pyramid_swagger Documentation

Release 0.1.0

Scott Triglia

Jul 06, 2018

Contents

This project offers convenient tools for using Swagger to define and validate your interfaces in a Pyramid webapp.

You must supply a working Pyramid application, and a Swagger schema describing your application's interface. In return, pyramid_swagger will provide:

- Request and response validation
- Swagger spec validation
- Automatically serving the swagger schema to interested clients (e.g. Swagger UI)

pyramid_swagger works for both the 1.2 and 2.0 Swagger specifications, although users are strongly encouraged to use 2.0 going forward.

Contents:

CHAPTER 1

What is Swagger?

Basic working knowledge of Swagger is a prerequisite for having pyramid_swagger make sense as a library.

Swagger <http://www.swagger.io> is a specification format for describing HTTP services, with a particular focus on RESTful APIs. The schema you write will describe your API comprehensively.

The benefit of going through the work of writing a Swagger schema for API is you then get access to a great number of tools which work off this spec. The Swagger website has an entire page devoted to *community tools which consume this schema <http://swagger.io/open-source-integrations/>*. In fact, you'll notice that pyramid_swagger is listed as one of these.

In short, your Swagger schema simply describes your API. For a more in-depth introduction, try *the official Swagger introduction article<http://swagger.io/getting-started-with-swagger-i-what-is-swagger/>*.

CHAPTER 2

Quickstart

So let's get your pyramid app up and running!

The core steps to use pyramid_swagger are quite simple:

1. Create a Swagger Schema for your service's endpoints
2. Add pyramid_swagger to your Pyramid application

2.1 Creating your first Swagger Schema

Creating your initial Swagger Schema can be intimidating but don't fear, it's not nearly as much work as it might initially appear.

To create your first Swagger Schema, I encourage you to take a look at Swagger's official PetStore example. You can even see the raw JSON for the Swagger Schema. You'll notice that Swagger has a lot of details, but the core part of building a schema is documenting each endpoint's inputs and outputs.

For your intial attempt, documenting an endpoint can be simplified to some basic components:

1. Documenting the core URI (e.g. /foo/bar)
2. Documenting request parameters (in the path, in the query arguments, and in the query body)
3. Documenting the response

There are many other pieces of your REST interface that Swagger can describe, but these are the core components. The PetStore example has some good examples of all of these various types, so it can be a useful reference as you get used to the syntax.

For any questions about various details of documenting your interface with Swagger, you can consult the official Swagger Spec, although you may find it somewhat difficult to parse for use as anything but a reference.

You may find that the process of writing your API down in the Swagger format is surprisingly hard. . . this is good! It probably suggests that your API is not terribly well understood or maybe even underspecified right now. Anecdotally, users commonly report that writing their first Swagger api-docs has the unintended side effect of forcing them to reconsider exactly how their service should be interacting with the outside world – a useful exercise!

2.2 Where to put your Swagger Schema

Great, so we have one large JSON file containing our API declaration for all endpoints our service supports. What now?

Now place the Swagger Schema in `api_docs/swagger.json`. The path has no relation to the paths described in your API declaration, it is only used internally to help Swagger discover your schemas.

2.3 Add pyramid_swagger to your webapp

Last but not least, we need to turn on the pyramid_swagger library within your application. This is quite easy by default, either by augmenting your PasteDeploy .ini file, or by adding a line to your webapp method.

We'll show you the .ini method here, but you can read how to imperatively add the library to your app (and much more) in the *configuration page* of these docs. For those using the .ini file, simply add the following line under your `[app:main]` section:

```
[app:main]
pyramid.includes = pyramid_swagger
```

With that, when your app starts you will get the benefit of:

- 4xx errors for requests not matching your schema
- 5xx errors for responses not matching your schema
- Automatic validation for correctness of your Swagger Schema at application startup
- Automatic serving of your Swagger Schema from the /swagger.json endpoint

2.4 Update the routes

For each of the routes declared in your swagger.json, you need to add the route to the Pyramid dispatch using traditional methods. For example, in your __init__.py:

```
def main(global_config, **settings):
    """ This function returns a Pyramid WSGI application.
    """
    config = Configurator(settings=settings)
    config.include('pyramid_chameleon')
    config.add_static_view('static', 'static', cache_max_age=3600)
    config.add_route('api.things.get', '/api/things', request_method='GET')
    #
    # Additional routes go here
    #
    config.scan()
    return config.make_wsgi_app()
```

2.5 Accessing request data

Now that `pyramid_swagger` is enabled you can create a view. All the values that are specified in the Swagger Schema for an endpoint are available from a single `dict` on the request `request.swagger_data`. These values are casted to the type specified by the Swagger Schema.

