JSL

Release 0.1.0

May 12, 2015

Contents

JSL is a DSL for describing JSON schemas.

Its code is open source and available at GitHub.

CHAPTER 1

Tutorial

For an overview of JSL's basic functionality, please see the Overview and Tutorial.

1.1 Overview and Tutorial

Welcome to JSL!

This document is a brief tour of JSL's features and a quick guide to its use. Additional documentation can be found in the API documentation.

1.1.1 Introduction

JSON Schema is a JSON-based format to define the structure of JSON data for validation and documentation.

JSL is a Python library that provides a DSL for describing JSON schemas.

Why inventing a DSL?

- A JSON schema in terms of the Python language is a dictionary. A JSON schema of a more or less complex data structure is a dictionary which most likely contains a lot of nested dictionaries of dictionaries of dictionaries. Writing and maintaining the readability of such a dictionary are not very rewarding tasks. They require typing a lot of quotes, braces, colons and commas and carefully indenting everything.
- The JSON schema standard is not always intuitive. It takes a little bit of practice to remember where to use the `maxItems` keyword and where the `maxLength`, or not to forget to set `additionalProperties` to false, and so on.
- The syntax is not very concise. The signal-to-noise ratio increases rapidly with the complexity of the schema, which makes large schemas difficult to read.

JSL is created to address there issues. It allows you to define JSON schemas as if they were ORM models – using classes and fields and relying on the deep metaclass magic under the hood.

Such an approach makes writing and reading schemas easier. It encourages the decomposition of large schemas into smaller readable pieces and makes schemas extendable using class inheritance. It enables the autocompleme feature or IDEs and makes any mistype in a JSON schema keyword cause a RuntimeError.

1.1.2 Quick Example

```
import jsl

class Entry(jsl.Document):
    name = jsl.StringField(required=True)

class File(Entry):
    content = jsl.StringField(required=True)

class Directory(Entry):
    content = jsl.ArrayField(jsl.OneOfField([
        jsl.DocumentField(File, as_ref=True),
        jsl.DocumentField(jsl.RECURSIVE_REFERENCE_CONSTANT)
    ]), required=True)
```

`Directory.get_schema(ordered=True)` returns the following schema:

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "definitions": {
        "directory": {
            "type": "object",
            "properties": {
                "name": {"type": "string"},
                "content": {
                    "type": "array",
                    "items": {
                        "oneOf": [
                            {"$ref": "#/definitions/file"},
                            {"$ref": "#/definitions/directory"}
                        ]
                    }
                }
            },
            "required": ["name", "content"],
            "additionalProperties": false
        },
        "file": {
            "type": "object",
            "properties": {
                "name": {"type": "string"},
                "content": {"type": "string"}
            },
            "required": ["name", "content"],
            "additionalProperties": false
        }
    },
    "$ref": "#/definitions/directory"
}
```

1.1.3 Main Features

JSL introduces the notion of a *document* and provides a set of *fields*.

The schema of a document is always `{"type": "object"}`, whose `properties` contain the schemas of the fields of the document. A document may be thought of as a *`DictField`* with some special abilities. A document

is a class, thus it has a name, by which it can be referenced from another document and either inlined or included using the `{"$ref": "..."}` syntax (see *DocumentField* and its `as_ref` parameter). Also documents can be recursive.

The most useful method of *Document* and the fields is *Document.get_schema()*.

Fields and their parameters are named correspondingly to the keywords described in the JSON Schema standard. So getting started with JSL will be easy for those familiar with the standard.

1.1.4 Variables and Scopes

Suppose there is an application that provides a JSON RESTful API backed by MongoDB. Let's describe a `User` data model:

```
class User(jsl.Document):
    id = jsl.StringField(required=True)
    login = jsl.StringField(required=True, min_length=3, max_length=20)
```

`User.get_schema(ordered=True)` produces the following schema:

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "type": "object",
    "additionalProperties": false,
    "properties": {
        "id": {"type": "string"},
        "login": {
            "type": "string",
            "minLength": 3,
            "maxLength": 20
        }
    },
    "required": ["id", "login"]
}
```

It describes a response of the imaginary `/users/<login>/` endpoint and perhaps a database document structure (if the application stores users "as is").

