Doctor Documentation

Release 3.0.0-beta.5

Upsight

Mar 05, 2018

Contents

1 Usage 3

2 Release History 45

3 Indices and tables 49

Python Module Index 51

HTTP Routing Table 53

This module uses python types to validate request and response data in Flask Python APIs and generate API documentation. It uses python 3 type hints to validate request parameters. It also supports generic schema validation for plain dictionaries.

CHAPTER 1

Usage

1.1 Using in Flask

doctor provides some helpers to for usage in a flask-restful application. You can find an example in `app.py`. To run this application, you'll first need to install both doctor, flask, and flask-restful. Then run:

```
python app.py
```

The application will be running on http://127.0.0.1:5000.

1.1.1 Doctor Types

doctor types are classes that represent your request data and your responses. Each parameter of your logic function must specify a type hint which is a subclass of one of the *builtin doctor types*. These types perform validation on the request parameters passed to the logic function. See *Types* for more information.

```
from doctor import types


# doctor provides helper functions to easily define simple types.
Body = types.string('Note body', example='body')
Done = types.boolean('Marks if a note is done or not.', example=False)
NoteId = types.integer('Note ID', example=1)
Status = types.string('API status')
NoteType = types.enum('The type of note', enum=['quick', 'detailed'],
                      example='quick')


# You can also inherit from type classes to create more complex types.
class Note(types.Object):
    description = 'A note object'
    additional_properties = False
```

(continues on next page)

(continued from previous page)

```
    properties = {
        'note_id': NoteId,
        'body': Body,
        'done': Done,
    }
    required = ['body', 'done', 'note_id']
    example = {
        'body': 'Example Body',
        'done': True,
        'note_id': 1,
    }


Notes = types.array('Array of notes', items=Note, example=[Note.example])
```

1.1.2 Logic Functions

Next, we'll also need some logic functions. doctor's *create_routes()* generates HTTP handler methods that wrap normal Python callables, so the code can focus on more logic and less on HTTP. These handlers and HTTP handler methods will be automatically generated for you based on your defined routes.

The logic function signature will be used to determine what the request parameters are for the route/HTTP method and to determine which are required. Each parameter must specify a type hint which is a subclass of one of the *builtin doctor types*. These types perform validation on the request parameters passed to the logic function. See *Types* for more information. Any argument without a default value is considered required while others are optional. For example in the *create_note* function below, *body* would be a required request parameter and *done* would be an optional request parameter.

Any parameters that don't validate will raise a *HTTP400Exception*. This exception will contain all validation errors and missing required properties. If there is more than one error, you can access them from the exception's `errobj` attribute.

To abstract out the HTTP layer in logic functions, doctor provides custom exceptions which will be converted to the correct HTTP Exception by the library. See the *Error Classes* documentation for more information on which exception your code should raise.

```
note = {'note_id': 1, 'body': 'Example body', 'done': True}


# Note the type annotations on this function definition. This tells Doctor how
# to parse and validate parameters for routes attached to this logic function.
# The return type annotation will validate the response conforms to an
# expected definition in development environments.  In non-development
# environments a warning will be logged.
def get_note(note_id: NoteId, note_type: NoteType) -> Note:
    """Get a note by ID."""
    if note_id != 1:
        raise NotFoundError('Note does not exist')
    return note


def get_notes() -> Notes:
    """Get a list of notes."""
```

(continues on next page)

(continued from previous page)

```
    return [note]


def create_note(body: Body, done: Done=False) -> Note:
    """Create a new note."""
    return {'note_id': 2,
            'body': body,
            'done': done}


def update_note(note_id: NoteId, body: Body=None, done: Done=None) -> Note:
    """Update an existing note."""
    if note_id != 1:
        raise NotFoundError('Note does not exist')
    new_note = note.copy()
    if body is not None:
        new_note['body'] = body
    if done is not None:
        new_note['done'] = done
    return new_note


def delete_note(note_id: NoteId):
    """Delete an existing note."""
    if note_id != 1:
        raise NotFoundError('Note does not exist')


def status() -> Status:
    return 'Notes API v1.0.0'
```

1.1.3 Creating Routes

Routes map a url to one or more HTTP methods which each map to a specific logic function. We define our routes by instantiating a *Route*. A *Route* requires 2 arguments. The first is the URL-matching pattern e.g. */foo/<int:foo_id>/*. The second is a tuple of allowed *HTTPMethod* s for the matching pattern: *get()*, *post()*, *put()* and *delete()*.

The HTTP method functions take one required argument which is the *logic function* to call when the http method for that uri is called.

```
routes = (
    Route('/', methods=(
        get(status),), heading='API Status'),
    Route('/note/', methods=(
        get(get_notes, title='Retrieve List'),
        post(create_note)), handler_name='NoteListHandler', heading='Notes (v1)'
    ),
    Route('/note/<int:note_id>/', methods=(
        delete(delete_note),
        get(get_note),
        put(update_note)), heading='Notes (v1)'
    ),
```

(continues on next page)

(continued from previous page)

```
)
```

We then create our Flask app and add our created resources to it. These resources are created by calling *create_routes()* with our routes we defined above.

```
app = Flask('Doctor example')

api = Api(app)
for route, resource in create_routes(routes):
    api.add_resource(resource, route)

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

1.1.4 Adding Response Headers

If you need more control over the response, your logic function can return a *Response* instance. For example if you would like to have your logic function force download a csv file you could do the following:

```
from doctor.response import Response

def download_csv():
    data = '1,2,3\n4,5,6\n'
    return Response(data, {
        'Content-Type': 'text/csv',
        'Content-Disposition': 'attachment; filename=data.csv',
    })
```

The *Response* class takes the response data as the first parameter and a dict of HTTP response headers as the second parameter. The response headers can contain standard and any custom values.

1.1.5 Response Validation

doctor can also validate your responses.

Enabling

By default doctor will only raise exceptions for invalid response when there is a truthy value for the environment variable *RAISE_RESPONSE_VALIDATION_ERRORS*. This will cause a HTTP 400 error which wil give details on why the response is not valid. If either of those conditions are not true only a warning will be logged.

Usage

To tell doctor to validate a response you must define a return annotation on your logic function. Simply use doctor types to define a valid response and annotate it on your logic function.

```
from doctor import types

Color = types.enum('A color', enum=['blue', 'green'])
Colors = types.array('Array of colors', items=Color)

def get_colors() -> Colors:
    # ... logic to fetch colors
    return colors
```

The return value of *get_colors* will be validated against the type that we created. If we have enabled raising response validation errors and our response does not validate, we will get a 400 response. If the above example returned an array with an integer like *[1]* our response would look like:

```
` Response to GET /colors `[1]` does not validate: {0: 'Must be a valid
choice.'} `
```

1.1.6 Example API Documentation

This API documentation is generated using the `autoflask` Sphinx directive. See the section on *Generating API Documentation* for more information.

API Status

Retrieve

GET /

Request Headers

- Authorization – The auth token for the authenticated user.

Example Request:

```
curl http://127.0.0.1:8080/ -X GET -H 'Authorization: testtoken'
```

Example Response:

```
"Notes API v1.0.0"
```

Notes (v1)

Create

POST /note/

Create a new note.

Request JSON Object

- **body** (*str*) – **Required**. Note body
- **done** (*bool*) – Marks if a note is done or not.

Request Headers

- Authorization – The auth token for the authenticated user.

Response JSON Object

- **body** (*str*) – Note body
- **done** (*bool*) – Marks if a note is done or not.
- **note_id** (*int*) – Note ID

Example Request:

```
curl http://127.0.0.1:8080/note/ -X POST -H 'Authorization: testtoken' \
  -H 'Content-Type: application/json' -d \
  '{
    "body": "body",
    "done": false
   }'
```

Example Response:

```
{"body": "body", "done": false, "note_id": 2}
```

Delete

DELETE /note/(**int:** *note_id*)**/**

Delete an existing note.

Query Parameters

- **note_id** (*int*) – **Required**. Note ID

Request Headers

- Authorization – The auth token for the authenticated user.

Example Request:

```
curl http://127.0.0.1:8080/note/1/ -X DELETE -H 'Authorization: testtoken'
```

Example Response:

```

```

Retrieve

GET /note/(**int:** *note_id*)**/**

Get a note by ID.

Query Parameters

- **note_id** (*int*) – **Required**. Note ID
- **note_type** (*str*) – **Required**. The type of note Must be one of: *['quick', 'detailed']*.

Request Headers

- Authorization – The auth token for the authenticated user.

Response JSON Object

- **body** (*str*) – Note body

- **done** (*bool*) – Marks if a note is done or not.
- **note_id** (*int*) – Note ID

Example Request:

```
curl 'http://127.0.0.1:8080/note/1/?note_type=quick' -X GET \
  -H 'Authorization: testtoken'
```

Example Response:

```
{"body": "Example body", "done": true, "note_id": 1}
```

Retrieve List

GET /note/
Get a list of notes.

Request Headers

- Authorization – The auth token for the authenticated user.
- *X-GeoIp-Country* – An ISO 3166-1 alpha-2 country code.

Response JSON Array of Objects

- **body** (*str*) – Note body
- **done** (*bool*) – Marks if a note is done or not.
- **note_id** (*int*) – Note ID

Example Request:

```
curl http://127.0.0.1:8080/note/ -X GET -H 'Authorization: testtoken' \
  -H 'X-GeoIp-Country: US'
```

Example Response:

```
[{"body": "Example body", "done": true, "note_id": 1}]
```

Update

PUT /note/(int: *note_id*)/
Update an existing note.

Request JSON Object

- **note_id** (*int*) – **Required**. Note ID
- **body** (*str*) – Note body
- **done** (*bool*) – Marks if a note is done or not.

Request Headers

- Authorization – The auth token for the authenticated user.

Response JSON Object

- **body** (*str*) – Note body

- **done** (*bool*) – Marks if a note is done or not.
- **note_id** (*int*) – Note ID

Example Request:

```
curl http://127.0.0.1:8080/note/1/ -X PUT -H 'Authorization: testtoken' \
  -H 'Content-Type: application/json' -d \
  '{
    "body": "body",
    "done": false,
    "note_id": 1
   }'
```

Example Response:

