JsonWeb Documentation

Release 0.8.2

shawn adams

Oct 10, 2018

Contents

Quickly add JSON encoding/decoding to your python objects.

CHAPTER 1

Main documentation

To get the best understanding of JsonWeb you should read the documentation in order. As each section builds a little bit on the last.

1.1 `jsonweb.encode` – encode your python classes

Often times in a web application the data you wish to return to users is described by some sort of data model or resource in the form of a class object. This module provides an easy way to encode your python class instances to JSON. Here is a quick example:

```
>>> from jsonweb.encode import to_object, dumper
>>> @to_object()
... class DataModel(object):
...       def __init__(self, id, value):
...           self.id = id
...           self.value = value

>>> data = DataModel(5, "foo")
>>> dumper(data)
'{"__type__": "DataModel", "id": 5, "value": "foo"}'
```

If you have a class you wish to serialize to a JSON object decorate it with *`to_object()`*. If your class should serialize into a JSON list decorate it with *`to_list()`*.

1.1.1 dumper

`jsonweb.encode.`**`dumper`**(*obj*, ***kw*)

JSON encode your class instances by calling this function as you would call `json.dumps()`. `kw` args will be passed to the underlying json.dumps call.

Parameters

- **`handlers`** – A dict of type name/handler callable to use. ie {“Person:” person_handler}

- **cls** – To override the given encoder. Should be a subclass of *JsonWebEncoder*.
- **suppress** – A list of extra fields to suppress (as well as those suppressed by the class).
- **exclude_nulls** – Set True to suppress keys with null (None) values from the JSON output. Defaults to False.

1.1.2 Decorators

jsonweb.encode.**to_object**(*cls_type=None*, *suppress=None*, *handler=None*, *exclude_nulls=False*)

To make your class instances JSON encodable decorate them with *to_object()*. The python built-in dir() is called on the class instance to retrieve key/value pairs that will make up the JSON object (*Minus any attributes that start with an underscore or any attributes that were specified via the* suppress *keyword argument*).

Here is an example:

```
>>> from jsonweb import to_object
>>> @to_object()
... class Person(object):
...     def __init__(self, first_name, last_name):
...         self.first_name = first_name
...         self.last_name = last_name

>>> person = Person("Shawn", "Adams")
>>> dumper(person)
'{"__type__": "Person", "first_name": "Shawn", "last_name": "Adams"}'
```

A __type__ key is automatically added to the JSON object. Its value should represent the object type being encoded. By default it is set to the value of the decorated class's __name__ attribute. You can specify your own value with cls_type:

```
>>> from jsonweb import to_object
>>> @to_object(cls_type="PersonObject")
... class Person(object):
...     def __init__(self, first_name, last_name):
...         self.first_name = first_name
...         self.last_name = last_name

>>> person = Person("Shawn", "Adams")
>>> dumper(person)
'{"__type__": "PersonObject", "first_name": "Shawn", "last_name": "Adams"}'
```

If you would like to leave some attributes out of the resulting JSON simply use the suppress kw argument to pass a list of attribute names:

```
>>> from jsonweb import to_object
>>> @to_object(suppress=["last_name"])
... class Person(object):
...     def __init__(self, first_name, last_name):
...         self.first_name = first_name
...         self.last_name = last_name

>>> person = Person("Shawn", "Adams")
>>> dumper(person)
'{"__type__": "Person", "first_name": "Shawn"}'
```

You can even suppress the __type__ attribute

```
@to_object(suppress=["last_name", "__type__"])
...
```

Sometimes it's useful to suppress `None` values from your JSON output. Setting `exclude_nulls` to `True` will accomplish this

```
>>> from jsonweb import to_object
>>> @to_object(exclude_nulls=True)
... class Person(object):
...     def __init__(self, first_name, last_name):
...         self.first_name = first_name
...         self.last_name = last_name

>>> person = Person("Shawn", None)
>>> dumper(person)
'{"__type__": "Person", "first_name": "Shawn"}'
```

Note: You can also pass most of these arguments to *`dumper()`*. They will take precedence over what you passed to *`to_object()`* and only effects that one call.