Let's now describe a structure of the data required to create a new user (i.e., a JSON-payload of `POST`-requests to the imaginary `/users/` endpoint). The data may and may not contain `id`; if `id` is not present, it will be generated by the application:

```
class UserCreationRequest(jsl.Document):
    id = jsl.StringField()
    login = jsl.StringField(required=True, min_length=3, max_length=20)
```

The only difference between `User` and `UserCreationRequest` is whether the `"id"` field is required or not.

JSL provides means not to repeat ourselves.

Using Variables

Let's start with describing *variables*. Variables are objects which value depends on a given role. Which value must be used for which role is determined by a list of rules. A rule is a pair of a matcher and a value. A matcher is a callable that returns `True` or `False` (or a string or an iterable that will be converted to a lambda). Here's what it may look like:

```
>>> var = jsl.Var([
...     # the same as (lambda r: r == 'role_1', 'A')
...     ('role_1', 'A'),
...     # the same as (lambda r: r in ('role_2', 'role_3'), 'A')
...     (('role_2', 'role_3'), 'B'),
...     (lambda r: r.startswith('bad_role_'), 'C'),
... ], default='D')
>>> var.resolve('role_1')
Resolution(value='A', role='role_1')
>>> var.resolve('role_2')
Resolution(value='B', role='role_2')
>>> var.resolve('bad_role_1')
Resolution(value='C', role='bad_role_1')
>>> var.resolve('qwerty')
Resolution(value='D', role='qwerty')
```

Variables can be used instead of regular values almost everywhere in JSL – e.g., they can be added to documents, passed as arguments to *fields* or even used as properties of a *DictField*.

Let's introduce a couple of **roles** for our `User` document:

```
# to describe structures of POST requests
REQUEST_ROLE = 'request'
# to describe structures of responses
RESPONSE_ROLE = 'response'
# to desribe structures of database documents
DB_ROLE = 'db'
```

And describe `User` and `UserCreationRequest` using a single document and a variable:

```
true_if_not_request = jsl.Var({
    jsl.not_(REQUEST_ROLE): True
})

class User(jsl.Document):
    id = jsl.StringField(required=true_if_not_request)
    login = jsl.StringField(required=True, min_length=3, max_length=20)
```

Then the `role` argument can be specified for the *Document.get_schema()* method:

```
User.get_schema(ordered=True, role=REQUEST_ROLE)
```

The resulting schema:

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "type": "object",
    "additionalProperties": false,
    "properties": {
        "id": {"type": "string"},
        "login": {
            "type": "string",
            "minLength": 3,
            "maxLength": 20
        }
    },
    "required": ["login"]
}
```

Using Scopes

Let's add a `version` field to the `User` document with the following requirements in mind: it is stored in the database, but must not appear neither in the request nor the response (a reason for this can be that HTTP headers such as `ETag` and `If-Match` are used for concurrency control).

One way is to turn the `version` field into a variable that only resolves to the field when the current role is `DB_ROLE` and resolves to `None` otherwise:

```
class User(jsl.Document):
    id = jsl.StringField(required=true_if_not_request)
    login = jsl.StringField(required=True, min_length=3, max_length=20)
    version = jsl.Var({
        DB_ROLE: jsl.StringField(required=True)
    })
```

Another (and more preferable) way is to use *scopes*:

```
class User(jsl.Document):
    id = jsl.StringField(required=true_if_not_request)
    login = jsl.StringField(required=True, min_length=3, max_length=20)

    with jsl.Scope(DB_ROLE) as db_scope:
        db_scope.version = jsl.StringField(required=True)
```

A scope is a set of *fields* and a matcher. A scope can be added to a document, and if the matcher of a scope returns `True`, its fields will be present in the resulting schema.

A document may contain arbitrary number of scopes:

```
class Message(jsl.Document):
    created_at = jsl.IntField(required=True)
    content = jsl.StringField(required=True)

class User(jsl.Document):
    id = jsl.StringField(required=true_if_not_request)
    login = jsl.StringField(required=True, min_length=3, max_length=20)

    with jsl.Scope(jsl.not_(REQUEST_ROLE)) as full_scope:
        # a new user can not have messages
        full_scope.messages = jsl.ArrayField(
            jsl.DocumentField(Message), required=True)

    with jsl.Scope(DB_ROLE) as db_scope:
        db_scope.version = jsl.StringField(required=True)
```

Now `User.get_schema(ordered=True, role=DB_ROLE)` returns the following schema:

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "type": "object",
    "additionalProperties": false,
    "properties": {
        "id": {"type": "string"},
        "login": {
            "type": "string",
            "minLength": 3,
            "maxLength": 20
        },
        "messages": {
```

```
            "type": "array",
            "items": {
                "type": "object",
                "additionalProperties": false,
                "properties": {
                    "created_at": {
                        "type": "integer"
                    },
                    "content": {
                        "type": "string"
                    }
                },
                "required": ["created_at", "content"]
            }
        },
        "version": {"type": "string"}
    },
    "required": ["id", "login", "messages", "version"]
}
```

1.1.5 More Examples

A JSON schema from the official documentation defined using JSL:

```
class DiskDevice(jsl.Document):
    type = jsl.StringField(enum=['disk'], required=True)
    device = jsl.StringField(pattern='^/dev/[^/]+(/[^/]+)*$', required=True)

class DiskUUID(jsl.Document):
    type = jsl.StringField(enum=['disk'], required=True)
    label = jsl.StringField(pattern='^[a-fA-F0-9]{8}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-'
                                    '[a-fA-F0-9]{4}-[a-fA-F0-9]{12}$',
                            required=True)

class NFS(jsl.Document):
    type = jsl.StringField(enum=['nfs'], required=True)
    remotePath = jsl.StringField(pattern='^(/[^/]+)+$', required=True)
    server = jsl.OneOfField([
        jsl.StringField(format='ipv4'),
        jsl.StringField(format='ipv6'),
        jsl.StringField(format='host-name'),
    ], required=True)

class TmpFS(jsl.Document):
    type = jsl.StringField(enum=['tmpfs'], required=True)
    sizeInMb = jsl.IntField(minimum=16, maximum=512, required=True)

class FSTabEntry(jsl.Document):
    class Options(object):
        description = 'schema for an fstab entry'

    storage = jsl.OneOfField([
        jsl.DocumentField(DiskDevice, as_ref=True),
        jsl.DocumentField(DiskUUID, as_ref=True),
        jsl.DocumentField(NFS, as_ref=True),
        jsl.DocumentField(TmpFS, as_ref=True),
    ], required=True)
```

```
    fstype = jsl.StringField(enum=['ext3', 'ext4', 'btrfs'])
    options = jsl.ArrayField(jsl.StringField(), min_items=1, unique_items=True)
    readonly = jsl.BooleanField()
```

CHAPTER 2

API Documentation

2.1 Document

class jsl.document.**Options**(*additional_properties=False*, *pattern_properties=None*, *min_properties=None*, *max_properties=None*, *title=None*, *description=None*, *default=None*, *enum=None*, *id=''*, *schema_uri='http://json-schema.org/draft-04/schema#'*, *definition_id=None*, *roles_to_propagate=None*)

A container for options.

All the arguments are the same and work exactly as for *fields.DictField* except properties (since it is automatically populated with the document fields) and these:

Parameters

- **definition_id** (*str*) – A unique string to be used as a key for this document in the "definitions" schema section. If not specified, will be generated from module and class names.
- **schema_uri** (*str*) – An URI of the JSON Schema meta-schema.
- **roles_to_propagate** (*callable, string or iterable*) – A matcher. If it returns True for a role, it will be passed to nested documents.

class jsl.document.**Document**

A document. Can be thought as a kind of *fields.DictField*, which properties are defined by the fields and scopes added to the document class.

It can be tuned using special Options attribute (see *Options* for available settings):

```
class User(Document):
    class Options(object):
        title = 'User'
        description = 'A person who uses a computer or network service.'
    login = StringField(required=True)
```

classmethod is_recursive(*role='default'*)

Returns True if there is a *DocumentField*-references cycle that contains cls.

Parameters **role** (*str*) – A current role.

classmethod get_definition_id()

Returns a unique string to be used as a key for this document in the "definitions" schema section.

classmethod resolve_field(*field*, *role='default'*)

Resolves a field with the name field using role.