```
{"body": "body", "done": false, "note_id": 1}
```

1.1.7 Flask Module Documentation

exception doctor.flask.**HTTP400Exception**(*description=None*, *errors=None*)
Represents a HTTP 400 error.

Parameters

- **description** (Optional[str]) – The error description.
- **errors** (Optional[dict]) – A dict containing all validation errors during the request. The key is the param name and the value is the error message.

exception doctor.flask.**HTTP401Exception**(*description=None*, *errors=None*)

exception doctor.flask.**HTTP403Exception**(*description=None*, *errors=None*)

exception doctor.flask.**HTTP404Exception**(*description=None*, *errors=None*)

exception doctor.flask.**HTTP409Exception**(*description=None*, *errors=None*)

exception doctor.flask.**HTTP500Exception**(*description=None*, *errors=None*)

exception doctor.flask.**SchematicHTTPException**(*description=None*, *errors=None*)
Schematic specific sub-class of werkzeug's BadRequest.

Note that this adds a flask-restful specific data attribute to the class, as the error wouldn't render properly without it.

Parameters

- **description** (Optional[str]) – The error description.
- **errors** (Optional[dict]) – A dict containing all validation errors during the request. The key is the param name and the value is the error message.

doctor.flask.**create_routes**(*routes*)
A thin wrapper around create_routes that passes in flask specific values.

Parameters **routes** (Tuple[*Route*]) – A tuple containing the route and another tuple with all http methods allowed for the route.

Return type List[Tuple[str, Resource]]

Returns A list of tuples containing the route and generated handler.

doctor.flask.**handle_http**(*handler*, *args*, *kwargs*, *logic*)
Handle a Flask HTTP request

Parameters

- **handler** (Resource) – flask_restful.Resource: An instance of a Flask Restful resource class.
- **args** (*tuple*) – Any positional arguments passed to the wrapper method.
- **kwargs** (*dict*) – Any keyword arguments passed to the wrapper method.
- **logic** (*callable*) – The callable to invoke to actually perform the business logic for this request.

doctor.flask.**should_raise_response_validation_errors**()
Returns if the library should raise response validation errors or not.

If the environment variable *RAISE_RESPONSE_VALIDATION_ERRORS* is set, it will return True.

Return type bool

Returns True if it should, False otherwise.

1.2 Generating API Documentation

You can use Doctor to generate Sphinx documentation for your API. It will introspect the list of routes for your Flask app, and will use the values from your schema to generate a list of parameters for those routes.

For example, to generate API documentation for the example Flask app, you would add `doctor.docs.flask` to the extensions list in Sphinx's conf.py file:

```
extensions = [
    'sphinx.ext.autodoc',
    'sphinx.ext.coverage',
    'sphinx.ext.viewcode',
    'doctor.docs.flask',
]
```

You'll also need to import and instantiate *AutoFlaskHarness* in conf.py:

```
from doctor.docs.flask import AutoFlaskHarness
autoflask_harness = AutoFlaskHarness(
    routes_filename='examples/flask/app.py',
    url_prefix='http://127.0.0.1:8080')
```

This harness class provides setup and teardown handlers that are used to load your Flask application. The documentation directives use the harness to introspect and make mock requests against your app. If you have custom setup and teardown steps that you would like to take (such as loading fixtures into a database), you can subclass it and customize it. Take a look at *BaseHarness* for a list of the hooks that are available.

If you are adding extra logic to the harness and subclassing *AutoFlaskHarness*, make note of the signature of *setup_app()*. The sphinx_app parameter is not the Flask application. To access the Flask application object, use *self.app*. e.g.

```
from doctor.docs.flask import AutoFlaskHarness
from myapp import db
class MyCustomHarness(AutoFlaskHarness):
    def setup_app(self, sphinx_app):
```

(continues on next page)

(continued from previous page)

```
        super(MyCustomHarness, self).setup_app(sphinx_app)
        with self.app.app_context():
            db.init_app(self.app) # initialize sqlalchemy db extension
```

Then, add an `autoflask` directive to one of your rst files:

```
API Documentation
-----------------

.. autoflask::
```

When you run Sphinx, it will render documentation like this:

1.2.1 API Status

Retrieve

GET /

Request Headers

- Authorization – The auth token for the authenticated user.

Example Request:

```
curl http://127.0.0.1:8080/ -X GET -H 'Authorization: testtoken'
```

Example Response:

```
"Notes API v1.0.0"
```

1.2.2 Notes (v1)

Create

POST /note/

Create a new note.

Request JSON Object

- **body** (*str*) – **Required**. Note body
- **done** (*bool*) – Marks if a note is done or not.

Request Headers

- Authorization – The auth token for the authenticated user.

Response JSON Object

- **body** (*str*) – Note body
- **done** (*bool*) – Marks if a note is done or not.
- **note_id** (*int*) – Note ID

Example Request:

```
curl http://127.0.0.1:8080/note/ -X POST -H 'Authorization: testtoken' \
  -H 'Content-Type: application/json' -d \
  '{
    "body": "body",
    "done": false
   }'
```

Example Response:

```
{"body": "body", "done": false, "note_id": 2}
```

Delete

DELETE /note/(**int:** *note_id*)**/**

Delete an existing note.

Query Parameters

- **note_id** (*int*) – **Required**. Note ID

Request Headers

- Authorization – The auth token for the authenticated user.

Example Request:

```
curl http://127.0.0.1:8080/note/1/ -X DELETE -H 'Authorization: testtoken'
```

Example Response:

```

```

Retrieve

GET /note/(**int:** *note_id*)**/**

Get a note by ID.

Query Parameters

- **note_id** (*int*) – **Required**. Note ID
- **note_type** (*str*) – **Required**. The type of note Must be one of: *['quick', 'detailed']*.

Request Headers

- Authorization – The auth token for the authenticated user.

Response JSON Object

- **body** (*str*) – Note body
- **done** (*bool*) – Marks if a note is done or not.
- **note_id** (*int*) – Note ID

Example Request:

```
curl 'http://127.0.0.1:8080/note/1/?note_type=quick' -X GET \
  -H 'Authorization: testtoken'
```

Example Response:

```
{"body": "Example body", "done": true, "note_id": 1}
```

Retrieve List

GET /note/
Get a list of notes.

Request Headers

- Authorization – The auth token for the authenticated user.
- *X-GeoIp-Country* – An ISO 3166-1 alpha-2 country code.

Response JSON Array of Objects

- **body** (*str*) – Note body
- **done** (*bool*) – Marks if a note is done or not.
- **note_id** (*int*) – Note ID

Example Request:

```
curl http://127.0.0.1:8080/note/ -X GET -H 'Authorization: testtoken' \
  -H 'X-GeoIp-Country: US'
```

Example Response:

```
[{"body": "Example body", "done": true, "note_id": 1}]
```

Update

PUT /note/(int: *note_id*)/
Update an existing note.

Request JSON Object

- **note_id** (*int*) – **Required**. Note ID
- **body** (*str*) – Note body
- **done** (*bool*) – Marks if a note is done or not.

Request Headers

- Authorization – The auth token for the authenticated user.

Response JSON Object

- **body** (*str*) – Note body
- **done** (*bool*) – Marks if a note is done or not.
- **note_id** (*int*) – Note ID

Example Request:

```
curl http://127.0.0.1:8080/note/1/ -X PUT -H 'Authorization: testtoken' \
  -H 'Content-Type: application/json' -d \
  '{
    "body": "body",
    "done": false,
    "note_id": 1
   }'
```

Example Response:

```
{"body": "body", "done": false, "note_id": 1}
```

1.2.3 Grouping Related API Endpoints Under A Heading

You can specify a heading to group together related api routes when generating api documentation. To do this, simply pass a value to the *heading* kwarg when defining your Route.

```
from doctor.routing import delete, get, put, post, Route

routes = (
    Route('/', methods=(
        get(status, title='Show API Version'),), heading='API Status'),
    Route('/note/', methods=(
        get(get_notes, title='Get Notes'),
        post(create_note, title='Create Note'), heading='Notes')
    ),
    Route('/note/<int:note_id>/', methods=(
        delete(delete_note, title='Delete Note'),
        get(get_note, title='Get Note'),
        put(update_note, title='Update Note'), heading='Notes')
    ),
)
```

1.2.4 Customizing API Endpoint Headings

You can specify a short title when creating the routes which will show up as a sub link below the group heading. To do this, pass a value to *title* kwarg when defining your http methods for a route. If a title is not provided, one will be generated based on the http method. The automatic title will be one of *Retrieve*, *Delete*, *Create*, or *Update*.

```
from doctor.routing import delete, get, put, post, Route

routes = (
    Route('/', methods=(
        get(status, title='Show API Version'),)),
    Route('/note/', methods=(
        get(get_notes, title='Get Notes'),
        post(create_note, title='Create Note'))
    ),
    Route('/note/<int:note_id>/', methods=(
        delete(delete_note, title='Delete Note'),
        get(get_note, title='Get Note'),
        put(update_note, title='Update Note'))
    ),
)
```

1.2.5 Overriding Example Values For Specific Endpoints

By default doctor will use the example value you specified on your custom type or if one wasn't given, the default example for the subclass of your type. Sometimes you need to set a very specific value for a parameter in a request when generating documentation. doctor supports this behavior by using *define_example_values()*. This method allows you to override parameters on a per request basis. To do this subclass the *AutoFlaskHarness* and override the *setup_app()* method. Then you can define example values for a particular route and method.

```
from doctor.docs.flask import AutoFlaskHarness

class MyHarness(AutoFlaskHarness):
    def setup_app(self, sphinx_app):
        super(MyHarness, self).setup_app(sphinx_app)
        self.define_example_values('GET', '^/foo/bar/?$', {'foobar': 1})
```

The above code sample will change the parameters sent when sending a *GET* request to */foo/bar* when generating documentation for that route. You can call this method for as many routes as you need to provide custom parameters.

Remember if you create your own harness you'll need to update the harness class that you instantiate in *conf.py*.

Note: For a flask api the 2nd parameter passed to *define_example_values()* is the route pattern as a string. e.g. */foo/bar/*.

1.2.6 Documenting and Sending Headers on Requests

If you need to pass header values for a request you can define them in two ways.

The first method will add the header to all requests when generating documentation. An example where this may be useful is an Authorization header. To add this simply define a *headers* dict on your harness. If you would like to provide a definition in the documentation for the header, also define a *header_definitions* dict where the header key matches the header you wish to document.

