incoming Documentation

Release 0.3.1

Vaidik Kapoor

April 08, 2015

Contents

incoming is a JSON validation framework.

CHAPTER 1

Overview

Validating anything can get really messy. JSON being one of the most used formats for data exchange, **incoming** aims at solving the problem of validating JSON with structure and ease.

incoming is a small framework for validating JSON. Its up to you where and how to use it. A common use-case (and the primary reason why I wrote this framework) was using it for writing HTTP servers to validate incoming JSON payload.

1.1 Features

- Classes that can be sub-classed for writing structured validators.
- Basic validators (or *datatypes*) for performing common validations, like string, numbers, booleans, lists, nested JSON, etc.
- Allows extending validators (*datatypes*) to write your own.
- Allows writing callables for validating values.
- Captures errors during validation and returns a complete report of errors.
- Allows reporting different errors for different validation test failures for the same value.

CHAPTER 2

Installation

Installation is simple.

```
python setup.py install
```

CHAPTER 3

Basic Usage

```python
import json

from datetime import date
from incoming import datatypes, PayloadValidator


class MovieValidator(PayloadValidator):

    name = datatypes.String()
    rating = datatypes.Function('validate_rating',
                                error='Rating must be in between 1 and 10.')
    actors = datatypes.Array()
    is_3d = datatypes.Boolean()
    release_year = datatypes.Function('validate_release_year',
                                      error=('Release year must be in between '
                                             '1800 and current year.'))

    # validation method can be a regular method
    def validate_rating(self, val, *args, **kwargs):
        if not isinstance(val, int):
            return False

        if val < 1 or val > 10:
            return False

        return True

    # validation method can be a staticmethod as well
    @staticmethod
    def validate_release_year(val, *args, **kwargs):
        if not isinstance(val, int):
            return False

        if val < 1800 or val > date.today().year:
            return False

        return True

payload = {
    'name': 'Avengers',
    'rating': 5,
    'actors': [
        'Robert Downey Jr.',
        'Samual L. Jackson',
        'Scarlett Johansson',
        'Mark Ruffalo'
    ],
    'is_3d': True,
    'release_year': 2012
}
result, errors = MovieValidator().validate(payload)
assert result and errors is None, 'Validation failed.\n%s' % json.dumps(errors, indent=2)

payload = {
    'name': 'Avengers',
    'rating': 11,
    'actors': [
        'Robert Downey Jr.',
        'Samual L. Jackson',
        'Scarlett Johansson',
        'Mark Ruffalo'
    ],
    'is_3d': 'True',
    'release_year': 2014
}
result, errors = MovieValidator().validate(payload)
assert result and errors is None, 'Validation failed.\n%s' % json.dumps(errors, indent=2)
```

Run the above script, you shall get a response like so:

```
Traceback (most recent call last):
  File "code.py", line 67, in <module>
    assert result and errors is None, 'Validation failed.\n%s' % json.dumps(errors, indent=2)
AssertionError: Validation failed.
{
  "rating": [
    "Rating must be in between 1 and 10."
  ],
  "is_3d": [
    "Invalid data. Expected a boolean value."
  ],
  "release_year": [
    "Release year must be in between 1800 and current year."
  ]
}
```

CHAPTER 4

Tests

Run tests like so:

```
python setup.py test
```

or:

```
py.test incoming
```

CHAPTER 5

User Guide

5.1 Writing Validators

PayloadValidator is the main validator class that must be sub-classed to write validators for validating your JSON. For example:

```
>>> class PersonValidator(PayloadValidator):
...     name = datatypes.String()
...     age = datatypes.Integer()
>>>
>>> payload = dict(name='Man', age=23)
>>> PersonValidator().validate(payload)
(True, None)
```

Every field/key in the payload that you wish to validate must be added as an attribute in the sub-class of incoming.PayloadValidator. The attribute must be initialized and should be a an object of one of the classes of incoming.datatypes. These classes provide validation tests. See the *Available Datatypes*. And see *Creating your own datatypes*.