Raises `AttributeError`

classmethod resolve_and_iter_fields(*role='default'*)
The same as *iter_fields()*, but *resolvables* are resolved using `role`.

classmethod resolve_and_walk(*role='default'*, *through_document_fields=False*, *visited_documents=frozenset([])*)
The same as *walk()*, but *resolvables* are resolved using `role`.

classmethod iter_fields()
Iterates over the fields of the document, resolving its *resolvables* to all possible values.

classmethod walk(*through_document_fields=False*, *visited_documents=frozenset([])*)
Iterates recursively over the fields of the document, resolving occurring *resolvables* to their all possible values.

Visits fields in a DFS order.

Parameters

- **through_document_fields** (*bool*) – If `True`, walks through nested *DocumentField* fields.
- **visited_documents** (*set*) – Keeps track of visited *documents* to avoid infinite recursion when `through_document_field` is `True`.

Returns iterable of *BaseField*

classmethod get_schema(*role='default'*, *ordered=False*)
Returns a JSON schema (draft v4) of the document.

Parameters

- **role** (*str*) – A role.
- **ordered** (*bool*) – If `True`, the resulting schema dictionary is ordered. Fields are listed in the order they are added to the class. Schema properties are also ordered in a sensible and consistent way, making the schema more human-readable.

Raises *SchemaGenerationException*

Return type dict or OrderedDict

classmethod get_definitions_and_schema(*role='default'*, *res_scope=ResolutionScope(base=, current=, output=)*, *ordered=False*, *ref_documents=None*)
Returns a tuple of two elements.

The second element is a JSON schema of the document, and the first is a dictionary that contains definitions that are referenced from the schema.

Parameters

- **role** (*str*) – A role.
- **ordered** (*bool*) – If `True`, the resulting schema dictionary is ordered. Fields are listed in the order they are added to the class. Schema properties are also ordered in a sensible and consistent way, making the schema more human-readable.
- **res_scope** (*ResolutionScope*) – The current resolution scope.
- **ref_documents** (*set*) – If subclass of *Document* is in this set, all *DocumentField* s pointing to it will be resolved as a reference: `{"$ref": "#/definitions/..."}`. Note: resulting definitions will not contain schema for this document.

Raises *SchemaGenerationException*

Return type (dict, dict or OrderedDict)

class jsl.document.**DocumentMeta**
A metaclass for *Document*. It's responsible for collecting options, fields and scopes registering the document in the registry, making it the owner of nested *document fields* s and so on.

options_container
A class to be used by *create_options()*. Must be a subclass of *Options*.

alias of *Options*

classmethod collect_fields(*mcs*, *bases*, *attrs*)
Collects fields from the current class and its parent classes.

Return type a dictionary mapping field names to fields

classmethod collect_options(*mcs*, *bases*, *attrs*)
Collects options from the current class and its parent classes.

Returns a dictionary of options

classmethod create_options(*options*)
Wraps `options` into a container class (see *options_container*).

Parameters **options** – a dictionary of options

Returns an instance of *options_container*

2.2 Fields

2.2.1 Primitive Fields

class jsl.fields.**NullField**(*id=''*, *default=None*, *enum=None*, *title=None*, *description=None*, ***kwargs*)
A null field.

class jsl.fields.**BooleanField**(*id=''*, *default=None*, *enum=None*, *title=None*, *description=None*, ***kwargs*)
A boolean field.

class jsl.fields.**NumberField**(*multiple_of=None*, *minimum=None*, *maximum=None*, *exclusive_minimum=None*, *exclusive_maximum=None*, ***kwargs*)
A number field.

Parameters

- **multiple_of** (number or *Resolvable*) – A value must be a multiple of this factor.
- **minimum** (number or *Resolvable*) – A minimum allowed value.
- **exclusive_minimum** (bool or *Resolvable*) – Whether a value is allowed to exactly equal the minimum.
- **maximum** (number or *Resolvable*) – A maximum allowed value.
- **exclusive_maximum** (bool or *Resolvable*) – Whether a value is allowed to exactly equal the maximum.

multiple_of = None

minimum = None

exclusive_minimum = None

maximum = None

exclusive_maximum = None

class jsl.fields.**IntField**(*multiple_of=None*, *minimum=None*, *maximum=None*, *exclusive_minimum=None*, *exclusive_maximum=None*, ***kwargs*)
Bases: jsl.fields.primitive.NumberField

An integer field.

class jsl.fields.**StringField**(*pattern=None*, *format=None*, *min_length=None*, *max_length=None*, ***kwargs*)
A string field.