```
from doctor.docs.flask import AutoFlaskHarness

class MyHarness(AutoFlaskHarness):
    headers = {'Authorization': 'testtoken'}
    header_definitions = {
        'Authorization': 'The auth token for the authenticated user.'}

    def setup_app(self, sphinx_app):
        super(MyHarness, self).setup_app(sphinx_app)
```

If you need to define a header for a specific route and method you can set those up in your harness using *define_header_values()*.

```
from doctor.docs.flask import AutoFlaskHarness

class MyHarness(AutoFlaskHarness):
    headers = {'Authorization': 'testtoken'}
    header_definitions = {
        'Authorization': 'The auth token for the authenticated user.',
        'X-GeoIp-Country': 'An ISO 3166-1 alpha-2 country code.'}

    def setup_app(self, sphinx_app):
```

(continues on next page)

(continued from previous page)

```
        super(MyHarness, self).setup_app(sphinx_app)
        self.define_header_values('GET', '/foo/bar/', {'X-GeoIp-Country': 'US'})
```

The above harness will send the *Authorization* header on all requests and will additionally send the *X-GeoIp-Country* header on a GET request to */foo/bar/*.

1.2.7 Module Documentation

This module can be used to generate Sphinx documentation for an API.

class doctor.docs.base.**BaseDirective**(*name*, *arguments*, *options*, *content*, *lineno*, *content_offset*, *block_text*, *state*, *state_machine*)

Bases: docutils.parsers.rst.Directive

Base class for doctor Sphinx directives.

You probably want to use *AutoFlaskDirective* instead of this class.

_prepare_env()

Setup the document's environment, if necessary.

_render_rst()

Render lines of reStructuredText for items yielded by *iter_annotations()*.

directive_name = None

Name to use for this directive.

This is the identifier used within the Sphinx documentation to trigger this directive. This value should be set by subclasses. For example, in AutoFlaskDirective, this is set to "autoflask".

classmethod get_outdated_docs(*app*, *env*, *added*, *changed*, *removed*)

Handler for Sphinx's env-get-outdated event.

This handler gives a Sphinx extension a chance to indicate that some set of documents are out of date and need to be re-rendered. The implementation here is stupid, for now, and always says that anything that uses the directive needs to be re-rendered.

We should make it smarter, at some point, and have it figure out which modules are used by the associated handlers, and whether they have actually been updated since the last time the given document was rendered.

harness = None

Harness for the Flask app this directive is documenting. This is responsible for setting up and tearing down the mock app. It is defined in Sphinx's conf.py file, and set on the directive in run_setup(). It should be an instance of *BaseHarness*.

has_content = True

Indicates to Sphinx that this directive will yield content.

classmethod purge_docs(*app*, *env*, *docname*)

Handler for Sphinx's env-purge-doc event.

This event is emitted when all traces of a source file should be cleaned from the environment (that is, if the source file is removed, or before it is freshly read). This is for extensions that keep their own caches in attributes of the environment.

For example, there is a cache of all modules on the environment. When a source file has been changed, the cache's entries for the file are cleared, since the module declarations could have been removed from the file.

run()
Called by Sphinx to generate documentation for this directive.

classmethod setup(*app*)
Called by Sphinx to setup an extension.

class doctor.docs.base.**BaseHarness**(*url_prefix*)
Bases: object

Base class for doctor directive harnesses. A harness is defined in Sphinx's conf.py, and the directive invokes the various methods at the appropriate times, so the app can bootstrap a mock version of itself.

_get_annotation_heading(*handler*, *route*, *heading=None*)
Returns the heading text for an annotation.

Attempts to get the name of the heading from the handler attribute *schematic_title* first.

If *schematic_title* it is not present, it attempts to generate the title from the class path. This path: advertiser_api.handlers.foo_bar.FooListHandler would translate to 'Foo Bar'

If the file name with the resource is generically named handlers.py or it doesn't have a full path then we attempt to get the resource name from the class name. So FooListHandler and FooHandler would translate to 'Foo'. If the handler class name starts with 'Internal', then that will be appended to the heading. So InternalFooListHandler would translate to 'Foo (Internal)'

Parameters

- **handler** (*mixed*) – The handler class. Will be a flask resource class
- **route** (*str*) – The route to the handler.

Returns The text for the heading as a string.

_get_example_values(*route*, *annotation*)
Gets example values for all properties in the annotation's schema.

Parameters

- **route** (*werkzeug.routing.Rule for a flask api.*) – The route to get example values for.
- **annotation** (doctor.resource.ResourceAnnotation) – Schema annotation for the method to be requested.

Retruns A dict containing property names as keys and example values as values.

Return type Dict[str, Any]

_get_headers(*route*, *annotation*)
Gets headers for the provided route.

Parameters

- **route** (*werkzeug.routing.Rule for a flask api.*) – The route to get example values for.
- **annotation** (doctor.resource.ResourceAnnotation) – Schema annotation for the method to be requested.

Retruns A dict containing headers.

Return type Dict[~KT, ~VT]

define_example_values(*http_method*, *route*, *values*, *update=False*)
Define example values for a given request.

By default, example values are determined from the example properties in the schema. But if you want to change the example used in the documentation for a specific route, and this method lets you do that.

Parameters

- **http_method** (*str*) – An HTTP method, like "get".
- **route** (*str*) – The route to match.
- **values** (*dict*) – A dictionary of parameters for the example request.
- **update** (*bool*) – If True, the values will be merged into the default example values for the request. If False, the values will replace the default example values.

define_header_values(*http_method*, *route*, *values*, *update=False*)

Define header values for a given request.

By default, header values are determined from the class attribute *headers*. But if you want to change the headers used in the documentation for a specific route, this method lets you do that.

Parameters

- **http_method** (*str*) – An HTTP method, like "get".
- **route** (*str*) – The route to match.
- **values** (*dict*) – A dictionary of headers for the example request.
- **update** (*bool*) – If True, the values will be merged into the default headers for the request. If False, the values will replace the default headers.

defined_header_values = None

Stores headers for particular methods and routes.

header_definitions = None

Stores definitions for header keys for documentation.

headers = None

Stores global headers to use on all requests

iter_annotations()

Yield a tuple for each schema annotated handler to document.

This must be implemented by subclasses. See *AutoFlaskHarness* for an example implementation.

request(*route*, *handler*, *annotation*)

Make a request against the app.

This must be implemented by subclasses. See *AutoFlaskHarness* for an example implementation.

setup_app(*sphinx_app*)

Called once before building documentation.

Parameters **sphinx_app** – Sphinx application object.

setup_request(*sphinx_directive*, *route*, *handler*, *annotation*)

Called before each request to the mock app.

Parameters

- **sphinx_directive** (BaseDirective) – The directive that is making the mock request.
- **route** – Path for the route. For Flask, this will be a Route object.
- **handler** – Flask resource for the route.

- **annotation** (ResourceAnnotation) – Annotation for the request.

teardown_app(*sphinx_app*)
Called once after building documentation.

Parameters **sphinx_app** – Sphinx application object.

teardown_request(*sphinx_directive*, *route*, *handler*, *annotation*)
Called after each request to the mock app.

Parameters

- **sphinx_directive** (BaseDirective) – The directive that is making the mock request.
- **route** – Path for the route. For Flask, this will be a Route object.
- **handler** – Flask resource for the route.
- **annotation** (ResourceAnnotation) – Annotation for the request.

doctor.docs.base.**CAMEL_CASE_RE = re.compile('[A-Z][^A-Z]*')**
Used to transform a class name into it's various words, splitting on uppercase characters. So *MyClassName* becomes ['My', 'Class', 'Name']

class doctor.docs.base.**DirectiveState**
Bases: object

This is used to hold Sphinx serialized state for our directives.

doctor.docs.base.**HTTP_METHODS = ('get', 'post', 'put', 'patch', 'delete')**
These are the HTTP methods that will be documented on handlers.

Note that HEAD and OPTIONS aren't included here.

doctor.docs.base.**TYPE_MAP = {'array': 'list', 'boolean': 'bool', 'integer': 'int', 'numb**
Used to map the JSON schema types to more Pythonic types for consistency.

doctor.docs.base.**URL_PARAMS_RE = re.compile('\\(([a-zA-Z_]+)\\:')**
Used to get all url parameter names.

doctor.docs.base.**get_example_curl_lines**(*method*, *url*, *params*, *headers*)
Render a cURL command for the given request.

Parameters

- **method** (*str*) – HTTP request method (e.g. "GET").
- **url** (*str*) – HTTP request URL.
- **params** (*dict*) – JSON body, for POST and PUT requests.
- **headers** (*dict*) – A dict of HTTP headers.

Return type List[str]

Returns list

doctor.docs.base.**get_example_lines**(*headers*, *method*, *url*, *params*, *response*)
Render a reStructuredText example for the given request and response.

Parameters

- **headers** (*dict*) – A dict of HTTP headers.
- **method** (*str*) – HTTP request method (e.g. "GET").
- **url** (*str*) – HTTP request URL.

- **params** (*dict*) – Form parameters, for POST and PUT requests.
- **response** (*str*) – Text response body.

Return type List[str]

Returns list

doctor.docs.base.**get_json_lines**(*annotation*, *field*, *route*, *request=False*)

Generate documentation lines for the given annotation.

This only documents schemas of type "object", or type "list" where each "item" is an object. Other types are ignored (but a warning is logged).

Parameters

- **annotation** (doctor.resource.ResourceAnnotation) – Annotation object for the associated handler method.
- **field** (*str*) – Sphinx field type to use (e.g. '<json').
- **route** (*str*) – The route the annotation is attached to.
- **request** (*bool*) – Whether the resource annotation is for the request or not.

Return type List[~T]

Returns list of strings, one for each line.

doctor.docs.base.**get_json_object_lines**(*annotation*, *properties*, *field*, *url_params*, *request=False*, *object_property=False*)

Generate documentation for the given object annotation.

Parameters

- **annotation** (doctor.resource.ResourceAnnotation) – Annotation object for the associated handler method.
- **field** (*str*) – Sphinx field type to use (e.g. '<json').
- **url_params** (*list*) – A list of url parameter strings.
- **request** (*bool*) – Whether the schema is for the request or not.
- **object_property** (*bool*) – If True it indicates this is a property of an object that we are documenting. This is only set to True when called recursively when encountering a property that is an object in order to document the properties of it.

Return type List[str]

Returns list of strings, one for each line.

doctor.docs.base.**get_name**(*value*)

Return a best guess at the qualified name for a class or function.

Parameters **value** (*class or function*) – A class or function object.

Returns str

Return type str

doctor.docs.base.**normalize_route**(*route*)

Strip some of the ugly regexp characters from the given pattern.