If you need greater control over how your object is encoded you can specify a `handler` callable. It should accept one argument, which is the object to encode, and it should return a dict. This would override the default object handler *`JsonWebEncoder.object_handler()`*.

Here is an example:

```
>>> from jsonweb import to_object
>>> def person_encoder(person):
...     return {"FirstName": person.first_name,
...         "LastName": person.last_name}
...
>>> @to_object(handler=person_encoder)
... class Person(object):
...     def __init__(self, first_name, last_name):
...         self.guid = 12334
...         self.first_name = first_name
...         self.last_name = last_name

>>> person = Person("Shawn", "Adams")
>>> dumper(person)
'{"FirstName": "Shawn", "LastName": "Adams"}'
```

You can also use the alternate decorator syntax to accomplish this. See *`jsonweb.encode.handler()`*.

`jsonweb.encode.`**`to_list`**(*handler=None*)

If your class instances should serialize into a JSON list decorate it with *`to_list()`*. By default The python built in `list` will be called with your class instance as its argument. ie **list(obj)**. This means your class needs to define the `__iter__` method.

Here is an example:

```
@to_object(suppress=["__type__"])
class Person(object):
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name
```

(continues on next page)

(continued from previous page)

```
@to_list()
class People(object):
    def __init__(self, *persons):
        self.persons = persons

    def __iter__(self):
        for p in self.persons:
            yield p

people = People(
    Person("Luke", "Skywalker"),
    Person("Darth", "Vader"),
    Person("Obi-Wan" "Kenobi")
)
```

Encoding `people` produces this JSON:

```
[
    {"first_name": "Luke", "last_name": "Skywalker"},
    {"first_name": "Darth", "last_name": "Vader"},
    {"first_name": "Obi-Wan", "last_name": "Kenobi"}
]
```

New in version 0.6.0: You can now specify a custom handler callable with the `handler` kw argument. It should accept one argument, your class instance. You can also use the *`jsonweb.encode.handler()`* decorator to mark one of the class's methods as the list handler.

`jsonweb.encode.`**`handler`**(*func*)

Use this decorator to mark a method on a class as being its jsonweb encode handler. It will be called any time your class is serialized to a JSON string.

```
>>> from jsonweb import encode
>>> @encode.to_object()
... class Person(object):
...     def __init__(self, first_name, last_name):
...         self.first_name = first_name
...         self.last_name = last_name
...     @encode.handler
...     def to_obj(self):
...         return {"FirstName": person.first_name,
...             "LastName": person.last_name}
...
>>> @encode.to_list()
... class People(object):
...     def __init__(self, *persons):
...         self.persons = persons
...     @encode.handler
...     def to_list(self):
...         return self.persons
...
>>> people = People(
...     Person("Luke", "Skywalker"),
...     Person("Darth", "Vader"),
...     Person("Obi-Wan" "Kenobi")
... )
...
```

(continues on next page)

(continued from previous page)

```
>>> print dumper(people, indent=2)
[
  {
    "FirstName": "Luke",
    "LastName": "Skywalker"
  },
  {
    "FirstName": "Darth",
    "LastName": "Vader"
  },
  {
    "FirstName": "Obi-Wan",
    "LastName": "Kenobi"
  }
]
```

1.1.3 JsonWebEncoder

class jsonweb.encode.**JsonWebEncoder**(***kw*)

This `json.JSONEncoder` subclass is responsible for encoding instances of classes that have been decorated with *`to_object()`* or *`to_list()`*. Pass *`JsonWebEncoder`* as the value for the `cls` keyword argument to `json.dump()` or `json.dumps()`.

Example:

```
json.dumps(obj_instance, cls=JsonWebEncoder)
```

Using *`dumper()`* is a shortcut for the above call to `json.dumps()`

```
dumper(obj_instance) #much nicer!
```

default(*o*)

Implement this method in a subclass such that it returns a serializable object for `o`, or calls the base implementation (to raise a `TypeError`).

For example, to support arbitrary iterators, you could implement default like this:

```
def default(self, o):
    try:
        iterable = iter(o)
    except TypeError:
        pass
    else:
        return list(iterable)
    # Let the base class default method raise the TypeError
    return JSONEncoder.default(self, o)
```

list_handler(*obj*)

Handles encoding instance objects of classes decorated by *`to_list()`*. Simply calls `list` on `obj`.