Parameters

- **pattern** (string or *Resolvable*) – A regular expression (ECMA 262) that a string value must match.
- **format** (string or *Resolvable*) – A semantic format of the string (for example, "date-time", "email", or "uri").
- **min_length** (int or *Resolvable*) – A minimum length.
- **max_length** (int or *Resolvable*) – A maximum length.

pattern = None

format = None

min_length = None

max_length = None

class jsl.fields.**EmailField**(*pattern=None*, *format=None*, *min_length=None*, *max_length=None*, ***kwargs*)
Bases: jsl.fields.primitive.StringField

An email field.

class jsl.fields.**IPv4Field**(*pattern=None*, *format=None*, *min_length=None*, *max_length=None*, ***kwargs*)
Bases: jsl.fields.primitive.StringField

An IPv4 field.

class jsl.fields.**DateTimeField**(*pattern=None*, *format=None*, *min_length=None*, *max_length=None*, ***kwargs*)
Bases: jsl.fields.primitive.StringField

An ISO 8601 formatted date-time field.

class jsl.fields.**UriField**(*pattern=None*, *format=None*, *min_length=None*, *max_length=None*, ***kwargs*)
Bases: jsl.fields.primitive.StringField

A URI field.

2.2.2 Compound Fields

jsl.fields.**RECURSIVE_REFERENCE_CONSTANT**
A special value to be used as an argument to create a recursive *DocumentField*.

class jsl.fields.**DocumentField**(*document_cls*, *as_ref=False*, ***kwargs*)
A reference to a nested document.

Parameters

- **document_cls** – A string (dot-separated path to document class, i.e. `"app.resources.User"`), *RECURSIVE_REFERENCE_CONSTANT* or a *Document* subclass.
- **as_ref** (*bool*) – If `True`, the schema of `document_cls` is placed into the definitions dictionary, and the field schema just references to it: `{"$ref": "#/definitions/..."}`. It may make a resulting schema more readable.

owner_cls = None
A *Document* this field is attached to.

as_ref = None

document_cls
A *Document* this field points to.

class jsl.fields.**ArrayField**(*items=None*, *additional_items=None*, *min_items=None*, *max_items=None*, *unique_items=None*, ***kwargs*)
An array field.

Parameters

- **items** – Either of the following:
  - *BaseField* – all items of the array must match the field schema;
  - a list or a tuple of *fields* – all items of the array must be valid according to the field schema at the corresponding index (tuple typing);
  - a *Resolvable* resolving to either of the first two options.
- **min_items** (int or *Resolvable*) – A minimum length of an array.
- **max_items** (int or *Resolvable*) – A maximum length of an array.
- **unique_items** (bool or *Resolvable*) – Whether all the values in the array must be distinct.
- **additional_items** (bool or *BaseField* or *Resolvable*) – If the value of `items` is a list or a tuple, and the array length is larger than the number of fields in `items`, then the additional items are described by the *BaseField* passed using this argument.

items = None

min_items = None

max_items = None

unique_items = None

additional_items = None

class jsl.fields.**DictField**(*properties=None*, *pattern_properties=None*, *additional_properties=None*, *min_properties=None*, *max_properties=None*, ***kwargs*)
A dictionary field.

Parameters

- **properties** (dict[str -> *BaseField* or *Resolvable*]) – A dictionary containing fields.

- **pattern_properties** (dict[str -> *BaseField* or *Resolvable*]) – A dictionary whose keys are regular expressions (ECMA 262). Properties match against these regular expressions, and for any that match, the property is described by the corresponding field schema.
- **additional_properties** (bool or *BaseField* or *Resolvable*) – Describes properties that are not described by the `properties` or `pattern_properties`.
- **min_properties** (int or *Resolvable*) – A minimum number of properties.
- **max_properties** (int or *Resolvable*) – A maximum number of properties

properties = None

pattern_properties = None

additional_properties = None

min_properties = None

max_properties = None

class jsl.fields.**NotField**(*field*, ***kwargs*)

Parameters **field** (*BaseField* or *Resolvable*) – A field to negate.

