in adapter. [Colin Hostetter]
- Use toolbox response func instead of building our own responses. [Colin Hostetter]
- Throw an error if an HTTP method is declared without a matching class method. [Colin Hostetter]
- Style changes. [Colin Hostetter]
- Use new style classes. [Colin Hostetter]
- Fix tests to work in CI. [Colin Hostetter]
- Another version bump. [Colin Hostetter]
- Add adapter to replace flask-restful Api class. [Colin Hostetter]
- Add support for exception logging via New Relic. [Colin Hostetter]
- Version bump. [Colin Hostetter]
- Only configure Bugsnag when a BUGSNAG_API_KEY is provided. [Colin Hostetter]

  This helps prevent spam when running automated tests, developing locally, etc.
- Add support for HTTP 422 error. [Colin Hostetter]
- Setup Jenkins (#5) [barak]
  - setup Jenkins
  - add dockerfile
  - fixup
- Increment version. [Colin Hostetter]
- Consolidate JSON loading error handling. [Colin Hostetter]
- Correctly format errors raised by request.get_json() [Colin Hostetter]
- Bump version to 1.0.0. [barak-plangrid]
- Namespace this package (#2) [barak]
  - Namespace the package
  - fixup
- Notify on 500. (#1) [Julius Alexander IV]
- Fixup. [barak-plangrid]
- Initial commit. [barak-plangrid]

Index

T

U

V