5.1.1 Validating Nested JSON

Validating nested JSON is similar to how you validate other payloads. Simply, write a validator for each JSON and then use the incoming.datatypes.JSON datatype to validate the nested JSON:

```
>>> class AddressValidator(PayloadValidator):
...      street = datatypes.String()
...      country = datatypes.String()
...
>>> class PersonValidator(PayloadValidator):
...      name = datatypes.String()
...      age = datatypes.Integer()
...      address = datatypes.JSON(AddressValidator)
...
>>> PersonValidator().validate(dict(name='Some name', age=19, address=dict(street='Brannan, SF', cou
(True, None)
>>>
>>> PersonValidator().validate(dict(name='Some name', age=19, address=dict(street='Brannan, SF', cou
(False, {'address': ['Invalid data. Expected JSON.', {'country': ['Invalid data. Expected a string.']
```

If you want to use a nested class instead, just remember to define the inner class before using it, so that the name of the inner class is available in the scope of the parent class:

```
>>> class PersonValidator(PayloadValidator):
...     class AddressValidator(PayloadValidator):
...         street = datatypes.String()
...         country = datatypes.String()
...
...     name = datatypes.String()
...     age = datatypes.Integer()
...     address = datatypes.JSON(AddressValidator)
...
>>> PersonValidator().validate(dict(name='Some name', age=19, address=dict(street='Brannan, SF', cou
(True, None)
>>>
>>> PersonValidator().validate(dict(name='Some name', age=19, address=dict(street='Brannan, SF', cou
(False, {'address': ['Invalid data. Expected JSON.', {'country': ['Invalid data. Expected a string.']
```

5.1.2 Custom error messages for every field

Every `datatype` provides its own default error message (`_DEFAULT_ERROR`) which are very generic and good enough if you are using just those datatypes and not validation functions. However, you would mostly want to have your own error messages for every field according to your application's requirements.

`incoming` allows you to specify different error messages for every field so that whenever validation fails, these error messages are used instead of the default error messages.

For example:

```
def validate_age(val, *args, **kwargs):
    if not isinstance(val, int):
        return False
    if val < 0:
        return False
    return True

class PersonValidator(PayloadValidator):
    required = False

    name = datatypes.String(required=True, error='Name must be a string.')
    age = datatypes.Function(validate_age,
                         error='Age can never be negative.')
    hobbies = datatypes.Array(error='Please provide hobbies in an array.')
```

5.1.3 Custom validation methods

`incoming` lets you write custom validation methods for validating complex cases as well. These functions must return a `bool`, `True` if validation passes, else `False`.

A few things to keep in mind

- A validation function can be any callable. The callable should be passed to `incoming.datatypes.Function` as an argument.
- If the callable is a regular method or `classmethod` or `staticmethod` on on the validator class, then pass just the name of the method as a string.

- Validation functions (callables) must return `bool` values only. `True` shall be passed when the validation test passes, `False` otherwise.
- Validation callables get the following arguments: * `val` - the value which must be validated * `key` - the field/key in the payload that held `val`. * `payload` - the entire payload that is being validated. * `errors` - `incoming.incoming.PayloadErrors` object.
- For the above point mentioned above, you may want to use your validation methods elsewhere outside the scope of `incoming` where the above arguments are not necessary. In that case, use `*args` and `**kwargs`.

Writing validation functions

Write your validation functions anywhere and pass `incoming.datatypes.Function` the function you have written for validation.

```
def validate_age(val, *args, **kwargs):
    if not isinstance(val, int):
        return False
    if val < 0:
        return False
    return True

class PersonValidator(PayloadValidator):
    required = False
    name = datatypes.String(required=True)
    age = datatypes.Function(validate_age)
    hobbies = datatypes.Array()
```

Validation functions bound by other classes

If you would like to organize your validation functions by keeping them in a different class for some sort of organization that you prefer, you can do that like so:

```
class Validations(object):
    # You can write staticmethods
    @staticmethod
    def validate_age(val, *args, **kwargs):
        if not isinstance(val, int):
            return False
        if val < 0:
            return False
        return True

    # You can write classmethods as well
    @classmethod
    def validate_hobbies(val, *args, **kwargs):
        if not isinstance(val, list):
            return False
        return True

class PersonValidator(PayloadValidator):
    required = False

    name = datatypes.String(required=True)
    age = datatypes.Function(Validations.validate_age)
    hobbies = datatypes.Function(Validations.validate_hobbies)
```