field = None

class jsl.fields.**OneOfField**(*fields*, ***kwargs*)

Parameters **fields** (list[*BaseField* or *Resolvable*]) – A list of fields, exactly one of which describes the data.

fields = None

class jsl.fields.**AnyOfField**(*fields*, ***kwargs*)

Parameters **fields** (list[*BaseField* or *Resolvable*]) – A list of fields, at least one of which describes the data.

fields = None

class jsl.fields.**AllOfField**(*fields*, ***kwargs*)

Parameters **fields** (list[*BaseField* or *Resolvable*]) – A list of fields, all of which describe the data.

fields = None

2.2.3 Base Classes

class jsl.fields.**BaseField**(*required=False*)

A base class for fields of *documents*. Instances of this class may be added to a document to define its properties.

Implements the *Resolvable* interface.

Parameters **required** (bool or *Resolvable*) – If the field is required. Defaults to `False`.

required = None

Whether the field is required.

resolve(*role*)

Implements the *Resolvable* interface.

Always returns a `Resolution(self, role)`.

Return type *Resolution*

iter_possible_values()
Implements the *Resolvable* interface.

Yields a single value – `self`.

get_definitions_and_schema(*role='default'*, *res_scope=ResolutionScope(base=*, *current=*, *output=)*, *ordered=False*, *ref_documents=None*)
Returns a tuple of two elements.

The second element is a JSON schema of the data described by this field, and the first is a dictionary that contains definitions that are referenced from the schema.

Parameters

- **role** (*str*) – A role.
- **ordered** (*bool*) – If `True`, the resulting schema dictionary is ordered. Fields are listed in the order they are added to the class. Schema properties are also ordered in a sensible and consistent way, making the schema more human-readable.
- **res_scope** (*ResolutionScope*) – The current resolution scope.
- **ref_documents** (*set*) – If subclass of `Document` is in this set, all *DocumentField*s pointing to it will be resolved to a reference: `{"$ref": "#/definitions/..."}`. Note: resulting definitions will not contain schema for this document.

Raises *SchemaGenerationException*

Return type (dict, dict or OrderedDict)

get_schema(*ordered=False*, *role='default'*)
Returns a JSON schema (draft v4) of the field.

Parameters

- **role** (*str*) – A role.
- **ordered** (*bool*) – If `True`, the resulting schema dictionary is ordered. Fields are listed in the order they are added to the class. Schema properties are also ordered in a sensible and consistent way, making the schema more human-readable.

Raises *SchemaGenerationException*

Return type dict or OrderedDict

resolve_attr(*attr*, *role='default'*)
Resolves an attribure with the name `field` using `role`.

If the value of `attr` is *resolvable*, it resolves it using a given `role` and returns the result. Otherwise it returns the raw value and `role` unchanged.

Raises `AttributeError`

Return type *Resolution*

class `jsl.fields.`**BaseSchemaField**(*id=''*, *default=None*, *enum=None*, *title=None*, *description=None*, ***kwargs*)
A base class for fields that directly map to JSON Schema validator.

Parameters

- **required** (bool or *Resolvable*) – If the field is required. Defaults to `False`.
- **id** (*str*) – A string to be used as a value of the "id" keyword of the resulting schema.

- **default** (any JSON-representable object, a callable or a *Resolvable*) – The default value for this field. May be a callable.
- **enum** (list, tuple, set, callable or *Resolvable*) – A list of valid choices. May be a callable.
- **title** (str or *Resolvable*) – A short explanation about the purpose of the data described by this field.
- **description** (str or *Resolvable*) – A detailed explanation about the purpose of the data described by this field.

id = None
A string to be used as a value of the "id" keyword of the resulting schema.

title = None
A short explanation about the purpose of the data.

description = None
A detailed explanation about the purpose of the data.

get_enum(*role='default'*)
Returns a list to be used as a value of the `"enum"` schema keyword.

get_default(*role='default'*)
Returns a value of the `"default"` schema keyword.

iter_fields()
Iterates over the nested fields of the document examining all possible values of the occuring *resolvables*.

walk(*through_document_fields=False*, *visited_documents=frozenset([])*)
Iterates recursively over the nested fields, examining all possible values of the occuring *resolvables*.

Visits fields in a DFS order.