> **Note:** Override this method if you wish to change how ALL objects are encoded into JSON lists.

object_handler(*obj*)

Handles encoding instance objects of classes decorated by *`to_object()`*. Returns a dict containing all

the key/value pairs in `obj.__dict__`. Excluding attributes that

- start with an underscore.
- were specified with the `suppress` keyword argument.

The returned dict will be encoded into JSON.

Note: Override this method if you wish to change how ALL objects are encoded into JSON objects.

1.2 `jsonweb.decode` – decode your python classes

Sometimes it would be nice to have `json.loads()` return class instances. For example if you do something like this

```
person = json.loads('''
{
    "__type__": "Person",
    "first_name": "Shawn",
    "last_name": "Adams"
}
''')
```

it would be pretty cool if instead of `person` being a `dict` it was an instance of a class we defined called `Person`. Luckily the python standard `json` module provides support for *class hinting* in the form of the `object_hook` keyword argument accepted by `json.loads()`.

The code in *`jsonweb.decode`* uses this `object_hook` interface to accomplish the awesomeness you are about to witness. Lets turn that `person` `dict` into a proper `Person` instance.

```
>>> from jsonweb.decode import from_object, loader

>>> @from_object()
... class Person(object):
...     def __init__(self, first_name, last_name):
...         self.first_name = first_name
...         self.last_name = last_name
...

>>> person = loader('''
... {
...     "__type__": "Person",
...     "first_name": "Shawn",
...     "last_name": "Adams"
... }
... ''')

>>> print type(person)
<class 'Person'>
>>> print person.first_name
"Shawn"
```

But how was `jsonweb` able to determine how to instantiate the `Person` class? Take a look at the *`from_object()`* decorator for a detailed explanation.

1.2.1 loader

jsonweb.decode.**loader**(*json_str*, ***kw*)

Call this function as you would call `json.loads()`. It wraps the *Object hook* interface and returns python class instances from JSON strings.

Parameters

- **ensure_type** – Check that the resulting object is of type `ensure_type`. Raise a ValidationError otherwise.
- **handlers** – is a dict of handlers. see *`object_hook()`*.
- **as_type** – explicitly specify the type of object the JSON represents. see *`object_hook()`*
- **validate** – Set to False to turn off validation (ie dont run the schemas) during this load operation. Defaults to True.
- **kw** – the rest of the kw args will be passed to the underlying `json.loads()` calls.

1.2.2 Decorators

jsonweb.decode.**from_object**(*handler=None*, *type_name=None*, *schema=None*)

Decorating a class with *`from_object()`* will allow `json.loads()` to return instances of that class.

`handler` is a callable that should return your class instance. It receives two arguments, your class and a python dict. Here is an example:

```
>>> from jsonweb.decode import from_object, loader
>>> def person_decoder(cls, obj):
...     return cls(
...         obj["first_name"],
...         obj["last_name"]
...     )
...
>>> @from_object(person_decoder)
... class Person(object):
...     def __init__(self, first_name, last_name):
...         self.first_name
...         self.last_name = last_name
...
>>> person_json = '{"__type__": "Person", "first_name": "Shawn", "last_name":
↪"Adams"}'
>>> person = loader(person_json)
>>> person
<Person object at 0x1007d7550>
>>> person.first_name
'Shawn'
```

The `__type__` key is very important. Without it `jsonweb` would not know which handler to delegate the python dict to. By default *`from_object()`* assumes `__type__` will be the class's `__name__` attribute. You can specify your own value by setting the `type_name` keyword argument

```
@from_object(person_decoder, type_name="PersonObject")
```

Which means the json string would need to be modified to look like this:

```
'{"__type__": "PersonObject", "first_name": "Shawn", "last_name": "Adams"}'
```

If a handler cannot be found for `__type__` an exception is raised

```
>>> luke = loader('{"__type__": "Jedi", "name": "Luke"}')
Traceback (most recent call last):
    ...
ObjectNotFoundError: Cannot decode object Jedi. No such object.
```

You may have noticed that `handler` is optional. If you do not specify a `handler` `jsonweb` will attempt to generate one. It will inspect your class's `__init__` method. Any positional arguments will be considered required while keyword arguments will be optional.