```
>>> normalize_route('^/user/<user_id:int>/?$')
u'/user/(user_id:int)/'
```

Return type str

doctor.docs.base.**prefix_lines**(*lines*, *prefix*)

Add the prefix to each of the lines.

```
>>> prefix_lines(['foo', 'bar'], '  ')
['  foo', '  bar']
>>> prefix_lines('foo\nbar', '  ')
['  foo', '  bar']
```

Parameters

- **or str lines** (*list*) – A string or a list of strings. If a string is passed, the string is split using splitlines().
- **prefix** (*str*) – Prefix to add to the lines. Usually an indent.

Returns list

This module provides Sphinx directives to generate documentation for Flask resources which have been annotated with schema information. It is broken into a separate module so that Flask applications using doctor for validation don't need to include Sphinx in their runtime dependencies.

class doctor.docs.flask.**AutoFlaskDirective**(*name*, *arguments*, *options*, *content*, *lineno*, *content_offset*, *block_text*, *state*, *state_machine*)

Bases: *doctor.docs.base.BaseDirective*

Sphinx directive to document schema annotated Flask resources.

class doctor.docs.flask.**AutoFlaskHarness**(*app_module_filename*, *url_prefix*)

Bases: *doctor.docs.base.BaseHarness*

iter_annotations()

Yield a tuple for each Flask handler containing annotated methods.

Each tuple contains a heading, routing rule, the view class associated with the rule, and the annotations for the methods in that class.

request(*rule*, *view_class*, *annotation*)

Make a request against the app.

This attempts to use the schema to replace any url params in the path pattern. If there are any unused parameters in the schema, after substituting the ones in the path, they will be sent as query string parameters or form parameters. The substituted values are taken from the "example" value in the schema.

Returns a dict with the following keys:

- **url** – Example URL, with url_prefix added to the path pattern, and the example values substituted in for URL params.
- **method** – HTTP request method (e.g. "GET").
- **params** – A dictionary of query string or form parameters.
- **response** – The text response to the request.

Parameters

- **route** – Werkzeug Route object.
- **view_class** – View class for the annotated method.

- **annotation** (doctor.resource.ResourceAnnotation) – Annotation for the method to be requested.

Returns dict

setup_app(*sphinx_app*)
Called once before building documentation.

Parameters **sphinx_app** – Sphinx application object.

doctor.docs.flask.**setup**(*app*)
This setup function is called by Sphinx.

1.3 Schemas

class doctor.schema.**Schema**(*schema*, *schema_path=None*)
This class is used to manipulate JSON schemas and validate values against the schema.

Parameters

- **schema** (*dict*) – The loaded schema.
- **schema_path** (*str*) – The absolute path to the directory of local schemas.

classmethod from_file(*schema_filepath*, **args*, ***kwargs*)
Create an instance from a YAML or JSON schema file.

Any additional args or kwargs will be passed on when constructing the new schema instance (useful for subclasses).

Parameters **schema_filepath** (*str*) – Path to the schema file.

Returns an instance of the class.

Raises ***SchemaLoadingError*** – for invalid input files.

get_validator(*schema=None*)
Get a jsonschema validator.

Parameters **schema** (*dict*) – A custom schema to validate against.

Returns an instance of jsonschema Draft4Validator.

resolve(*ref*, *document=None*)
Resolve a ref within the schema.

This is just a convenience method, since RefResolver returns both a URI and the resolved value, and we usually just need the resolved value.

Parameters

- **ref** (*str*) – URI to resolve.
- **document** (*dict*) – Optional schema in which to resolve the URI.

Returns the portion of the schema that the URI references.

See *SchemaRefResolver.resolve()*

resolver
jsonschema RefResolver object for the base schema.

validate(*value*, *validator*)
Validates and returns the value.

If the value does not validate against the schema, SchemaValidationError will be raised.

Parameters

- **value** – A value to validate (usually a dict).
- **validator** – An instance of a jsonschema validator class, as created by Schema.get_validator().

Returns the passed value.

Raises

- ***SchemaValidationError*** –
- **Exception** –

validate_json(*json_value*, *validator*)
Validates and returns the parsed JSON string.

If the value is not valid JSON, ParseError will be raised. If it is valid JSON, but does not validate against the schema, SchemaValidationError will be raised.

Parameters

- **json_value** (*str*) – JSON value.
- **validator** – An instance of a jsonschema validator class, as created by Schema.get_validator().

Returns the parsed JSON value.

class doctor.schema.**SchemaRefResolver**(*base_uri*, *referrer*, *store=()*, *cache_remote=True*, *handlers=()*, *urljoin_cache=None*, *remote_cache=None*)
Subclass in order to provide support for loading YAML files.

_format_stack(*stack*, *current=None*)
Prettifies a scope stack for use in error messages.

Parameters

- **stack** (*list (str)*) – List of scopes.
- **current** (*str*) – The current scope. If specified, will be appended onto the stack before formatting.

Returns str

resolve(*ref*, *document=None*)
Resolve a fragment within the schema.

If the resolved value contains a $ref, it will attempt to resolve that as well, until it gets something that is not a reference. Circular references will raise a SchemaError.

Parameters

- **ref** (*str*) – URI to resolve.
- **document** (*dict*) – Optional schema in which to resolve the URI.

Returns a tuple of the final, resolved URI (after any recursion) and resolved value in the schema that the URI references.

Raises ***SchemaError*** –

resolve_remote(*uri*)
Add support to load YAML files.

This will attempt to load a YAML file first, and then go back to the default behavior.

Parameters **uri** (*str*) – the URI to resolve

Returns the retrieved document

1.4 Resource Schemas

doctor.resource.**HTTP_METHOD_TITLES = {'DELETE': 'Delete', 'GET': 'Retrieve', 'POST': 'Creat**
A mapping of HTTP method to title that should be used for it in API documentation.

class doctor.resource.**ResourceAnnotation**(*logic*, *http_method*, *title=None*)
Bases: object

Metadata about the types used for a given request method.

Parameters

- **logic** (Callable) – The logic function for the resource.
- **http_method** (str) – The http method for this resource. One of *DELETE*, *GET*, *POST* or *PUT*.
- **title** (Optional[str]) – The title to be used by the api documentation for this resource.

class doctor.resource.**ResourceSchema**(*schema*, *handle_http=None*, *raise_response_validation_errors=False*, ***kwargs*)
Bases: *doctor.schema.Schema*

This class extends *Schema* with methods for generating HTTP handler functions that automatically parse and validate the request and response objects with a given schema.

_create_request_schema(*params*, *required*)
Create a JSON schema for a request.

Parameters

- **params** (*list*) – A list of keys specifying which definitions from the base schema should be allowed in the request.
- **required** (*list*) – A subset of the params that the requester must specify in the request.

Returns a JSON schema dict

class doctor.resource.**ResourceSchemaAnnotation**(*logic*, *http_method*, *schema*, *request_schema*, *response_schema*, *title=None*)
Bases: object

Metadata about the schema used for a given request method.

An instance of this class is attached to each handler method in a _schema_annotation attribute. It can be used for introspection about the schemas, to generate things like API documentation and hyper schemas from the code.

Parameters

- **logic** (*func*) – Logic function which will handle the request.
- **http_method** (*str*) – The HTTP request method for this request (e.g. GET).

- **schema** (doctor.resource.ResourceSchema) – The resource schema object for this handler.
- **request_schema** (*dict*) – The schema used to validate the request.
- **response_schema** (*dict*) – The schema used to validate the response.
- **title** (*str*) – A short title for the route. e.g. 'Create Foo' might be used for a POST method on a FooListHandler.

classmethod get_annotation(*fn*)

Find the _schema_annotation attribute for the given function.

This will descend through decorators until it finds something that has the attribute. If it doesn't find it anywhere, it will return None.

Parameters **fn** (*func*) – Find the attribute on this function.

Returns an instance of *ResourceSchemaAnnotation* or None.

1.5 Response Module Documentation

class doctor.response.**Response**(*content*, *headers=None*)

Represents a response.

This object contains the response itself along with any additional headers that should be added and returned with the response data. An instance of this class can be returned from a logic function in order to modify response headers.

Parameters

- **content** – The data to be returned with the response.
- **headers** (*dict*) – A dict of response headers to include with the response

1.6 Routing

1.6.1 Module Documentation

class doctor.routing.**HTTPMethod**(*method*, *logic*, *allowed_exceptions=None*, *title=None*)

Bases: object

Represents and HTTP method and it's configuration.

When instantiated the logic attribute will have 3 attributes added to it:

- *_doctor_allowed_exceptions* - A list of excpetions that are allowed to be re-reaised if encountered during a request.
- *_doctor_params* - A *Params* instance.
- *_doctor_signature* - The parsed function Signature.
- *_doctor_title* - The title that should be used in api documentation.

Parameters

- **method** (str) – The HTTP method. One of: (delete, get, post, put).
- **logic** (Callable) – The logic function to be called for the http method.

- **allowed_exceptions** (Optional[List[~T]]) – If specified, these exception classes will be re-raised instead of turning them into 500 errors.
- **title** (Optional[str]) – An optional title for the http method. This will be used when generating api documentation.

class doctor.routing.**Route**(*route*, *methods*, *heading=None*, *base_handler_class=None*, *handler_name=None*)

Bases: object

Represents a route.

Parameters

- **route** (str) – The route path, e.g. *r'^/foo/<int:foo_id>/?$'*
- **methods** (Tuple[*HTTPMethod*]) – A tuple of defined HTTPMethods for the route.
- **heading** (Optional[str]) – An optional heading that this route should be grouped under in the api documentation.
- **base_handler_class** – The base handler class to use.
- **handler_name** (Optional[str]) – The name that should be given to the handler class.

doctor.routing.**create_http_method**(*logic*, *http_method*, *handle_http*)

Create a handler method to be used in a handler class.

Parameters

- **logic** (*callable*) – The underlying function to execute with the parsed and validated parameters.
- **http_method** (*str*) – HTTP method this will handle.
- **handle_http** (Callable) – The HTTP handler function that should be used to wrap the logic functions.

Return type Callable

Returns A handler function.

doctor.routing.**create_routes**(*routes*, *handle_http*, *default_base_handler_class*)

Creates handler routes from the provided routes.

Parameters

- **routes** (Tuple[*HTTPMethod*]) – A tuple containing the route and another tuple with all http methods allowed for the route.
- **handle_http** (Callable) – The HTTP handler function that should be used to wrap the logic functions.
- **default_base_handler_class** (Any) – The default base handler class that should be used.

Return type List[Tuple[str, Any]]

Returns A list of tuples containing the route and generated handler.

doctor.routing.**delete**(*func*, *allowed_exceptions=None*, *title=None*)

Returns a HTTPMethod instance to create a DELETE route.

See *HTTPMethod*

Return type *HTTPMethod*

doctor.routing.**get**(*func*, *allowed_exceptions=None*, *title=None*)
Returns a HTTPMethod instance to create a GET route.

See *HTTPMethod*

Return type *HTTPMethod*

doctor.routing.**get_handler_name**(*route*, *logic*)
Gets the handler name.

Parameters

- **route** (*Route*) – A Route instance.
- **logic** (Callable) – The logic function.

Return type str

Returns A handler class name.

doctor.routing.**post**(*func*, *allowed_exceptions=None*, *title=None*)
Returns a HTTPMethod instance to create a POST route.

See *HTTPMethod*

Return type *HTTPMethod*

doctor.routing.**put**(*func*, *allowed_exceptions=None*, *title=None*)
Returns a HTTPMethod instance to create a PUT route.

See *HTTPMethod*

Return type *HTTPMethod*

1.7 Parsing Helpers

This is a collection of functions used to convert untyped param strings into their appropriate JSON schema types.

doctor.parsers.**_parse_array**(*value*)
Coerce value into an list.

Parameters **value** (*str*) – Value to parse.

Returns list or None if the value is not a JSON array

Raises TypeError or ValueError if value appears to be an array but can't be parsed as JSON.

doctor.parsers.**_parse_boolean**(*value*)
Coerce value into an bool.

Parameters **value** (*str*) – Value to parse.

Returns bool or None if the value is not a boolean string.

doctor.parsers.**_parse_object**(*value*)
Coerce value into a dict.

Parameters **value** (*str*) – Value to parse.

Returns dict or None if the value is not a JSON object

Raises TypeError or ValueError if value appears to be an object but can't be parsed as JSON.

doctor.parsers.**_parse_string**(*value*)
Coerce value into a string.

This is usually a no-op, but if value is a unicode string, it will be encoded as UTF-8 before returning.

Parameters **value** (*str*) – Value to parse.

Returns str

doctor.parsers.**parse_json**(*value*)
Parse a value as JSON.

This is just a wrapper around json.loads which re-raises any errors as a ParseError instead.

Parameters **value** (*str*) – JSON string.

Returns the parsed JSON value

doctor.parsers.**parse_value**(*value*, *allowed_types*, *name='value'*)
Parse a value into one of a number of types.

This function is used to coerce untyped HTTP parameter strings into an appropriate type. It tries to coerce the value into each of the allowed types, and uses the first that evaluates properly.

Because this is coercing a string into multiple, potentially ambiguous, types, it tests things in the order of least ambiguous to most ambiguous:

- The "null" type is checked first. If allowed, and the value is blank (""), None will be returned.
- The "boolean" type is checked next. Values of "true" (case insensitive) are True, and values of "false" are False.
- Numeric types are checked next – first "integer", then "number".
- The "array" type is checked next. A value is only considered a valid array if it begins with a "[" and can be parsed as JSON.
- The "object" type is checked next. A value is only considered a valid object if it begins with a "{" and can be parsed as JSON.
- The "string" type is checked last, since any value is a valid string. Unicode strings are encoded as UTF-8.

Parameters

- **value** (*str*) – Parameter value. Example: "1"
- **allowed_types** (*list*) – Types that should be attempted. Example: ["integer", "null"]
- **name** (*str*) – Parameter name. If not specified, "value" is used. Example: "campaign_id"

Returns a tuple of a type string and coerced value

Raises ParseError if the value cannot be coerced to any of the types

1.8 Error Classes

These error classes should be used in your *Logic Functions* to abstract out the HTTP layer. Doctor provides custom exceptions which will be converted to the correct HTTP Exception by the library. This allows logic functions to be easily reused by other logic in your code base without it having knowledge of the HTTP layer.

exception doctor.errors.**ForbiddenError**
Bases: *doctor.errors.SchematicError*

Raised when a request is forbidden for the authorized user.

Corresponds to a HTTP 403 Forbidden error.

exception doctor.errors.**ImmutableError**
Bases: *doctor.errors.SchematicError*

Raised for immutable errors for a schema.

Corresponds to a HTTP 409 Conflict error.

exception doctor.errors.**InternalError**
Bases: *doctor.errors.SchematicError*

Raised when there is an internal server error.

Corresponds to a HTTP 500 Internal Server Error.

exception doctor.errors.**InvalidValueError**
Bases: *doctor.errors.SchematicError*

Raised for errors when doing more complex validation that can't be done in a schema.

Corresponds to a HTTP 400 Bad Request error.

exception doctor.errors.**NotFoundError**
Bases: *doctor.errors.SchematicError*

Raised when a resource is not found.

Corresponds to a HTTP 404 Not Found error.

exception doctor.errors.**ParseError**
Bases: *doctor.errors.SchematicError*

Raised when a value cannot be parsed into an appropriate type.

exception doctor.errors.**SchemaError**
Bases: *doctor.errors.SchematicError*

Raised for errors in a schema.

exception doctor.errors.**SchemaLoadingError**
Bases: *doctor.errors.SchematicError*

Raised when loading a resource and it is invalid.

exception doctor.errors.**SchemaValidationError**(*message*, *errors=None*)
Bases: *doctor.errors.SchematicError*

Raised for errors when validating things against a schema.

exception doctor.errors.**SchematicError**
Bases: ValueError

Base error class for doctor.

exception doctor.errors.**TypeSystemError**(*detail=None*, *cls=None*, *code=None*, *errors=None*)
Bases: Exception

An error that represents an invalid value for a type.

This is borrowed from apistar: https://github.com/encode/apistar/blob/master/apistar/exceptions.py#L1-L15

Parameters

- **detail** (Union[str, dict, None]) – Detail about the error.
- **cls** (Optional[type]) – The class type that was being instantiated.

- **code** (Optional[str]) – The error code.
- **errors** (Optional[dict]) – A dict containing all validation errors during the request. The key is the param name and the value is the error message.

exception doctor.errors.**UnauthorizedError**

Bases: *doctor.errors.SchematicError*

Raised when a request is unauthorized.

Corresponds to a HTTP 401 Unauthorized error.

1.9 Types

Doctor *types* validate request parameters passed to logic functions. Every request parameter that gets passed to your logic function should define a type from one of those below. See *quick type creation* for functions that allow you to create types easily on the fly.

1.9.1 String

A *String* type represents a *str* and allows you to define several attributes for validation.

Attributes

- *description* - A human readable description of what the type represents. This will be used when generating documentation.
- *example* - An example value to send to the endpoint when generating API documentation. This is optional and a default example value will be generated for you.
- *format* - An identifier indicating a complex datatype with a string representation. For example *date*, to represent an ISO 8601 formatted date string. The following formats are supported:
  - *date* - Will parse the string as a *datetime.datetime* instance. Expects the format *'%Y-%m-%d'*
  - *date-time* - Will parse the string as a *datetime.datetime* instance. Expects a valid ISO8601 string. e.g. *'2018-02-21T16:09:02Z'*
  - *email* - Does basic validation that the string is an email by checking for an *'@'* character in the string.
  - *time* - Will parse the string as a *datetime.datetime* instance. Expects the format *'%H:%M:%S'*
  - *uri* - Will validate the string is a valid URI.
- *max_length* - The maximum length of the string.
- *min_length* - The minimum length of the string.
- *pattern* - A regex pattern the string should match anywhere whitin it. Uses *re.search*.
- *trim_whitespace* - If *True* the string will be trimmed of whitespace.

Example