Example:

```
from pyramid.view import view_config

@view_config(route_name='api.things.get')
def get_things(request):
    # Returns thing_id as an int (assuming the swagger type is integer)
    thing_id = request.swagger_data['thing_id']
    ...
    return {...}
```

The raw values (not-casted to any type) are still available from their usual place on the request (*matchdict*, *GET*, *POST*, *json()*, etc)

If you have `pyramid_swagger.use_models` set to true, you can interact with models defined in `#/definitions` as Python classes instead of dicts.

```
{
  "swagger": "2.0",
  "definitions": {
    "User": {
      "type": "object",
      "properties": {
        "first_name": {
          "type": "string"
        },
        "last_name": {
          "type": "string"
        }
      }
    }
  }
}
```

```
@view_config(route_name='add.user')
def add_user(request):
    user = request.swagger_data['user']
    assert isinstance(user, bravado_core.models.User)
    first_name = user.first_name
    ...
```

Otherwise, models are represented as dicts.

```
@view_config(route_name='add.user')
def add_user(request):
    user = request.swagger_data['user']
    assert isinstance(user, dict)
    first_name = user['first_name']
    ...
```

Note: Values in `request.swagger_data` are only available if `pyramid_swagger.enable_request_validation` is enabled.

2.6 Accessing Swagger Operation

During the implementation of an endpoint you could eventually have need of accessing the Swagger Specs that defined that specific view. `pyramid_swagger` will inject in the request object a new property (that will be evaluated only if accessed) called `operation`.

`request.operation` will be set to `None` for Swagger 1.2 defined endpoints, while it will be an Operation object if the endpoint is defined by Swagger 2.0 specs.

2.7 pyramid_swagger renderer

Using `pyramid_swagger` you will get automatic conversions of the input JSON objects to easy to handle python objects. An example could be a swagger object string property using the `date` format , the library will take care of converting the ISO 8601 date representation to an easy to handle python `datetime.date` object.

While defining the `pyramid` view that will handle the endpoint you have to make sure that the chosen renderer will be able to properly render your response. In the case of an endpoint that returns *objects* that requires a special handling (like `datetime.date`) the developer is *forced* to:

- manually convert the python object to an object that could be handled by the renderer
- add an adapter for instructing pyramid to handle your object
- define a custom renderer that is able to properly serialize the object

`pyramid_swagger` provides:

- a new renderer, called `pyramid_swagger`
- a new renderer renderer factory, called `pyramid_swagger.renderer.PyramidSwaggerRendererFactory`

2.7.1 How pyramid_swagger renderer works

The new `pyramid_swagger` renderer is a wrapper around the default `pyramid.renderers.JSON` renderer.

`pyramid_swagger` will receive, from your pyramid view, the object that has to be rendered, perform the marshaling operations and then call the default JSON renderer.

Note: The usage of this renderer allows to get full support of custom formats

Let's assume that your view returns `{'date': datetime.date.today()}` and that your response spec is similar to

```
{
    "200": {
        "description": "HTTP/200",
        "schema": {
            "properties": {
                "date": {
                    "type": "string",
                    "format": "date"
                }
            }
        }
```

(continues on next page)

(continued from previous page)

```
    }
}
```

If your view is configured to use `json` renderer then your endpoint will surprisingly return HTTP/500 errors. The errors are caused by the fact that `pyramid.renderers.JSON` is not aware on how to convert a `datetime.date` object.

If your view is configured to use `pyramid_swagger` renderer then your endpoint will provide HTTP/200 responses similar to `{"date": "2017-09-16"}`.