Warning: A handler cannot be generated from a method signature containing only `*args` and `**kwargs`. The handler would not know which keys to pull out of the python dict.

Lets look at a few examples:

```
>>> from jsonweb import from_object
>>> @from_object()
... class Person(object):
...     def __init__(self, first_name, last_name, gender):
...         self.first_name = first_name
...         self.last_name = last_name
...         self.gender = gender

>>> person_json = '{"__type__": "Person", "first_name": "Shawn", "last_name":
↪"Adams", "gender": "male"}'
>>> person = loader(person_json)
```

What happens if we dont want to specify `gender`:

```
>>> person_json = '''{
...     "__type__": "Person",
...     "first_name": "Shawn",
...     "last_name": "Adams"
... }'''
>>> person = loader(person_json)
Traceback (most recent call last):
    ...
ObjectAttributeError: Missing gender attribute for Person.
```

To make `gender` optional it must be a keyword argument:

```
>>> from jsonweb import from_object
>>> @from_object()
... class Person(object):
...     def __init__(self, first_name, last_name, gender=None):
...         self.first_name = first_name
...         self.last_name = last_name
...         self.gender = gender

>>> person_json = '{"__type__": "Person", "first_name": "Shawn", "last_name":
↪"Adams"}'
>>> person = loader(person_json)
```

(continues on next page)

(continued from previous page)

```
>>> print person.gender
None
```

You can specify a json validator for a class with the `schema` keyword agrument. Here is a quick example:

```
>>> from jsonweb import from_object
>>> from jsonweb.schema import ObjectSchema
>>> from jsonweb.validators import ValidationError, String
>>> class PersonSchema(ObjectSchema):
...     first_name = String()
...     last_name = String()
...     gender = String(optional=True)
...
>>> @from_object(schema=PersonSchema)
... class Person(object):
...     def __init__(self, first_name, last_name, gender=None):
...         self.first_name = first_name
...         self.last_name = last_name
...         self.gender = gender
...
>>> person_json = '{"__type__": "Person", "first_name": 12345, "last_name": "Adams
↪"}'
>>> try:
...     person = loader(person_json)
... except ValidationError, e:
...     print e.errors["first_name"].message
Expected str got int instead.
```

Schemas are useful for validating user supplied json in web services or other web applications. For a detailed explanation on using schemas see the *jsonweb.schema*.

1.2.3 Object hook

jsonweb.decode.**object_hook**(*handlers=None*, *as_type=None*, *validate=True*)

Wrapper around *ObjectHook*. Calling this function will configure an instance of *ObjectHook* and return a callable suitable for passing to `json.loads()` as `object_hook`.

If you need to decode a JSON string that does not contain a `__type__` key and you know that the JSON represents a certain object or list of objects you can use `as_type` to specify it

```
>>> json_str = '{"first_name": "bob", "last_name": "smith"}'
>>> loader(json_str, as_type="Person")
<Person object at 0x1007d7550>
>>> # lists work too
>>> json_str = '''[
...     {"first_name": "bob", "last_name": "smith"},
...     {"first_name": "jane", "last_name": "smith"}
... ]'''
>>> loader(json_str, as_type="Person")
[<Person object at 0x1007d7550>, <Person object at 0x1007d7434>]
```

Note: Assumes every object WITHOUT a `__type__` kw is of the type specified by `as_type` .