Validation methods in `incoming.PayloadValidator` sub-classes

For the sake of organization, it would probably make more sense to have all the validation methods in your validator classes. You can do that like so:

```
class PersonValidator(PayloadValidator):
    required = True

    name = datatypes.String()

    # Note that validation method's name is provided as an str value
    age = datatypes.Function('validate_age')
    hobbies = datatypes.Function('validate_hobbies')
    sex = datatypes.Function('validate_sex')

    # You can write regular methods for validation
    def validate_age(self, val, *args, **kwargs):
        if not isinstance(val, int):
            return False
        if val < 0:
            return False
        return True

    # You can write staticmethods for validation
    @staticmethod
    def validate_hobbies(val, *args, **kwargs):
        if not isinstance(val, list):
            return False
        return True

    # You can write classmethods for validation
    @classmethod
    def validate_sex(cls, val, *wargs, **kwargs):
        if val not in ('male', 'female'):
            return False
        return True
```

5.1.4 Specifying required fields

`required` can be set on all the fields or on particular fields as well. All fields are required by default. You can explicitly specify this like so:

```
>>> class PersonValidator(PayloadValidator):
...     required = False
...
...     name = datatypes.String(required=True)
...     age = datatypes.Integer(required=True)
...     hobbies = datatypes.Array()
>>>
>>> payload = dict(name='Man', age=23)
>>> PersonValidator().validate(payload)
(True, None)
```

Depending upon your use-case, you may have more required fields than fields that are not required and vice-versa. `incoming` can help you with that. The above example can be written this way as well:

```
>>> class PersonValidator(PayloadValidator):
...     required = True
...
...     name = datatypes.String())
...     age = datatypes.Integer()
...     hobbies = datatypes.Array(required=False)
>>>
>>> payload = dict(name='Man', age=23)
>>> PersonValidator().validate(payload)
(True, None)
```

Overriding `required` at the time of validation

There can be some cases in which you might want to override the `required` setting defined in the validator class at the time of validating the payload.

This can be done like so:

```
>>> class PersonValidator(PayloadValidator):
...     required = False
...
...     name = datatypes.String(required=True)
...     age = datatypes.Integer(required=True)
...     hobbies = datatypes.Array()
>>>
>>> payload = dict(name='Man', age=23)
>>> PersonValidator().validate(payload, required=True)
(False, {'hobbies': ['Expecting a value for this field.']})
```

Error messages for `required` fields

Check `incoming.PayloadValidator.required_error` for the default error message for `required` fields i.e. when required fields are missing in the payload.

Override this error message like so:

```
>>> class PersonValidator(PayloadValidator):
...     required = False
...     required_error = 'A value for this field must be provided.'
...
...     name = datatypes.String(required=True)
...     age = datatypes.Integer(required=True)
...     hobbies = datatypes.Array()
```

5.1.5 `strict` Mode

`strict` mode, when turned on, makes sure that no extra key/field in the payload is provided. If any extra key/field is provided, validation fails and reports in the errors.

Example:

```
>>> class PersonValidator(PayloadValidator):
...     strict = True
...
...     name = datatypes.String()
...     age = datatypes.Integer()
...     hobbies = datatypes.Array(required=False)
>>>
>>> payload = dict(name='Man', age=23, extra=0)
>>> PersonValidator().validate(payload)
(False, {'extra': ['Unexpected field.']})
```

Note: `strict` mode is turned **off** by default.

Overriding `strict` at the time of validation

There can be some cases in which you might want to override the `strict` setting defined in the validator class at the time of validating the payload.

This can be done like so:

```
>>> class PersonValidator(PayloadValidator):
...     strict = True
...
...     name = datatypes.String()
...     age = datatypes.Integer()
...     hobbies = datatypes.Array(required=False)
>>>
>>> payload = dict(name='Man', age=23, extra=0)
>>> PersonValidator().validate(payload, strict=False)
(True, None)
```

Error messages for `strict` fields

Check `incoming.PayloadValidator.strict_error` for the default error message for `strict` mode i.e. when `strict` mode is on and when extra fields are present in the payload.