This is possible because the marsharling of the view return value converts the `datetime.date` object to its ISO 8601 string representation that could be handled by the default JSON renderer.

Note: The marshaling operation will be performed according to the specific response schema defined for the particular endpoint. It means that if your response doesn't specify a field it will be transparently passed to the wrapped renderer.

2.7.2 How PyramidSwaggerRendererFactory works

`PyramidSwaggerRendererFactory` allows you to create a custom renderer that operates on the marshaled result from your view.

The defined renderer will operate on the marshaled, according to the Swagger Specification, response.

Example of definition of a custom renderer

```
class MyPersonalRendererFactory(object):
    def __init__(self, info):
        # Initialize your factory (refer to standard documentation for more␣
↪information)
        pass

    def __call__(self, value, system):
        # ``value`` contain the marshaled representation of the object returned by␣
↪your view.
        # If your view is returning a ``datetime.date`` object for a field with date␣
↪format
        # you can assume that the field has already been converted to its ISO 8601␣
↪representation

        # perform your personal rendering operations
        # you can assume that value is a marshaled response, so already JSON␣
↪serializable object
        return rendered_value
```

Once you have defined your own renderer you have to wrap the new renderer in `PyramidSwaggerRendererFactory` and register it to the pyramid framework as described by Adding and Changing Renderers pyramid documentation.

```
config.add_renderer(name='custom_renderer',␣
↪factory=PyramidSwaggerRendererFactory(MyPersonalRendererFactory))
```

CHAPTER 3

Changelog

3.1 2.6.2 (2018-07-02)

- Fix bug that prevents library usage on Windows Platform (see #234)

3.2 2.6.1 (2018-05-24)

- Fix operation extraction in *PyramidSwaggerRendererFactory* in case of Swagger1.2 endpoint (see #230)
- Fix request body extraction if body is not set. (see #231)

3.3 2.6.0 (2017-11-14)

- Support setting bravado-core config values by prefixing them with `bravado_core.` in the pyramid_swagger config (see #221)
- Support raw_bytes response attribute, required for msgpack wire format support in outgoing responses (see #222)

3.4 2.5.0 (2017-10-26)

- Support *include_missing_properties* bravado-core flag in pyramid configuration
- Outsource flattening logic to bravado-core library.
- Expose bravado-core `operation` in request object
- Add `pyramid_renderer` and `PyramidSwaggerRendererFactory`

3.5 2.4.1 (2017-06-14)

- Bugfix: add a quick fix to prevent resolve_refs from making empty json keys on external refs (see #206)

3.6 2.4.0 (2017-05-30)

- Bugfix: prevent resolve_refs from resolution failures when flattening specs with recursive $refs (see #204)
- Allow serving of api_docs from paths besides /api_docs (see #187)
- Support virtual hosting via SCRIPT_NAME (see #201 and https://www.python.org/dev/peps/pep-0333/)

3.7 2.3.2 (2017-04-10)

- Fix reading configuration values from INI files (see #182, #200)

3.8 2.3.1 (2017-03-27)

- Fix validation context for swagger 2.0 requests
- Added docs for validation context
- Preserved original exception when reraising for validation context exceptions
- Remove support for Python 2.6, newer Pyramid versions don't support it either
- Fix issue with missing content type when using webob >= 1.7 (see #185)

3.9 2.3.0 (2016-09-27)

- Fix installation with Python 3 on systems using a POSIX/ASCII locale.

3.10 2.3.0-rc3 (2016-06-28)

- Adds `dereference_served_schema` config flag to force served spec to be a single file. Useful for avoiding mixed-spec inconsistencies when running multiple versions of your service simultaneously.

3.11 2.3.0-rc2 (2016-05-09)

- Add ability for a single spec to serve YAML or JSON to clients
- Support multi-file local specs, serving them over multiple HTTP endpoints
- Improve Swagger validation messages when Pyramid cannot find your route (see #163)
- Bugfix: responses with headers in the spec no longer break request validation (see #159)

3.12 2.3.0-rc1 (2016-03-21)

- Support for YAML spec files
- Bugfix: remove extraneous x-scope in digested spec (see https://github.com/Yelp/bravado-core/issues/78)

3.13 2.2.3 (2016-02-09)

- Restore testing of py3x versions
- Support pyramid 1.6 and beyond.
- Support specification of routes using route_prefix

3.14 2.2.2 (2015-10-12)

- Upgrade to bravado-core 3.0.0, which includes a change in the way user-defined formats are registered. See the Bravado 3.0.0 changelog entry for more detail.