`handlers` is a dict with this format:

```
{"Person": {"cls": Person, "handler": person_decoder, "schema": PersonSchema)}
```

If you do not wish to decorate your classes with *from_object()* you can specify the same parameters via the `handlers` keyword argument. Here is an example:

```
>>> class Person(object):
...     def __init__(self, first_name, last_name):
...         self.first_name = first_name
...         self.last_name = last_name
...
>>> def person_decoder(cls, obj):
...     return cls(obj["first_name"], obj["last_name"])

>>> handlers = {"Person": {"cls": Person, "handler": person_decoder}}
>>> person = loader(json_str, handlers=handlers)
>>> # Or invoking the object_hook interface ourselves
>>> person = json.loads(json_str, object_hook=object_hook(handlers))
```

Note: If you decorate a class with *from_object()* you can override the `handler` and `schema` values later. Here is an example of overriding a schema you defined with *from_object()* (some code is left out for brevity):

```
>>> from jsonweb import from_object
>>> @from_object(schema=PersonSchema)
>>> class Person(object):
    ...

>>> # and later on in the code...
>>> handlers = {"Person": {"schema": NewPersonSchema}}
>>> person = loader(json_str, handlers=handlers)
```

If you need to use `as_type` or `handlers` many times in your code you can forgo using *loader()* in favor of configuring a "custom" object hook callable. Here is an example

```
>>> my_obj_hook = object_hook(handlers)
>>> # this call uses custom handlers
>>> person = json.loads(json_str, object_hook=my_obj_hook)
>>> # and so does this one ...
>>> another_person = json.loads(json_str, object_hook=my_obj_hook)
```

class jsonweb.decode.**ObjectHook**(*handlers*, *validate=True*)

This class does most of the work in managing the handlers that decode the json into python class instances. You should not need to use this class directly. *object_hook()* is responsible for instantiating and using it.

decode_obj(*obj*)

This method is called for every dict decoded in a json string. The presence of the key `__type__` in `obj` will trigger a lookup in `self.handlers`. If a handler is not found for `__type__` then an `ObjectNotFoundError` is raised. If a handler is found it will be called with `obj` as it only argument. If an `ObjectSchema` was supplied for the class, `obj` will first be validated then passed to handler. The handler should return a new python instant of type `__type__`.

1.2.4 `as_type` context mananger

jsonweb.decode.**ensure_type**(**args*, ***kwds*)

This context manager lets you "inject" a value for `ensure_type` into *`loader()`* calls made in the active context. This will allow a `ValidationError` to bubble up from the underlying *`loader()`* call if the resultant type is not of type `ensure_type`.

Here is an example

```
# example_app.model.py
from jsonweb.decode import from_object
# import db model stuff
from example_app import db

@from_object()
class Person(db.Base):
    first_name = db.Column(String)
    last_name = db.Column(String)

    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name


# example_app.__init__.py
from example_app.model import session, Person
from jsonweb.decode import from_object, ensure_type
from jsonweb.schema import ValidationError
from flask import Flask, request, abort

app.errorhandler(ValidationError)
def json_validation_error(e):
    return json_response({"error": e})


def load_request_json():
    if request.headers.get('content-type') == 'application/json':
        return loader(request.data)
    abort(400)


@app.route("/person", methods=["POST", "PUT"])
def add_person():
    with ensure_type(Person):
        person = load_request_json()
    session.add(person)
    session.commit()
    return "ok"
```

The above example is pretty contrived. We could have just made `load_json_request` accept an `ensure_type` kw, but imagine if the call to *`loader()`* was burried deeper in our api and such a thing was not possible.

Table of Contents

1.3 jsonweb.schema

1.3.1 Declarative

jsonweb.schema provides a layer of validation before `json.decode` returns your object instances. It can also simply be used to validate the resulting python data structures returned from `json.loads()`. It's main use is through a declarative style api. Here is an example of validating the structure of a python dict:

```
>>>
>>> from jsonweb.schema import ObjectSchema, ValidationError
>>> from jsonweb.validators import String

>>> class PersonSchema(ObjectSchema):
...     first_name = String()
...     last_name = String()

>>> try:
...     PersonSchema().validate({"first_name": "shawn"})
... except ValidationError, e:
...     print e.errors
{"last_name": "Missing required parameter."}
```

Validating plain old python data structures is fine, but the more interesting exercise is tying a schema to a class definition:

```
>>> from jsonweb.decode import from_object, loader
>>> from jsonweb.schema import ObjectSchema, ValidationError
>>> from jsonweb.validators import String, Integer, EnsureType

>>> class PersonSchema(ObjectSchema):
...    id = Integer()
...    first_name = String()
...    last_name = String()
...    gender = String(optional=True)
...    job = EnsureType("Job")
```

You can make any field optional by setting `optional` to `True`.