Override this error message like so:

```
>>> class PersonValidator(PayloadValidator):
...     strict = True
...     strict_error = 'This field is not allowed.'
...
...     name = datatypes.String()
...     age = datatypes.Integer()
...     hobbies = datatypes.Array(required=False)
```

5.1.6 PayloadValidator Class

class `incoming.`**`PayloadValidator`**(**args*, ***kwargs*)

Main validator class that must be sub-classed to define the schema for validating incoming payload.

`required` **= True**

If all fields/keys are required by default When required is set in the `incoming.PayloadValidator` sub-class, then all the fields are required by default. In case a field must not be required, it should be set at the field/key level.

Note: this attribute can be overridden in the sub-class.

`required_error` = **'Expecting a value for this field.'**
Error message used for reporting when a required field is missing

Note: this attribute can be overridden in the sub-class.

`strict` = **False**
`strict` mode is off by default. `strict` mode makes sure that any field that is not defined in the schema, validation result would fail and the extra key would be reported in errors.

Note: this attribute can be overridden in the sub-class.

`strict_error` = **'Unexpected field.'**
The error message that will be used when `strict` mode is on and an extra field is present in the payload.

Note: this attribute can be overridden in the sub-class.

`validate`(*payload*, *required=None*, *strict=None*)
Validates a given JSON payload according to the rules defiined for all the fields/keys in the sub-class.

Parameters

- **payload** (*dict*) – deserialized JSON object.
- **required** (*bool*) – if every field/key is required and must be present in the payload.
- **strict** (*bool*) – if `validate()` should detect and report any fields/keys that are present in the payload but not defined in the sub-class.

Returns a tuple of two items. First item is a `bool` indicating if the payload was successfully validated and the second item is `None`. If the payload was not valid, then then the second item is a `dict` of errors.

5.1.7 PayloadErrors Class

class `incoming.incoming.`**`PayloadErrors`**
PayloadErrors holds errors detected by `PayloadValidator` and provides helper methods for working with errors. An instance of this class maintains a dictionary of errors where the keys are supposed to be the type of error and the value of the keys are `list` objects that hold error strings.

Ideally, the keys should be name of the field/key in the payload and the value should be a `list` object that holds errors related to that field.

`has_errors`()
Checks if any errors were added to an object of this class.

Returns bool True if has errors, else False.

`to_dict`()
Return a `dict` of errors with keys as the type of error and values as a list of errors.

Returns dict a dictionary of errors

5.2 Datatypes

Incoming provides basic datatypes for validation. These datatypes are responsible for performing validation tests on values. Incoming provides some datatypes and it also allows writing your own datatypes.

Note: also see using *Custom validation methods* for implementing custom validations.

5.2.1 Available Datatypes

Following datatypes are provided:

class `incoming.datatypes.`**`Integer`**(*required=None*, *error=None*, **args*, ***kwargs*)

Sub-class of `Types` class for Integer type. Validates if a value is a `int` value.

Parameters

- **required** (*bool*) – if a particular (this) field is required or not. This param allows specifying field level setting if a particular field is required or not.
- **error** (*str*) – a generic error message that will be used by `incoming.PayloadValidator` when the validation test fails.

class `incoming.datatypes.`**`Float`**(*required=None*, *error=None*, **args*, ***kwargs*)

Sub-class of `Types` class for Float type. Validates if a value is a `float` value.

Parameters

- **required** (*bool*) – if a particular (this) field is required or not. This param allows specifying field level setting if a particular field is required or not.
- **error** (*str*) – a generic error message that will be used by `incoming.PayloadValidator` when the validation test fails.

class `incoming.datatypes.`**`Number`**(*required=None*, *error=None*, **args*, ***kwargs*)

Sub-class of `Types` class for Number type. Validates if a value is a `int` value or `float` value.

Parameters

- **required** (*bool*) – if a particular (this) field is required or not. This param allows specifying field level setting if a particular field is required or not.
- **error** (*str*) – a generic error message that will be used by `incoming.PayloadValidator` when the validation test fails.

class `incoming.datatypes.`**`String`**(*required=None*, *error=None*, **args*, ***kwargs*)

Sub-class of `Types` class for String type. Validates if a value is a `str` value or `unicode` value.