3.15 2.2.1 (2015-08-20)

- No longer attempts to validate error responses, which typically don't follow the same format as successful responses. (Closes: #121)

3.16 2.2.0 (2015-08-19)

- Added `prefer_20_routes` configuration option to ease incremental migrations from v1.2 to v2.0. (See *Incremental Migration*)

3.17 2.1.0 (2015-08-14)

- Added `user_formats` configuration option to provide user-defined formats which can be used for validations and conversions to wire-python-wire formats. (See *user_formats (Swagger 2.0 only)*)
- Added support for relative cross-refs in Swagger v2.0 specs.

3.18 2.0.0 (2015-06-25)

- Added `use_models` configuration option for Swagger 2.0 backwards compatibility with existing pyramid views

3.19 2.0.0-rc2 (2015-05-26)

- Upgraded bravado-core to 1.0.0-rc1 so basePath is used when matching a request to an operation
- Updates for refactored SwaggerError exception hierarchy in bravado-core
- Fixed file uploads that use Content-Type: multipart/form-data

3.20 2.0.0-rc1 (2015-05-13)

Backwards Incompatible

- Support for Swagger 2.0 - See Migrating to Swagger 2.0

3.21 1.5.0 (2015-05-12)

- Now using swagger_spec_validator package for spec validation. Should be far more robust than the previous implementation.

3.22 1.5.0-rc2 (2015-04-1)

- Form-encoded bodies are now validated correctly.
- Fixed bug in *required* swagger attribute handling.

3.23 1.5.0-rc1 (2015-03-30)

- Added `enable_api_docs_views` configuration option so /api-docs auto-registration can be disabled in situations where users want to serve the Swagger spec in a nonstandard way.
- Added `exclude_routes` configuration option. Allows a blacklist of Pyramid routes which will be ignored for validation purposes.
- Added `generate_resource_listing` configuration option to allow pyramid_swagger to generate the `apis` section of the resource listing.
- Bug fix for issues relating to `void` responses (See Issue 79)
- Added support for header validation.
- Make casted values from the request available through `request.swagger_data`

3.24 1.4.0 (2015-01-27)

- Added `validation_context_path` setting which allows the user to specify a path to a contextmanager to custom handle request/response validation exceptions.

3.25 1.3.0 (2014-12-02)

- Now throws RequestValidationError and ResponseValidationError instead of HTTPClientError and HTTPInternalServerError respectively. The new errors subclass the old ones for backwards compatibility.

3.26 1.2.0 (2014-10-21)

- Added `enable_request_validation` setting which toggles whether request content is validated.
- Added `enable_path_validation` setting which toggles whether HTTP calls to endpoints will 400 if the URL is not described in the Swagger schema. If this flag is disabled and the path is not found, no validation of any kind is performed by pyramid-swagger.
- Added `exclude_paths` setting which duplicates the functionality of *skip_validation*. *skip_validation* is deprecated and scheduled for removal in the 2.0.0 release.
- Adds LICENSE file
- Fixes misuse of webtest which could cause `make test` to pass while functionality was broken.

3.27 1.1.1 (2014-08-26)

- Fixes bug where response bodies were not validated correctly unless they were a model or primitive type.
- Fixes bug where POST bodies could be mis-parsed as query arguments.
- Better backwards compatibility warnings in this changelog!

3.28 1.1.0 (2014-07-14)

- Swagger schema directory defaults to `api_docs/` rather than being a required configuration line.
- If the resource listing or API declarations are not at the filepaths expected, readable errors are raised.
- This changelog is now a part of the build documentation and backfilled to the initial package version.