Parameters

- **through_document_fields** (*bool*) – If `True`, walks through nested *DocumentField* fields.
- **visited_documents** (*set*) – Keeps track of visited *documents* to avoid infinite recursion when `through_document_field` is `True`.

Returns iterable of *BaseField*

resolve_and_iter_fields(*role='default'*)
The same as *iter_fields()*, but *resolvables* are resolved using `role`.

resolve_and_walk(*role='default'*, *through_document_fields=False*, *visited_documents=frozenset([])*)
The same as *walk()*, but *resolvables* are resolved using `role`.

2.3 Roles

jsl.roles.**DEFAULT_ROLE**
A default role.

class jsl.roles.**Resolution**(*value*, *role*)
A resolution result, a `namedtuple`.

value
A resolved value (the first element).

role
A role to be used for visiting nested objects (the second element).

class jsl.roles.**Resolvable**
An interface that represents an object which value varies depending on a role.

resolve(*role*)
Returns a value for a given `role`.

Parameters **role** (*str*) – A role.

Returns A *resolution*.

iter_possible_values()
Iterates over all possible values except `None` ones.

class jsl.roles.**Var**(*values=None*, *default=None*, *propagate=<function all_>*)
A *Resolvable* implementation.

Parameters

- **values** (*dict or list of pairs*) – A dictionary or a list of key-value pairs, where keys are matchers and values are corresponding values.

  Matchers are callables returning boolean values. Strings and iterables are also accepted and processed as follows:

  – A string `s` will be replaced with a lambda `lambda r: r == s`;

  – An iterable `i` will be replaced with a lambda `lambda r: r in i`.
- **default** – A value to return if all matchers returned `False`.
- **propagate** (*callable, string or iterable*) – A matcher that determines which roles are to be propagated down to the nested objects. Default is *all_* that matches all roles.

values
A list of pairs (matcher, value).

propagate
A matcher that determines which roles are to be propagated down to the nested objects.

iter_possible_values()
Implements the *Resolvable* interface.

Yields non-`None` values from *values*.

resolve(*role*)
Implements the *Resolvable* interface.

Parameters **role** (*str*) – A role.

Returns

A *resolution*,

which value is the first value which matcher returns `True` and the role is either a given `role` (if `propagate`' matcher returns `True`) or *DEFAULT_ROLE* (otherwise).

class jsl.roles.**Scope**(*matcher*)
A scope consists of a set of fields and a matcher. Fields can be added to a scope as attributes:

```
scope = Scope('response')
scope.name = StringField()
scope.age = IntField()
```

A scope can then be added to a *Document*. During a document class construction process, fields of each of its scopes are added to the resulting class as *variables* which only resolve to fields when the matcher of the scope returns `True`.

If two fields with the same name are assigned to different document scopes, the matchers of the corresponding *Var* will be the matchers of the scopes in order they were added to the class.

Scope can also be used as a context manager. At the moment it does not do anything and only useful as a syntactic sugar – to introduce an extra indentation level for the fields defined within the same scope.

For example:

```
class User(Document):
    with Scope('db_role') as db:
        db._id = StringField(required=True)
        db.version = StringField(required=True)
    with Scope('response_role') as db:
        db.version = IntField(required=True)
```

Is an equivalent of:

```
class User(Document):
    db._id = Var([
        ('db_role', StringField(required=True))
    ])
    db.version = Var([
        ('db_role', StringField(required=True))
        ('response_role', IntField(required=True))
    ])
```

Parameters **matcher** (*callable, string or iterable*) – A matcher.

__field__
An ordered dictionary of *fields*.

__matcher__
A matcher.

2.3.1 Helpers

jsl.roles.**all_**(*role*)
A matcher that always returns `True`.

Return type bool

jsl.roles.**not_**(**roles*)
Returns a matcher that returns `True` for all roles except those are listed as arguments.

Return type callable

2.4 Exceptions

class jsl.exceptions.**SchemaGenerationException**(*message*)
Raised when a valid JSON schema can not be generated from a JSL object.

Examples of such situation are the following:

- A *variable* resolves to an integer but a *BaseField* expected;

- All choices of *OneOfField* are variables and all resolve to `None`.

Note: this error can only happen if variables are used in a document or field description.

Parameters **message** (*str*) – A message.

message = None
A message.

steps = None
A deque of *steps*, ordered from the first (the least specific) to the last (the most specific).