Parameters

- **required** (*bool*) – if a particular (this) field is required or not. This param allows specifying field level setting if a particular field is required or not.
- **error** (*str*) – a generic error message that will be used by `incoming.PayloadValidator` when the validation test fails.

class `incoming.datatypes.`**`Boolean`**(*required=None*, *error=None*, **args*, ***kwargs*)

Sub-class of `Types` class for Boolean type. Validates if a value is a `bool`.

Parameters

- **required** (*bool*) – if a particular (this) field is required or not. This param allows specifying field level setting if a particular field is required or not.
- **error** (*str*) – a generic error message that will be used by `incoming.PayloadValidator` when the validation test fails.

class `incoming.datatypes.`**`Array`**(*required=None*, *error=None*, **args*, ***kwargs*)
Sub-class of `Types` class for Array type. Validates if a value is a `list` object.

Parameters

- **required** (*bool*) – if a particular (this) field is required or not. This param allows specifying field level setting if a particular field is required or not.
- **error** (*str*) – a generic error message that will be used by `incoming.PayloadValidator` when the validation test fails.

class `incoming.datatypes.`**`Function`**(*func*, **args*, ***kwargs*)
Sub-class of `Types` class for Function type. This type allows using functions for validation. Using `incoming.datatypes.Function`, validation tests can be written in a function or a regular method, a `staticmethod` or a `classmethod` on the sub-class of `incoming.PayloadValidator`.

Parameters **func** – any callable that accepts `val`, `*args` and

`**kwawrgs` and returns a `bool` value, True if `val` validates, False otherwise.

class `incoming.datatypes.`**`JSON`**(*cls*, **args*, ***kwargs*)
Sub-class of `Types` class for JSON type. This type allows writing validation for nested JSON.

Parameters **cls** – sub-class of `incoming.PayloadValidator` for

validating nested JSON. If you are using a class nested within the parent validator, you must define the inner class before using it, so that the name of the inner class is defined in the scope of the parent class.

5.2.2 Creating your own datatypes

You can create your own set of datatypes if you like.

Note: also see using *Custom validation methods* for implementing custom validations.

A few things to keep in mind

- Sub-class `incoming.datatypes.Types` to implement a custom datatype.
- The sub-class must add `_DEFAULT_ERROR` attribute to provide the default error message that will be used when validation fails.
- The sub-class must implement `validate()` method as a regular method or as a `staticmethod` or as a `classmethod`.
- `validate()` must return a `bool` value. `True` if validation passes, `False` otherwise.
- `validate()` method gets the following arguments: * `val` - the value which must be validated * `key` - the field/key in the payload that held `val`. * `payload` - the entire payload that is being validated. * `errors` - `incoming.incoming.PayloadErrors` object.
- For the above point mentioned above, you may want to use your `validate()` method elsewhere outside the scope of incoming where the above arguments are not necessary. In that case, use `*args` and `**kwargs`.

Example

The example below shows how you can use a `staticmethod` to implement validation test.

```
class Gender(datatypes.Types):
    _DEFAULT_ERROR = 'Gender must be either male or female.'

    @staticmethod
    def validate(val, *args, **kwargs):
        if not isinstance(val, str):
            return False
        if val.lower() not in ('male', 'female'):
            return False
        return True

class PersonValidator(PayloadValidator):
    name = datatypes.String()
    age = datatypes.Integer()
    gender = Gender()
```

5.2.3 datatypes.Types Base Class

class `incoming.datatypes.`**`Types`**(*required=None*, *error=None*, **args*, ***kwargs*)

Base class for creating new datatypes for validation. When this class is sub-classed, `validate()` must be implemented in the sub-class and this method will be resposible for the actual validation.

Parameters

- **required** (*bool*) – if a particular (this) field is required or not. This param allows specifying field level setting if a particular field is required or not.
- **error** (*str*) – a generic error message that will be used by `incoming.PayloadValidator` when the validation test fails.

`validate`(*val*, **args*, ***kwargs*)

This method must be overridden by the sub-class for implementing the validation test. The overridden method can be a regular method or `classmethod` or `staticmethod`. This method must only return a `bool` value.

Parameters **val** – the value that has to be validated.

Returns **bool** if the implemented validation test passed or not.

CHAPTER 6

Indices and tables

- *genindex*
- *modindex*
- *search*