3.29 1.0.0 (2014-07-08)

Backwards Incompatible

- Initial fully functional release.
- Your service now must supply both a resource listing and all accompanying api declarations.
- Swagger schemas are automatically served out of `/api-docs` by including the library.
- The api declaration basepath returned by hitting `/api-docs/foo` is guaranteed to be `Pyramid.request.application_url`.
- Void return types are now checked.

3.30 0.5.0 (2014-07-08)

- Added configurable list of regular expressions to not validate requests/responses against.
- Vastly improved documentation! Includes a quickstart for those new to the library.
- Adds coverage and code health badges to README

3.31 0.4.0 (2014-06-20)

- Request validation now works with path arguments.
- True acceptance testing implemented for all known features. Much improved coverage.

3.32 0.4.0 (2014-06-20)

- True acceptance testing implemented for all known features. Much improved coverage.

3.33 0.3.2 (2014-06-16)

- HEAD is now an allowed HTTP method

3.34 0.3.1 (2014-06-16)

- Swagger spec is now validated on startup
- Fixes bug where multiple methods with the same URL were not resolved properly
- Fixes bug with validating non-string args in paths and query args
- Fixes bug with referencing models from POST bodies

3.35 0.3.0 (2014-05-29)

- Response validation can be disabled via configuration
- Supports Python 3.3 and 3.4!

3.36 0.2.2 (2014-05-28)

- Adds readthedocs links, travis badge to README
- Requests missing bodies return 400 instead of causing tracebacks

3.37 0.2.1 (2014-05-15)

- Requests to non-existant endpoints now return 400 errors

3.38 0.1.1 (2014-05-13)

- Build docs now live at `docs/build/html`

3.39 0.1.0 (2014-05-12)

- Initial version. Supports very basic validation of incoming requests.

CHAPTER 4

Configuring pyramid_swagger

The pyramid_swagger library is intended to require very little configuration to get up and running.

A few relevant settings for your Pyramid .ini file (and their default settings):

```
[app:main]
# Add the pyramid_swagger validation tween to your app (required)
pyramid.includes = pyramid_swagger

# `api_docs.json` for Swagger 1.2 and/or `swagger.json` for Swagger 2.0
# directory location.
# Default: api_docs/
pyramid_swagger.schema_directory = schemas/live/here

# For Swagger 2.0, defines the relative file path (from
# `schema_directory`) to get to the base swagger spec.
# Supports JSON or YAML.
#
# Default: swagger.json
pyramid_swagger.schema_file = swagger.json

# Versions of Swagger to support. When both Swagger 1.2 and 2.0 are
# supported, it is required for both schemas to define identical APIs.
# In this dual-support mode, requests are validated against the Swagger
# 2.0 schema only.
# Default: 2.0
# Supported versions: 1.2, 2.0
pyramid_swagger.swagger_versions = 2.0

# Check the correctness of Swagger spec files.
# Default: True
pyramid_swagger.enable_swagger_spec_validation = true

# Check request content against Swagger spec.
# Default: True
pyramid_swagger.enable_request_validation = true
```

(continues on next page)

(continued from previous page)

```
# Check response content against Swagger spec.
# Default: True
pyramid_swagger.enable_response_validation = true

# Check path is declared in Swagger spec.
# If disabled and an appropriate Swagger schema cannot be
# found, then request and response validation is skipped.
# Default: True
pyramid_swagger.enable_path_validation = true

# Use Python classes instead of dicts to represent models in incoming
# requests.
# Default: False
pyramid_swagger.use_models = false

# Set value for property defined in swagger schema to None if value was not provided.
# Skip property if value is missed and include_missing_properties is False.
# Default: True
pyramid_swagger.include_missing_properties = true

# Exclude certain endpoints from validation. Takes a list of regular
# expressions.
# Default: ^/static/? ^/api-docs/? ^/swagger.json
pyramid_swagger.exclude_paths = ^/static/? ^/api-docs/? ^/swagger.json

# Exclude pyramid routes from validation. Accepts a list of strings
pyramid_swagger.exclude_routes = catchall no-validation

# Path to contextmanager to handle request/response validation
# exceptions. This should be a dotted python name as per
# http://docs.pylonsproject.org/projects/pyramid/en/latest/glossary.html#term-dotted-
↪python-name
# Default: None
pyramid_swagger.validation_context_path = path.to.user.defined.contextmanager

# Enable/disable automatic /api-doc endpoints to serve the swagger
# schemas (true by default)
pyramid_swagger.enable_api_doc_views = true

# Base path for api docs (empty by default)
# Examples:
#   - leave empty and get api doc with GET /swagger.yaml
#   - set to '/help' and get api doc with GET /help/swagger.yaml
pyramid_swagger.base_path_api_docs = ''

# Enable/disable generating the /api-doc endpoint from a resource
# listing template (false by default). See `generate_resource_listing`
# below for more details
pyramid_swagger.generate_resource_listing = false

# Enable/disable serving the dereferenced swagger schema in
# a single http call. This can be slow for larger schemas.
# Note: It is not suggested to use it with Python 2.6. Known issues with
#       os.path.relpath could affect the proper behaviour.
# Default: False
pyramid_swagger.dereference_served_schema = false
```