2.4.1 Steps

Steps attached to a *SchemaGenerationException* serve as a traceback and help a user to debug the error in the document or field description.

class jsl.exceptions.**Step**(*entity*, *role='default'*)
A step of the schema generation process that caused the error.

Parameters

- **entity** – An entity being processed.
- **role** (*str*) – A current role.

entity = None
An entity being processed.

role = None
A current role.

class jsl.exceptions.**DocumentStep**(*entity*, *role='default'*)
Bases: *jsl.exceptions.Step*

A step of processing a *document*.

Parameters

- **entity** (subclass of *Document*) – An entity being processed.
- **role** (*str*) – A current role.

class jsl.exceptions.**FieldStep**(*entity*, *role='default'*)
Bases: *jsl.exceptions.Step*

A step of processing a *field*.

Parameters

- **entity** (instance of *BaseField*) – An entity being processed.
- **role** (*str*) – A current role.

class jsl.exceptions.**AttributeStep**(*entity*, *role='default'*)
Bases: *jsl.exceptions.Step*

A step of processing an attribute of a field.

`entity` is the name of an attribute (e.g., `"properties"`, `"additional_properties"`, etc.)

Parameters

- **entity** (*str*) – An entity being processed.
- **role** (*str*) – A current role.

class jsl.exceptions.**ItemStep**(*entity*, *role='default'*)
Bases: *jsl.exceptions.Step*

A step of processing an item of an attribute.

`entity` is either a key or an index (e.g., it can be `"created_at"` if the current attribute is `properties` of a *DictField* or `0` if the current attribute is `items` of a *ArrayField*).

Parameters

- **entity** (*str or int*) – An entity being processed.
- **role** (*str*) – A current role.

2.5 Resolution Scope

class jsl.resolutionscope.**ResolutionScope**(*base=''*, *current=''*, *output=''*)
An utility class to help with translating `id` attributes of *fields* into JSON schema `"id"` properties.

Parameters

- **base** (*str*) – A URI, a resolution scope of the outermost schema.
- **current** (*str*) – A URI, a resolution scope of the current schema.
- **output** (*str*) – A URI, an output part (expressed by parent schema id properties) scope of the current schema.

base
A resolution scope of the outermost schema.

current
A resolution scope of the current schema.

output
An output part (expressed by parent schema id properties) scope of the current schema.

replace(*current=None*, *output=None*)
Returns a copy of the scope with the `current` and `output` scopes replaced.

alter(*field_id*)
Returns a pair, where the first element is the identifier to be used as a value for the `"id"` JSON schema field and the second is a new *ResolutionScope* to be used when visiting the nested fields of the field with id `field_id`.

Return type (str, *ResolutionScope*)

create_ref(*definition_id*)
Returns a reference (`{"$ref": ...}`) relative to the base scope.

jsl.resolutionscope.**EMPTY_SCOPE**
An empty *ResolutionScope*.

2.6 Changelog

2.6.1 0.1.0: 2015-05-13

- Introduce *roles*, *variables* and *scopes*;

- *NullField* by Igor Davydenko;
- Almost completely rewritten documentation;
- Various minor fixes.

2.6.2 0.0.10: 2015-04-28

- Fix spelling of `exclusiveMinimum` by Keith T. Star.

2.6.3 0.0.9: 2015-04-10

- Introduce the `ordered` argument for *get_schema()* that adds the ability to create more readable JSON schemas with ordered parameters.

2.6.4 0.0.8: 2015-03-21

- Add the ability to specify an id for documents and fields.

2.6.5 0.0.7: 2015-03-11

- More subclassing-friendly *DocumentMeta* which allows to override methods for collecting document fields and options and choose a container class for storing options;
- Various minor bugfixes.

2.6.6 0.0.5: 2015-03-01

- Python 3 support by Igor Davydenko.

2.7 Installation

```
$ pip install jsl
```

2.8 Contributing

The project is hosted on GitHub. Please feel free to send a pull request or open an issue.

2.8.1 Running the Tests

```
$ pip install -r ./requirements-dev.txt
$ ./test.sh
```

j

Symbols

A

B

C

D

E

F

G

I

J

M

N

O

P

R

S

T

U

V

W