```
from doctor.types import String

class FirstName(String):
    description = "A user's first name."
    min_length = 1
    max_length = 255
    trim_whitespace = True
```

1.9.2 Number

A *Number* type represents a *float* and allows you to define several attributes for validation.

Attributes

- *description* - A human readable description of what the type represents. This will be used when generating documentation.
- *example* - An example value to send to the endpoint when generating API documentation. This is optional and a default example value will be generated for you.
- *exclusive_maximum*- If *True* and *maximum* is set, the maximum value should be treated as exclusive (value can not be equal to maximum).
- *exclusive_minimum*- If *True* and *minimum* is set, the minimum value should be treated as exclusive (value can not be equal to minimum).
- *maximum* - The maximum value allowed.
- *minimum* - The minimum value allowed.
- *multiple_of* - The value is required to be a multiple of this value.

Example

```
from doctor.types import Number

class AverageRating(Number):
    description = 'The average rating.'
    exclusive_maximum = False
    exclusive_minimum = True
    minimum = 0.00
    maximum = 10.0
```

1.9.3 Integer

An *Integer* type represents an *int* and allows you to define several attributes for validation.

Attributes

- *description* - A human readable description of what the type represents. This will be used when generating documentation.

- *example* - An example value to send to the endpoint when generating API documentation. This is optional and a default example value will be generated for you.
- *exclusive_maximum*- If *True* and the *maximum* is set, the maximum value should be treated as exclusive (value can not be equal to maximum).
- *exclusive_minimum*- If *True* and the *minimum* is set, the minimum value should be treated as exclusive (value can not be equal to minimum).
- *maximum* - The maximum value allowed.
- *minimum* - The minimum value allowed.
- *multiple_of* - The value is required to be a multiple of this value.

Example