Note: `pyramid_swawgger` uses a `bravado_core.spec.Spec` instance for handling swagger related details. You can set bravado-core config values by adding a `bravado-core.` prefix to them.

Note that, equivalently, you can add these settings during webapp configuration:

```
def main(global_config, **settings):
    # ...
    settings['pyramid_swagger.schema_directory'] = 'schemas/live/here/'
    settings['pyramid_swagger.enable_swagger_spec_validation'] = True
    # ...and so on with the other settings...
    config = Configurator(settings=settings)
    config.include('pyramid_swagger')
```

4.1 user_formats (Swagger 2.0 only)

The option `user_formats` provides user defined formats which can be used for validations/format-conversions. This options can only be used via webapp configuration.

Sample usage:

```
def main(global_config, **settings):
    # ...
    settings['pyramid_swagger.user_formats'] = [user_format]
```

`user_format` used above is an instance of `bravado_core.formatter.SwaggerFormat` and can be defined like this:

```
import base64
from pyramid_swagger.tween import SwaggerFormat
user_format = SwaggerFormat(format='base64',
                            to_wire=base64.b64encode,
                            to_python=base64.b64decode,
                            validate=base64.b64decode,
                            description='base64 conversions')
```

After defining this format, it can be used in the Swagger Spec definition like so:

```
{
    "name": "petId",
    "in": "path",
    "description": "ID of pet to return",
    "required": true,
    "type": "string",
    "format": "base64"
}
```

Note: The `type` need not be `string` always. The feature also works for other primitive types like integer, boolean, etc. More details are in the Swagger Spec v2.0 Data Types.

There are two types of validations which happen for user-defined formats. The first one is the usual type checking which is similarly done for all the other values. The second check is done by the `validate` function (from the `user_format` you configured for this type) which is run on the serialised format. If the value doesn't conform to

the format, the `validate` function MUST raise an error and that error should be `bravado_core.exception.SwaggerValidationError`.

All the parameters to `SwaggerFormat` are mandatory. If you want any of the functions to behave as a no-op, assign them a value `lambda x: x`. On providing a user-format, the default marshal/unmarshal behavior associated with that primitive type gets overridden by the `to_wire`/`to_python` behavior registered with that user-format, respectively.

4.2 validation_context_path

Formatting validation errors for API requests/responses to fit every possible swagger spec and response type is very complicated and will never cover every scenario. The `validation_context_path` option provides a way to change or format the response returned when `pyramid_swagger` validation fails.

Sample usage:

```
from pyramid_swagger import exceptions

class UserDefinedResponseError(Exception):
    pass

def validation_context(request, response=None):
    try:
        yield
    except exceptions.RequestValidationError as e:
        # Content type will be application/json instead of plain/text
        raise exceptions.RequestValidationError(json=[str(e)])

    except exceptions.ResponseValidationError as e:
        # Reraise as non-pyramid exception
        raise UserDefinedResponseError(str(e))
```

The errors that are raised from the validation_context are defined in `pyramid_swagger.exceptions`.