Warning: The field is only optional at the schema level. If you've bound a schema to a class via *from_object()* and the underlying class requires that field a `ObjectAttributeError` will be raised if missing.

As you can see its fine to pass a class name as a string, which we have done for the `Job` class above. We must later define `Job` and decorate it with *from_object()*

```
>>> class JobSchema(ObjectSchema):
...     id = Integer()
...     title = String()

>>> @from_object(schema=JobSchema)
... class Job(object):
...     def __init__(self, id, title):
...         self.id = id
...         self.title = title

>>> @from_object(schema=PersonSchema)
... class Person(object):
...     def __init__(self, first_name, last_name, job, gender=None):
...         self.first_name = first_name
...         self.last_name = last_name
...         self.gender = gender
...         self.job = job
...     def __str__(self):
...         return '<Person name="%s" job="{0}">'.format(
...             " ".join((self.first_name, self.last_name)),
...             self.job.title
...         )

>>> person_json = '''
...     {
...         "__type__": "Person",
...         "id": 1,
...         "first_name": "Bob",
...         "last_name": "Smith",
...         "job": {"__type__": "Job", "id": 5, "title": "Police Officer"},
...     }'''
...

>>> person = loader(person_json)
>>> print person
<Person name="Bob" job="Police Officer">
```

1.3.2 Non-Declarative

New in version 0.8.1.

Use the staticmethod `ObjectSchema.create()` to build object schemas in a non declarative style. Handy for validating dicts with string keys that are not valid python identifiers (e.g "first-name"):

```
MySchema = ObjectSchema.create("MySchema", {
    "first-name": String(),
    "last-name": String(optional=True)
})
```

1.4 jsonweb.validators

class jsonweb.validators.**BaseValidator**(*optional=False*, *nullable=False*, *default=None*, *reason_code=None*)

Abstract base validator which all `JsonWeb` validators should inherit from. Out of the box `JsonWeb` comes

with a dozen or so validators.

All validators will override `BaseValidator._validate()` method which should accept an object and return the passed in object or raise a *ValidationError* if validation failed.

Note: You are not *required* to return the exact passed in object. You may for instance want to transform the object in some way. This is exactly what *DateTime* does.

__init__(*optional=False*, *nullable=False*, *default=None*, *reason_code=None*)
All validators that inherit from `BaseValidator` should pass `optional`, `nullable` and `default` as explicit kw arguments or `**kw`.

Parameters

- **optional** – Is the item optional?
- **nullable** – Can the item's value can be None?
- **default** – A default value for this item.
- **reason_code** – Failure reason_code that is passed to any *ValidationError* raised from this instance.

class jsonweb.validators.**String**(*min_len=None*, *max_len=None*, ***kw*)
Validates something is a string

```
>>> String().validate("foo")
... 'foo'
>>> String().validate(1)
Traceback (most recent call last):
    ...
ValidationError: Expected str got int instead.
```

Specify a maximum length

```
>>> String(max_len=3).validate("foobar")
Traceback (most recent call last):
...
ValidationError: String exceeds max length of 3.
```

Specify a minimum length

```
>>> String(min_len=3).validate("fo")
Traceback (most recent call last):
...
ValidationError: String must be at least length 3.
```

class jsonweb.validators.**Regex**(*regex*, ***kw*)
New in version 0.6.3: Validates a string against a regular expression ::

```
>>> Regex(r"^foo").validate("barfoo")
Traceback (most recent call last):
...
ValidationError: String does not match pattern '^foo'.
```

class jsonweb.validators.**Number**(***kw*)
Validates something is a number

```
>>> Number().validate(1)
... 1
>>> Number().validate(1.1)
>>> 1.1
>>> Number().validate("foo")
Traceback (most recent call last):
    ...
ValidationError: Expected number got int instead.
```

class jsonweb.validators.**Integer**(***kw*)
Validates something in an integer

class jsonweb.validators.**Float**(***kw*)
Validates something is a float

class jsonweb.validators.**Boolean**(***kw*)
Validates something is a Boolean (True/False)

class jsonweb.validators.**DateTime**(*format='%Y-%m-%d %H:%M:%S'*, ***kw*)
Validates that something is a valid date/datetime string and turns it into a `datetime.datetime` instance

```
>>> DateTime().validate("2010-01-02 12:30:00")
... datetime.datetime(2010, 1, 2, 12, 30)

>>> DateTime().validate("2010-01-02 12:300")
Traceback (most recent call last):
    ...
ValidationError: time data '2010-01-02 12:300' does not match format '%Y-%m-%d %H:
↪%M:%S'
```

