jsonpublish Documentation

Release 0.1.3

Andrey Popp

December 03, 2014

This package provides configurable JSON encoder based on `simplejson` or `json` module from Python's standard library.

When and why you should use jsonpublish:

- You want all JSON serialization code to be in one place.
- You want your serialization code to be flexible and structured.
- Sometimes you want to alter serialization for some objects.

CHAPTER 1

Custom types serialization

Suppose you have some data of your application modeled as Python's classes (it may be, for example, Django models or just plain old Python's classes):

```
class User(object):

  def __init__(self, username, birthday):
    self.username = username
    self.birthday = birthday
```

Now if you want to serialize `User` objects as JSON documents you can't simply use `json` module, because it just doesn't know how to represent your objects as JSON documents. So you need to write a function which converts `User` objects to something which can be serialized, for example `dict`. With time your app grows and complexity grows along so you need somehow to structure you serialization machinery, let's see how `jsonpublish` can help us there:

```
from jsonpublish import register_adapter

@register_adapter(User)
def adapt_user(user):
  return {
    "username": user.username,
    "birthday": user.birthday
  }
```

Now you can serialize your `User` objects:

```
>>> from jsonpublish import dumps
>>> print dumps(User("andrey", 1987))
{"username": "andrey", "birthday": 1987}
```

CHAPTER 2

Parametrized adapters

Sometimes you want to alter serialization of some object, For example, let's write another adapter for `User` objects which can change it behaviour based on arguments given:

```
@register_adapter(User)
def adapt_user(user, include_birthday=True):
  if include_birthday:
    return {
      "username": user.username,
      "birthday": user.birthday
    }
  else:
    return {"username": user.username}
```

The question now is how to pass `include_birthday` keyword argument right to adapter, the answer is to use `jsonpublish.jsonsettings()`:

```
>>> from jsonpublish import jsonsettings
>>> user = User("andrey", 1987)
>>> user_m = jsonsettings(user, include_birthday=False)

>>> print dumps(user)
{"username": "andrey", "birthday": 1987}

>>> print dumps(user_m)
{"username": "andrey"}
```

As you can see, by wrapping our `User` object in `jsonpublish.jsonsettings()` we can pass arbitrary keyword arguments to corresponding adapter so we can alter serialization by per-object basis.

Function `jsonsettings` actually doesn't alter object in any way, it just "annotates" it with some metadata needed for corresponding adapter. You can work with wrapped object as before – all methods and attributes are still there and even `isinstance` check works the right way:

```
>>> user_m == user
True
>>> user_m.username
"andrey"
>>> isinstance(user_m, User)
True
```

CHAPTER 3

Reporting bugs and working on jsonpublish

Development takes place at GitHub, you can clone source code repository with the following command:

```
% git clone git://github.com/andreypopp/jsonpublish.git
```

In case submitting patch or GitHub pull request please ensure you have corresponding tests for your bugfix or new functionality.

CHAPTER 4

API reference

jsonpublish.**dumps**(*obj*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *indent=None*, *separators=None*, *encoding='utf-8'*, *default=None*, *use_decimal=True*, *namedtuple_as_object=True*, *tuple_as_array=True*, ***kw*)

Serialize *obj* using globally configured JSON encoder

Accepted arguments are the same as `json.dumps()` accepts

jsonpublish.**register_adapter**(*typ*, *adapter=None*)

Register `adapter` for type `typ`

If no `adapter` supplied then this method returns decorator.

jsonpublish.**jsonsettings**(*o*, ***settings*)

Create a proxy which carries JSON encoder settings

class jsonpublish.**JSONEncoder**(**args*, ***kwargs*)

Configurable JSON encoder

It serializes object by consulting adapter registry. Registry can be modified by accessing *adapters* attribute of encoder which is of type *AdapterRegistry*.

Attr adapters instance of `AdapterRegistry` which is used for serialization by encoder

encode(*o*)

Return a JSON string representation of a Python data structure.

```
>>> JSONEncoder().encode({"foo": ["bar", "baz"]})
'{"foo": ["bar", "baz"]}'
```

iterencode(*o*, *_one_shot=False*)

Encode the given object and yield each string representation as available.

For example:

```
for chunk in JSONEncoder().iterencode(bigobject):
    mysocket.write(chunk)
```

class jsonpublish.**AdapterRegistry**

Registry of adapters

lookup_adapter(*typ*)

Lookup adapter for `typ`

register_adapter(*typ*, *adapter=None*)

Register `adapter` for type `typ`

If no `adapter` supplied then this method returns decorator.

Python Module Index

Index