```
from doctor.types import Integer

class Age(Integer):
    description = 'The age of the user.'
    exclusive_maximum = False
    exclusive_minimum = True
    minimum = 1
    maximum = 120
```

1.9.4 Boolean

A *Boolean* type represents a *bool*. This type will convert several common strings used as booleans to a boolean type when instaniated. The following *str* values (case-insensitve) will be converted to a boolean:

- *'true'|'false'*
- *'on'|'off'*
- *'1'|'0'*

It also accepts typical truthy inputs e.g. *True*, *False*, *1*, *0*.

Attributes

- *description* - A human readable description of what the type represents. This will be used when generating documentation.
- *example* - An example value to send to the endpoint when generating API documentation. This is optional and a default example value will be generated for you.

Example

```
from doctor.types import Boolean

class Accept(Boolean):
    description = 'Indicates if the user accepted the agreement or not.'
```

1.9.5 Enum

An *Enum* type represents a *str* that should be one of any defined values and allows you to define several attributes for validation.

Attributes

- *description* - A human readable description of what the type represents. This will be used when generating documentation.
- *enum* - A list of *str* containing valid values.
- *example* - An example value to send to the endpoint when generating API documentation. This is optional and a default example value will be generated for you.

Example

```
from doctor.types import Enum

class Color(Enum):
    description = 'A color.'
    enum = ['blue', 'green', 'purple', 'yellow']
```

1.9.6 Object

An *Object* type represents a *dict* and allows you to define properties and required properties.

Attributes

- *additional_properties* - If *True*, additional properties (that is, ones not defined in *properties*) will be allowed.
- *description* - A human readable description of what the type represents. This will be used when generating documentation.
- *example* - An example value to send to the endpoint when generating API documentation. This is optional and a default example value will be generated for you.
- *properties* - A dict containing a mapping of property name to expected type.
- *required* - A list of required properties.

Example

```
from doctor.types import Object, boolean, string

class Contact(Object):
    description = 'An address book contact.'
    additional_properties = True
    properties = {
        'name': string('The contact name', min_length=1, max_length=200),
        'is_primary', boolean('Indicates if this is a primary contact.'),
```

(continues on next page)

(continued from previous page)

```
    }
    required = ['name']
```

1.9.7 Array

An *Array* type represents a *list* and allows you to define properties and required properties.

Attributes

- *additional_items* - If *items* is a list and this is *True* then additional items whose types aren't defined are allowed in the list.
- *description* - A human readable description of what the type represents. This will be used when generating documentation.
- *example* - An example value to send to the endpoint when generating API documentation. This is optional and a default example value will be generated for you.
- *items* - The type each item should be, or a list of types where the position of the type in the list represents the type at that position in the array the item should be.
- *min_items* - The minimum number of items allowed in the list.
- *max_items* - The maximum number of items allowed in the list.
- *unique_items* - If *True*, items in the array should be unique from one another.

Example

```
from doctor.types import Array, string

class Countries(Array):
    description = 'An array of countries.'
    items = string('A country')
    min_items = 0
    max_items = 5
    unique_items = True
```

1.9.8 JsonSchema

A *JsonSchema* type is primarily meant to ease the transition from doctor *2.x.x* to *3.0.0*. It allows you to specify an already defined schema file to represent a type. You can use a definition within the schema as your type or the root type of the schema.

This type will use the json schema to set the description, example and native type attributes of the class. This type should not be used directly, instead you should use *json_schema_type()* to create your class.

Attributes

- *definition_key* - The key of the definition within your schema that should be used for the type.

- *description* - A human readable description of what the type represents. This will be used when generating documentation. This value will automatically get loaded from your schema definition.
- *example* - An example value to send to the endpoint when generating API documentation. This value will automatically get loaded from your schema definition.
- *schema_file* - The full path to the schema file. This attribute is required to be defined on your class.

Example

annotation.yaml

```
---
$schema: 'http://json-schema.org/draft-04/schema#'
description: An annotation.
definitions:
  annotation_id:
    description: Auto-increment ID.
    example: 1
    type: integer
  name:
    description: The name of the annotation.
    example: Example Annotation.
    type: string
type: object
properties:
  annotation_id:
    $ref: '#/definitions/annotation_id'
  name:
    $ref: '#/definitions/name'
additionalProperties: false
```

Using 'definition_key'

```
from doctor.types import json_schema_type

AnnotationId = json_schema_type(
    '/full/path/to/annoation.yaml', definition_key='annotation_id')
```

Without 'definition_key'

```
from doctor.types import json_schema_type

Annotation = json_schema_type('/full/path/to/annotation.yaml')
```

1.9.9 Quick Type Creation

Each type also has a function that can be used to quickly create a new type without having to define large classes. Each of these functions takes the description of the type as the first positional argument and any attributes the type accepts can be passed as keyword arguments. The following functions are provided:

- *array()* - Create a new *Array* type.
- *boolean()* - Create a new *Boolean* type.
- *enum()* - Create a new *Enum* type.
- *integer()* - Create a new *Integer* type.

- *json_schema_type()* - Create a new *JsonSchema* type.
- *new_type()* - Create a new user defined type.
- *number()* - Create a new *Number* type.
- *string()* - Create a new *String* type.

Examples

```
from doctor.errors import TypeSystemError
from doctor.types import (
    array, boolean, enum, integer, json_schema_type, new_type, number,
    string, String)

# Create a new array type of countries
Countries = array('List of countries', items=string('Country'), min_items=1)

# Create a new boolean type
Agreed = boolean('Indicates if user agreed or not')

# Create a new enum type
Color = enum('A color', enum=['blue', 'green', 'red'])

# Create a new integer type
AnnotationId = integer('Annotation PK', minimum=1)

# Create a new jsonschema type
Annotation = json_schema_type(schema_file='/path/to/annotation.yaml')

# Create a new type based on a String
class FooString(String):
    must_start_with_foo = True

    def __new__(cls, *args, **kwargs):
        value = super().__new__(cls, *args, **kwargs)
        if cls.must_start_with_foo:
            if not value.lower().startswith('foo'):
                raise TypeSystemError('Must start with foo', cls=cls)

MyFooString = new_type(FooString, 'My foo string')