The default datetime format is `%Y-%m-%d %H:%M:%S`. You can specify your own

```
>>> DateTime("%m/%d/%Y").validate("01/02/2010")
... datetime.datetime(2010, 1, 2, 0, 0)
```

class jsonweb.validators.**EnsureType**(*_type*, *type_name=None*, ***kw*)
Validates something is a certain type

```
>>> class Person(object):
...     pass
>>> EnsureType(Person).validate(Person())
... <Person>
>>> EnsureType(Person).validate(10)
Traceback (most recent call last):
    ...
ValidationError: Expected Person got int instead.
```

class jsonweb.validators.**List**(*validator*, ***kw*)
Validates a list of things. The List constructor accepts a validator and each item in the list will be validated against it

```
>>> List(Integer).validate([1,2,3,4])
... [1,2,3,4]

>>> List(Integer).validate(10)
Traceback (most recent call last):
    ...
ValidationError: Expected list got int instead.
```

Since `ObjectSchema` is also a validator we can do this

```
>>> class PersonSchema(ObjectSchema):
...     first_name = String()
...     last_name = String()
...
>>> List(PersonSchema).validate([
...     {"first_name": "bob", "last_name": "smith"},
...     {"first_name": "jane", "last_name": "smith"}
... ])
```

class jsonweb.validators.**Dict**(*validator*, *key_validator=None*, ***kw*)

New in version 0.8.

Validates a dict of things. The Dict constructor accepts a validator and each value in the dict will be validated against it

```
>>> Dict(Number).validate({'foo': 1})
... {'foo': 1}

>>> Dict(Number).validate({'foo': "abc"})
Traceback (most recent call last):
    ...
ValidationError: Error validating dict.
```

In order see what part of the dict failed validation we must dig deeper into the exception:

```
>>> str(e.errors["foo"])
... 'Expected number got str instead.'
```

`Dict` also accepts an optional `key_validator`, which must be a subclass of *`String`*:

```
validator = Dict(Number, key_validator=Regex(r'^[a-z]{2}_[A-Z]{2}$'))
try:
    validator.validate({"en-US": 1})
except ValidationError as e:
     print(e.errors["en-US"])
     print(e.errors["en-US"].reason_code)

# output
# String does not match pattern '^[a-z]{2}_[A-Z]{2}$'.
# invalid_dict_key
```

class jsonweb.validators.**OneOf**(**values*, ***kw*)

New in version 0.6.4: Validates something is a one of a list of allowed values::

```
>>> OneOf("a", "b", "c").validate(1)
Traceback (most recent call last):
    ...
ValidationError: Expected one of (a, b, c) got 1 instead.
```

class jsonweb.validators.**SubSetOf**(*super_set*, ***kw*)

New in version 0.6.4: Validates a list is subset of another list::

```
>>> SubSetOf([1, 2, 3, 4]).validate([1, 4])
... [1, 4]
>>> SubSetOf([1, 2, 3, 4]).validate([1,5])
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
    ...
ValidationError: [1, 5] is not a subset of [1, 2, 3, 4].
```

1.4.1 ValidationErrors

class jsonweb.validators.**ValidationError**(*reason*, *reason_code=None*, *errors=None*, ***extras*)

Raised from `JsonWeb` validators when validation of an object fails.

__init__(*reason*, *reason_code=None*, *errors=None*, ***extras*)

Parameters

- **reason** – A nice message describing what was not valid
- **reason_code** – programmatic friendly reason code
- **errors** – A `list` or `dict` of nested `ValidationError`
- **extras** – Any extra info about the error you want to convey

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

j

Index

Symbols

B

D

E

F

H

I

J

L

N

O

R

S

T

V