Note: By default `pyramid_swagger` validation errors return content type plain/text

4.3 generate_resource_listing (Swagger 1.2 only)

With a large API (many Resource objects) the boilerplate `apis` field of the Resource Listing document can become painful to maintain. This setting provides a way to relieve that burden.

When the `generate_resource_listing` option is enabled `pyramid_swagger` will automatically generate the `apis` section of the swagger Resource Listing from the list of `*.json` files in the schema directory. The `apis` listing is generated by using the name of the file (without the extension) as the `path`.

To use this feature, create an `api_docs.json` file in the schema directory. This file may contain any relevant field from Resource Listing, but it **must** exclude the `apis` field. In many cases this `api_docs.json` will only contain a single key `swaggerVersion: 1.2`.

Note: Generated Resource Listing documents will not have the optional `description` field.

4.3.1 Example

Given a schema directory with the following files

```
api_docs/
+── api_docs.json
+── pet.json
+── store.json
+── user.json
```

Previously you might have created an `api_docs.json` that looked like this

```
{
    "swaggerVersion": "1.2",
    "apiVersion": "1.0",
    "apis": [
        {
            "path": "/pet",
        },
        {
            "path": "/store",
        },
        {
            "path": "/user",
        },
    ]
}
```

When `generate_resource_listing` is enabled, the `api_docs.json` should be similar, but with the `apis` section removed.

```
{
    "swaggerVersion": "1.2",
    "apiVersion": "1.0",
}
```

`pyramid_swagger` will generate a Resource Listing which is equivalent to the original `api_docs.json` with a full `apis` list.

CHAPTER 5

Migrating to Swagger 2.0

So you're using pyramid_swagger with Swagger 1.2 and now it is time to upgrade to Swagger 2.0.

Just set the version of Swagger to support via configuration.

```
[app:main]
pyramid_swagger.swagger_versions = ['2.0']
```

If you would like to continue servicing Swagger 1.2 clients, pyramid_swagger has you covered.

```
[app:main]
pyramid_swagger.swagger_versions = ['1.2', '2.0']
```

Note: When both versions of Swagger are supported, all requests are validated against the 2.0 version of the schema only. Make sure that your 1.2 and 2.0 schemas define an identical set of APIs.

If you're not using an ini file, configuration in Python also works.

```
def main(global_config, **settings):
    # ...
    settings['pyramid_swagger.swagger_versions'] = ['2.0']
    # ...and so on with the other settings...
    config = Configurator(settings=settings)
    config.include('pyramid_swagger')
```

Next, create a Swagger 2.0 version of your swagger schema. There are some great resources to help you with the conversion process.

- Swagger 1.2 to 2.0 Migration Guide
- Swagger Converter
- Swagger 2.0 Specification

Finally, place your Swagger 2.0 schema `swagger.json` file in the same directory as your Swagger 1.2 schema and you're ready to go.

5.1 Incremental Migration

If your v1.2 spec is too large and you are looking to migrate specs incrementally, then the below config can be useful.

```
[app:main]
pyramid_swagger.prefer_20_routes = ['route_foo']
```

Note: The above config is read only when both *['1.2', '2.0']* are present in *swagger_versions* config. If that is the case and the request's route is present in *prefer_20_routes*, ONLY then the request is served through swagger 2.0 otherwise through 1.2. The only exception is either the config is not defined at all or both of the swagger versions are not enabled, in any of these cases, v2.0 is preferred (as mentioned in above note).

CHAPTER 6

External resources

There are a variety of external resources you will find useful when documenting your API with Swagger.

- Interactive Spec Editor
- Swagger 1.2 Specification
- Swagger 2.0 Specification
- “Pet Store” example API

CHAPTER 7

Glossary of Terms

Nothing more than some common vocabulary for you to absorb.

swagger api-docs (often swagger api) The preferred term for the resource listing and associated api declarations. This is so-named to avoid confusion with the Swagger Specification and the actual implementation of your service.

resource listing The top-level declaration of the various Swagger resources your service exposes. Each resource must have an associated api declaration.

api declaration The description of each endpoint a particular Swagger service provides, with complete input and output declared.

swagger spec The formal specification of a valid swagger api. The current public version is 2.0 and hosted on wordnik's Github.