# Create a new number type
ProductRating = number('Product rating', maximum=10, minimum=1)

# Create a new string type
FirstName = string('First name', min_length=2, max_length=255)
```

1.9.10 Module Documentation

This file is a modified version of the typingsystem.py module in apistar. https://github.com/encode/apistar/blob/973c6485d8297c1bcef35a42221ac5107dce25d5/apistar/typesystem.py

class doctor.types.**Array**(**args*, ***kwargs*)

Bases: *doctor.types.SuperType*, list

Represents a *list* type.

additional_items = False

If *items* is a list and this is *True* then additional items whose types aren't defined are allowed in the list.

classmethod get_example()

Returns an example value for the Array type.

If an example isn't a defined attribute on the class we return a list of 1 item containing the example value of the *items* attribute. If *items* is None we simply return a *[1]*.

Return type list

items = None

The type each item should be, or a list of types where the position of the type in the list represents the type at that position in the array the item should be.

max_items = None

The maxiimum number of items allowed in the list.

min_items = 0

The minimum number of items allowed in the list.

native_type

alias of builtins.list

unique_items = False

If *True* items in the array should be unique from one another.

class doctor.types.**Boolean**(**args*, ***kwargs*)

Bases: *doctor.types.SuperType*

Represents a *bool* type.

classmethod get_example()

Returns an example value for the Boolean type.

Return type bool

native_type
alias of builtins.bool

class doctor.types.**Enum**(**args*, ***kwargs*)
Bases: *doctor.types.SuperType*, str

Represents a *str* type that must be one of any defined allowed values.

enum = []
A list of valid values.

classmethod get_example()
Returns an example value for the Enum type.

Return type str

native_type
alias of builtins.str

class doctor.types.**Integer**(**args*, ***kwargs*)
Bases: *doctor.types._NumericType*, int

Represents an *int* type.

classmethod get_example()
Returns an example value for the Integer type.

Return type int

native_type
alias of builtins.int

doctor.types.**JSON_TYPES_TO_NATIVE = {'array': <class 'list'>, 'boolean': <class 'bool'>,**
A mapping of json types to native python types.

class doctor.types.**JsonSchema**(**args*, ***kwargs*)
Bases: *doctor.types.SuperType*

Represents a type loaded from a json schema.

NOTE: This class should not be used directly. Instead use *json_schema_type()* to create a new class based on this one.

definition_key = None
The key from the definitions in the schema file that the type should come from.

classmethod get_example()
Returns an example value for the JsonSchema type.

Return type Any

schema = None
The loaded ResourceSchema

schema_file = None
The full path to the schema file.

exception doctor.types.**MissingDescriptionError**
Bases: ValueError

An exception raised when a type is missing a description.

class doctor.types.**Number**(**args*, ***kwargs*)
Bases: *doctor.types._NumericType*, float

Represents a *float* type.

classmethod get_example()
Returns an example value for the Number type.

Return type float

native_type
alias of builtins.float

class doctor.types.**Object**(**args*, ***kwargs*)
Bases: *doctor.types.SuperType*, dict

Represents a *dict* type.

additional_properties = True
If True additional properties will be allowed, otherwise they will not.

classmethod get_example()
Returns an example value for the Dict type.

If an example isn't a defined attribute on the class we return a dict of example values based on each property's annotation.

Return type dict

native_type
alias of builtins.dict

properties = {}
A mapping of property name to expected type.

required = []
A list of required properties.

class doctor.types.**String**(**args*, ***kwargs*)
Bases: *doctor.types.SuperType*, str

Represents a *str* type.

format = None
Will check format of the string for *date*, *date-time*, *email*, *time* and *uri*.

classmethod get_example()
Returns an example value for the String type.

Return type str

max_length = None
The maximum length of the string.

min_length = None
The minimum length of the string.

native_type
alias of builtins.str

pattern = None
A regex pattern that the string should match.

trim_whitespace = True
Whether to trim whitespace on a string. Defaults to *True*.

class doctor.types.**SuperType**(**args*, ***kwargs*)
Bases: object

A super type all custom types must extend from.

This super type requires all subclasses define a description attribute that describes what the type represents. A *ValueError* will be raised if the subclass does not define a *description* attribute.

description = None
The description of what the type represents.

example = None
An example value for the type.

class doctor.types.**_NumericType**(**args*, ***kwargs*)
Bases: *doctor.types.SuperType*

Base class for both *Number* and *Integer*.

exclusive_maximum = False
The maximum value should be treated as exclusive or not.

exclusive_minimum = False
The minimum value should be treated as exclusive or not.

maximum = None
The maximum value allowed.

minimum = None
The minimum value allowed.

multiple_of = None
The value is required to be a multiple of this value.

doctor.types.**array**(*description*, ***kwargs*)
Create a *Array* type.

Parameters

- **description** – A description of the type.
- **kwargs** – Can include any attribute defined in *Array*

Return type Type[+CT]

doctor.types.**boolean**(*description*, ***kwargs*)
Create a *Boolean* type.

Parameters

- **description** – A description of the type.
- **kwargs** – Can include any attribute defined in *Boolean*

Return type Type[+CT]

doctor.types.**enum**(*description*, ***kwargs*)
Create a *Enum* type.

Parameters

- **description** – A description of the type.
- **kwargs** – Can include any attribute defined in *Enum*

Return type Type[+CT]

doctor.types.**get_value_from_schema**(*schema*, *definition*, *key*, *definition_key*, *resolve=False*)
Gets a value from a schema and definition.

Parameters

- **schema** (ResourceSchema) – The resource schema.
- **definition** (*dict*) – The definition dict from the schema.
- **key** (*str*) – The key to use to get the value from the schema.
- **definition_key** (*str*) – The name of the definition.
- **resolve** (*bool*) – If True we will attempt to resolve the definition from the schema.

Returns The value.

Raises ***TypeSystemError*** – If the key can't be found in the schema/definition or we can't resolve the definition.

doctor.types.**integer**(*description*, ***kwargs*)
Create a *Integer* type.

Parameters

- **description** – A description of the type.
- **kwargs** – Can include any attribute defined in *Integer*

Return type Type[+CT]

doctor.types.**json_schema_type**(*schema_file*, ***kwargs*)
Create a *JsonSchema* type.

This function will automatically load the schema and set it as an attribute of the class along with the description and example.

Parameters

- **schema_file** (str) – The full path to the json schema file to load.
- **kwargs** – Can include any attribute defined in *JsonSchema*

Return type Type[+CT]

doctor.types.**new_type**(*cls*, *description*, ***kwargs*)
Create a user defined type.

Parameters

- **description** – A description of the type.
- **kwargs** – Can include any attribute defined in the provided user defined type.

Return type Type[+CT]

doctor.types.**number**(*description*, ***kwargs*)
Create a *Number* type.

Parameters

- **description** – A description of the type.
- **kwargs** – Can include any attribute defined in *Number*

Return type Type[+CT]

doctor.types.**string**(*description*, ***kwargs*)
Create a *String* type.

Parameters

- **description** (str) – A description of the type.

- **kwargs** – Can include any attribute defined in *String*

Return type Type[+CT]

1.10 Utils

1.10.1 Module Documentation

doctor.utils.**DESCRIPTION_END_RE = re.compile(':(arg|param|returns|throws)', re.IGNORECASE)**
Used to identify the end of the description block, and the beginning of the parameters. This assumes that the parameters and such will always occur at the end of the docstring.

class doctor.utils.**Params**(*all*, *required*, *optional*, *logic*)
Bases: object

Represents parameters for a request.

Parameters

- **all** (List[str]) – A list of all paramter names for a request.
- **required** (List[str]) – A list of all required parameter names for a request.
- **optional** (List[str]) – A list of all optional parameter names for a request.
- **logic** (List[str]) – A list of all parameter names that are part ofthe logic function signature.

class doctor.utils.**RequestParamAnnotation**(*name*, *annotation*, *required=False*)
Bases: object

Represents a new request parameter annotation.

Parameters

- **name** (str) – The name of the parameter.
- **annotation** (A doctor type that should subclass *SuperType*.) – The annotation type of the parameter.
- **required** (bool) – Indicates if the parameter is required or not.

doctor.utils.**add_param_annotations**(*logic*, *params*)
Adds parameter annotations to a logic function.

This adds additional required and/or optional parameters to the logic function that are not part of it's signature. It's intended to be used by decorators decorating logic functions or middleware.

Parameters

- **logic** (Callable) – The logic function to add the parameter annotations to.
- **params** (List[*RequestParamAnnotation*]) – The list of RequestParamAnnotations to add to the logic func.

Return type Callable

Returns The logic func with updated parameter annotations.

doctor.utils.**get_description_lines**(*docstring*)
Extract the description from the given docstring.

This grabs everything up to the first occurrence of something that looks like a parameter description. The docstring will be dedented and cleaned up using the standard Sphinx methods.

Parameters **docstring** (*str*) – The source docstring.

Returns list

doctor.utils.**get_module_attr**(*module_filename*, *module_attr*, *namespace=None*)

Get an attribute from a module.

This uses exec to load the module with a private namespace, and then plucks and returns the given attribute from that module's namespace.

Note that, while this method doesn't have any explicit unit tests, it is tested implicitly by the doctor's own documentation. The Sphinx build process will fail to generate docs if this does not work.

Parameters

- **module_filename** (*str*) – Path to the module to execute (e.g. "../src/app.py").
- **module_attr** (*str*) – Attribute to pluck from the module's namespace. (e.g. "app").
- **namespace** (*dict*) – Optional namespace. If one is not passed, an empty dict will be used instead. Note that this function mutates the passed namespace, so you can inspect a passed dict after calling this method to see how the module changed it.

Returns The attribute from the module.

Raises **KeyError** – if the module doesn't have the given attribute.

doctor.utils.**get_params_from_func**(*func*, *signature=None*)

Gets all parameters from a function signature.

Parameters

- **func** (Callable) – The function to inspect.
- **signature** (Optional[Signature]) – An inspect.Signature instance.

Return type *Params*

Returns A named tuple containing information about all, optional, required and logic function parameters.

doctor.utils.**get_valid_class_name**(*s*)

Return the given string converted so that it can be used for a class name

Remove leading and trailing spaces; removes spaces and capitalizes each word; and remove anything that is not alphanumeric. Returns a pep8 compatible class name.

Parameters **s** (str) – The string to convert.

Return type str

Returns The updated string.

CHAPTER 2

Release History

2.1 Next release (in development)

2.2 v3.0.0-beta.5 (2018-03-05)

- Fixed doctor attempting to document non doctor type params (#70)
- String with format of date now returns datetime.date (#69)
- Fixed swallowing of TypeError from SuperType class in Object init (#68)
- Changed the flask code to only raise response validation errors if an environment variable is set. Before it also raised them when DEBUG was True in the config. In practice this was incredibly annoying and slowed down development. Especially in the case where a datetime string was returned that didn't include timezone information. Updated the docs to reflect this too.
- Fixed issue that could create duplicate handler names which would cause an exception in flask restful (#67)
- Made the *JsonSchema* doctor type work in validating/coercing params in the api and for generating api documentation.

2.3 v3.0.0-beta.4 (2018-03-02)

- Made validation errors better when raising http 400 exceptions. They now will display all missing required fields and all validation errors along with have the param in the error message.
- Fixed issue with doctor types being passed to logic functions. Instead the native types are now passed to prevent downstream issues from other code encountering unexpected/unknown types.

2.4 v3.0.0-beta.3 (2018-02-28)

- Added default example values for all doctor types.
- Documentation updates
- Updated doctor code to work agnostic of the framework so eventually other backends than flask could be used.

2.5 V3.0.0-beta (2018-02-27)

- First beta release of 3.0. This is a backwards incompatible change. It drops support for python 2 and defining request parameters through the usage of json schemas. It's still possible to use the json schemas from previous versions of doctor to generate new doctor types using doctor.types.json_schema_type. See the documentation for more information.

2.6 v1.3.5 (2018-01-23)

- Fixed a few deprecation warnings about inspect.getargspec when running doctor using python 3. It will now use inspect.getfullargspec. This also fixes the issue of not being able to use type hints on logic functions in python 3.

2.7 v1.3.4 (2017-12-04)

- Removed set operation on decorators when applying them to the logic function. Since set types don't have an explicit order it caused unpredicatable behavior as the decorators weren't always applied to the logic function in the same order with every call.

2.8 v1.3.3 (2017-10-18)

- Add request option to router HTTP method dictionary, which allows you to override the schema used to validate the request body.

2.9 v1.3.2 (2017-09-18)

- Fixed response validation when the response was an instance of doctor.response.Response

2.10 v1.3.1 (2017-08-29)

- Fixed bug when auto generating documentation for GET endpoints that contained a parameter that was an array or object. It wasn't getting json dumped, so when the request was made to generate the example response it would get a 400 error.
- Fixed a few typos and bugs in the README quick start example.

2.11 v1.3.0 (2017-08-11)

- Added a Response class that can be returned from logic functions in order to add/modify response headers.

2.12 v1.2.2 (2017-07-10)

- More fixes for Python 3.

2.13 v1.2.1 (2017-07-07)

- Fixed sphinx build error encountered on Sphinx v1.6.1+ when checking if the http domain has already been added.

2.14 v1.2.0 (2017-07-07)

- Added support for Python 3.

2.15 v1.1.4 (2017-05-04)

- Updates doctor to not parse json bodies on GET/DELETE requests, and instead try to parse them from the query string or form parameters.
- Fixes a bug introducded in v1.1.3. This bug would only occur if a logic function was decorated and that decorator passed a positional argument to the logic function. Doctor would think the positional argument passed by the decorator was a required request parameter even if it was specified to be omitted in the router using omit_args.

2.16 v1.1.3 (2017-04-28)

- Added new InternalError class to represent non-doctor internal errors.
- Updated sphinx pin version to be minimum 1.5.4 and added new *env* kwarg to make_field amd make_xref.
- Fixed bug where extra parameters passed on json requests would cause a *TypeError* if the logic function used a decorator.
- Made sure to make decorators a set when applying them to a logic function when creating routes. This is to prevent a decorator from wrapping a function twice if it's defined at the logic level and handler level when creating routes.

2.17 v1.1.2 (2017-02-27)

- Fixes a bug where the logic function wouldn't be undecorated properly.

2.18 v1.1.1 (2017-02-27)

- Made logic function exceptions always raise when applicaiton is in debug mode.
- Updated error message to be clearer when a logic function raises an exception.

2.19 v1.1.0 (2017-02-20)

- Added ability to override the schema used for an individual endpoint.

2.20 v1.0.1 (2017-02-17)

- Making required changes to setup.py for pypi.

2.21 v1.0.0 (2017-02-16)

- Initial release.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

doctor.docs.base, 17
doctor.docs.flask, 22
doctor.errors, 29
doctor.flask, 10
doctor.parsers, 28
doctor.resource, 25
doctor.response, 26
doctor.routing, 26
doctor.schema, 23
doctor.types, 37
doctor.utils, 43

/

```
GET /,7
```

/note

```
GET /note/,9
GET /note/(int:note_id)/,8
POST /note/,7
PUT /note/(int:note_id)/,9
DELETE /note/(int:note_id)/,8
```

Index

Symbols

_NumericType (class in doctor.types), 41
_create_request_schema() (doctor.resource.ResourceSchema method), 25
_format_stack() (doctor.schema.SchemaRefResolver method), 24
_get_annotation_heading() (doctor.docs.base.BaseHarness method), 18
_get_example_values() (doctor.docs.base.BaseHarness method), 18
_get_headers() (doctor.docs.base.BaseHarness method), 18
_parse_array() (in module doctor.parsers), 28
_parse_boolean() (in module doctor.parsers), 28
_parse_object() (in module doctor.parsers), 28
_parse_string() (in module doctor.parsers), 28
_prepare_env() (doctor.docs.base.BaseDirective method), 17
_render_rst() (doctor.docs.base.BaseDirective method), 17

A

add_param_annotations() (in module doctor.utils), 43
additional_items (doctor.types.Array attribute), 38
additional_properties (doctor.types.Object attribute), 40
Array (class in doctor.types), 38
array() (in module doctor.types), 41
AutoFlaskDirective (class in doctor.docs.flask), 22
AutoFlaskHarness (class in doctor.docs.flask), 22

B

BaseDirective (class in doctor.docs.base), 17
BaseHarness (class in doctor.docs.base), 18
Boolean (class in doctor.types), 38
boolean() (in module doctor.types), 41

C

CAMEL_CASE_RE (in module doctor.docs.base), 20
create_http_method() (in module doctor.routing), 27
create_routes() (in module doctor.flask), 10
create_routes() (in module doctor.routing), 27

D

define_example_values() (doctor.docs.base.BaseHarness method), 18
define_header_values() (doctor.docs.base.BaseHarness method), 19
defined_header_values (doctor.docs.base.BaseHarness attribute), 19
definition_key (doctor.types.JsonSchema attribute), 39
delete() (in module doctor.routing), 27
description (doctor.types.SuperType attribute), 41
DESCRIPTION_END_RE (in module doctor.utils), 43
directive_name (doctor.docs.base.BaseDirective attribute), 17
DirectiveState (class in doctor.docs.base), 20
doctor.docs.base (module), 17
doctor.docs.flask (module), 22
doctor.errors (module), 29
doctor.flask (module), 10
doctor.parsers (module), 28
doctor.resource (module), 25
doctor.response (module), 26
doctor.routing (module), 26
doctor.schema (module), 23
doctor.types (module), 37
doctor.utils (module), 43

E

Enum (class in doctor.types), 39
enum (doctor.types.Enum attribute), 39
enum() (in module doctor.types), 41
example (doctor.types.SuperType attribute), 41
exclusive_maximum (doctor.types._NumericType attribute), 41
exclusive_minimum (doctor.types._NumericType attribute), 41

F

ForbiddenError, 29
format (doctor.types.String attribute), 40
from_file() (doctor.schema.Schema class method), 23

G

get() (in module doctor.routing), 27
get_annotation() (doctor.resource.ResourceSchemaAnnotation class method), 26
get_description_lines() (in module doctor.utils), 43
get_example() (doctor.types.Array class method), 38
get_example() (doctor.types.Boolean class method), 38
get_example() (doctor.types.Enum class method), 39
get_example() (doctor.types.Integer class method), 39
get_example() (doctor.types.JsonSchema class method), 39
get_example() (doctor.types.Number class method), 39
get_example() (doctor.types.Object class method), 40
get_example() (doctor.types.String class method), 40
get_example_curl_lines() (in module doctor.docs.base), 20
get_example_lines() (in module doctor.docs.base), 20
get_handler_name() (in module doctor.routing), 28
get_json_lines() (in module doctor.docs.base), 21
get_json_object_lines() (in module doctor.docs.base), 21
get_module_attr() (in module doctor.utils), 44
get_name() (in module doctor.docs.base), 21
get_outdated_docs() (doctor.docs.base.BaseDirective class method), 17
get_params_from_func() (in module doctor.utils), 44
get_valid_class_name() (in module doctor.utils), 44
get_validator() (doctor.schema.Schema method), 23
get_value_from_schema() (in module doctor.types), 41

H

handle_http() (in module doctor.flask), 10
harness (doctor.docs.base.BaseDirective attribute), 17
has_content (doctor.docs.base.BaseDirective attribute), 17
header_definitions (doctor.docs.base.BaseHarness attribute), 19
headers (doctor.docs.base.BaseHarness attribute), 19
HTTP400Exception, 10
HTTP401Exception, 10
HTTP403Exception, 10
HTTP404Exception, 10
HTTP409Exception, 10
HTTP500Exception, 10
HTTP_METHOD_TITLES (in module doctor.resource), 25
HTTP_METHODS (in module doctor.docs.base), 20
HTTPMethod (class in doctor.routing), 26

I

ImmutableError, 30
Integer (class in doctor.types), 39
integer() (in module doctor.types), 42
InternalError, 30
InvalidValueError, 30
items (doctor.types.Array attribute), 38
iter_annotations() (doctor.docs.base.BaseHarness method), 19
iter_annotations() (doctor.docs.flask.AutoFlaskHarness method), 22

J

json_schema_type() (in module doctor.types), 42
JSON_TYPES_TO_NATIVE (in module doctor.types), 39
JsonSchema (class in doctor.types), 39

M

max_items (doctor.types.Array attribute), 38
max_length (doctor.types.String attribute), 40
maximum (doctor.types._NumericType attribute), 41
min_items (doctor.types.Array attribute), 38
min_length (doctor.types.String attribute), 40
minimum (doctor.types._NumericType attribute), 41
MissingDescriptionError, 39
multiple_of (doctor.types._NumericType attribute), 41

N

native_type (doctor.types.Array attribute), 38
native_type (doctor.types.Boolean attribute), 38
native_type (doctor.types.Enum attribute), 39
native_type (doctor.types.Integer attribute), 39
native_type (doctor.types.Number attribute), 40
native_type (doctor.types.Object attribute), 40
native_type (doctor.types.String attribute), 40
new_type() (in module doctor.types), 42
normalize_route() (in module doctor.docs.base), 21
NotFoundError, 30
Number (class in doctor.types), 39
number() (in module doctor.types), 42

O

Object (class in doctor.types), 40

P

Params (class in doctor.utils), 43
parse_json() (in module doctor.parsers), 29
parse_value() (in module doctor.parsers), 29
ParseError, 30
pattern (doctor.types.String attribute), 40
post() (in module doctor.routing), 28
prefix_lines() (in module doctor.docs.base), 22

properties (doctor.types.Object attribute), 40
purge_docs() (doctor.docs.base.BaseDirective class method), 17
put() (in module doctor.routing), 28

R

request() (doctor.docs.base.BaseHarness method), 19
request() (doctor.docs.flask.AutoFlaskHarness method), 22
RequestParamAnnotation (class in doctor.utils), 43
required (doctor.types.Object attribute), 40
resolve() (doctor.schema.Schema method), 23
resolve() (doctor.schema.SchemaRefResolver method), 24
resolve_remote() (doctor.schema.SchemaRefResolver method), 24
resolver (doctor.schema.Schema attribute), 23
ResourceAnnotation (class in doctor.resource), 25
ResourceSchema (class in doctor.resource), 25
ResourceSchemaAnnotation (class in doctor.resource), 25
Response (class in doctor.response), 26
Route (class in doctor.routing), 27
run() (doctor.docs.base.BaseDirective method), 17

S

Schema (class in doctor.schema), 23
schema (doctor.types.JsonSchema attribute), 39
schema_file (doctor.types.JsonSchema attribute), 39
SchemaError, 30
SchemaLoadingError, 30
SchemaRefResolver (class in doctor.schema), 24
SchematicError, 30
SchematicHTTPException, 10
SchemaValidationError, 30
setup() (doctor.docs.base.BaseDirective class method), 18
setup() (in module doctor.docs.flask), 23
setup_app() (doctor.docs.base.BaseHarness method), 19
setup_app() (doctor.docs.flask.AutoFlaskHarness method), 23
setup_request() (doctor.docs.base.BaseHarness method), 19
should_raise_response_validation_errors() (in module doctor.flask), 11
String (class in doctor.types), 40
string() (in module doctor.types), 42
SuperType (class in doctor.types), 40

T

teardown_app() (doctor.docs.base.BaseHarness method), 20
teardown_request() (doctor.docs.base.BaseHarness method), 20
trim_whitespace (doctor.types.String attribute), 40
TYPE_MAP (in module doctor.docs.base), 20
TypeSystemError, 30

U

UnauthorizedError, 31
unique_items (doctor.types.Array attribute), 38
URL_PARAMS_RE (in module doctor.docs.base), 20

V

validate() (doctor.schema.Schema method), 23
validate_json() (doctor.schema.Schema method), 24