Serializer Documentation

Release 0.1

Konsta Vesterinen

February 19, 2013

CONTENTS

Serializer provides Easy object serialization. Mimics RoR ActiveRecord serializer.

CHAPTER

ONE

QUICKSTART

1. Make your class inherit Serializable mixin.
2. Define attributes() method and make it return a list of property names

Example:

```
from serializer import Serializable


class User(Serializable):
    first_name = 'John'
    last_name = 'Matrix'
    email = 'john.matrix@example.com'
    protected_property = 'this value is not included in json'

    def attributes(self):
        return ['first_name', 'last_name', 'email']

user = User()
user.as_json()
# "{'first_name': 'John', 'last_name': 'Matrix'}"

user.as_json_dict()
# {'first_name': 'John', 'last_name': 'Matrix'}
```

CHAPTER

TWO

HOW TO HANDLE RELATIONS

Serializer supports serializing of deep object structures. Let's say we have two classes Event and Location and we want to jsonify an event along with its location.

```
class Location(Serializable):
    name = 'Some Location'

    def attributes(self):
        return ['name']

class Event(Serializable):
    name = 'Some Event'
    location = Location()

    def attributes(self):
        return ['name', 'location']


event = Event()
event.as_json()
# "{'name': 'Some Event', 'location': {'name': 'Some Location'}}"
```

CHAPTER

THREE

USING EXCLUDE, INCLUDE AND ONLY

You can fine-grain what gets included in json format by using exclude, include and only parameters.

```
User.as_json(only=['name', 'email'])  # include only name and email

# include all the fields defined in attributes as well as age
User.as_json(include=['age'])

# include all the field defined in attributes but exclude 'email'
User.as_json(include=['email'])
```

CHAPTER

FOUR

USING ATTRIBUTE SETS

Many times you may have situations where having one default attribute list is not enough. For example you may have multiple views that return user details and many views that return user with only its basic info.

```
class User(Serializable):
    first_name = 'John'
    last_name = 'Matrix'
    email = 'john.matrix@example.com'
    age = 33
    is_active = True

    def attributes(self):
        return ['first_name', 'last_name', 'email']

    def attribute_sets(self):
        return {'details': ['age', 'is_active']}


user = User()
user.as_json_dict(include='details')
'''
    {
        'first_name': 'John',
        'last_name': 'Matrix',
        'email': 'john.matrix@example.com',
        'age': 33,
        'is_active': True
    }
'''
```

4.1 API Documentation

class serializer.**Serializable**

This class mimics the functionality found in RoR ActiveModel. For more info see: http://api.rubyonrails.org/classes/ActiveModel/Serializers/JSON.html

as_json(*only=None*, *exclude=None*, *include=None*)

Returns object attributes as a dictionary with jsonified values

Parameters

- **only** – a list containing attribute names to only include in the returning dictionary
- **exclude** – a list containing attributes names to exclude from the returning dictionary

- **include** – a list containing attribute names to include in the returning dictionary

Without any options, the returned JSON string will include all the fields returned by the models attribute() method. For example:

```
>>> class User(Serializable):
...     def attributes(self):
...         return [
...             'id',
...             'first_name',
...             'last_name'
...         ]
...
>>> user = User()
>>> user.first_name = 'John'
>>> user.last_name = 'Matrix'
>>> user.as_json()
{"id": 1, "first_name": "John", "last_name": "Matrix"}
```

The 'only' and 'exclude' options can be used to limit the attributes included. For example:

```
>>> user.as_json(only=['id', 'first_name'])
{"id": 1, "first_name": "John"}

>>> user.as_json(exclude=['id'])
{"first_name": "John", "last_name": "Matrix"}
```

To include the result of some additional attributes, method calls, attribute sets or associations on the model use 'include'.

```
>>> user.weight = 120
>>> user.as_json(include=['weight'])
{"id": 1, "first_name": "John", "last_name": "Matrix", "weight": 120}
```

Sometimes its useful to assign aliases for attributes. This can be achieved using keyword 'as'.

```
>>> user.as_json(only=['first_name as alias'])
{"alias": "John"}
```

In order to fine grain what gets included in associations you can use 'include' parameter with additional arguments.

```
>>> user.as_json(include=[('posts', {'include': 'details'})]
{
    "id": 1,
    "name": "John Matrix",
    "first_name": "John",
    "last_name": "Matrix",
    "weight": 100,
    "posts": [
      {"id": 1, "author_id": 1, "title": "First post"},
      {"id": 2, author_id: 1, "title": "Second post"}
    ]
}
```

Second level and higher order associations work as well:

```
>>> user.as_json('include'= [('posts',
...       {
...        'include': [
...                 ('comments', {'only': ['body']})
```

```
...             ],
...         'only': ['title']
...         }
... )]
{
    "id": 1,
    "first_name": "John",
    "last_name": "Matrix",
    "weight": 100,
    "posts": [
        {
            "comments": [
                {"body": "1st post!"}, {"body": "Second!"}
            ],
            "title": "Welcome to the weblog"
        },
    ]
}
```

attribute_sets()

This method should return a dict with keys as attribute set names and values as desired attribute sets

Attribute sets can be used as a convenient shortcuts in as_json()

Examples:

```
>>> User(Serializable):
...     def attribute_sets(self):
...         return {'details': ['id', 'name', 'age']}
>>> User(id=1, name='someone',).as_json(only='details')
{'id': 1, 'name': 'Someone', 'age': 14}
```

attributes()

This method is being used by as_json for defining the default json attributes for this model.

Serializable objects can override this and return a list of desired default attributes.

Examples:

```
>>> User(Serializable):
...     def __init__(name, age):
...         self.name = name
...         self.age = age
...
...     def attributes(self):
...         return ['name', 'age']
...
>>> User('John', 50).as_json()
{'name': 'John', 'age': 50}
```

to_json(*only=None*, *exclude=None*, *include=None*)

Returns the object attributes serialized in json format

Parameters

- **only** – a list containing attribute names to only include in the returning json
- **exclude** – a list containing attributes names to exclude from the returning json
- **include** – a list containing attribute names to include in the returning json

to_xml (*only=None*, *exclude=None*, *include=None*, ***kwargs*)
Returns the object attributes serialized in xml format

Parameters

- **only** – a list containing attribute names to only include in the returning xml
- **exclude** – a list containing attributes names to exclude from the returning xml
- **include** – a list containing attribute names to include in the returning xml

serializer.**register_dumper** (*key*, *dumper_callable*)
Registers new dumper for given class type

Examples::

```
>>> class MyClassA(object):
...     pass
>>> class MyClassB(MyClassA):
...     pass
>>> register_dumper('MyClassA', lambda a: 'myclass')
>>> dump_object(MyClassA())
"myclass"
>>> dump_object(MyClassB())
<MyClassB instance>

>>> class MyClassA(object):
...     pass
>>> class MyClassB(MyClassA):
...     pass
>>> register_dumper(MyClassA, lambda a: 'myclass')
>>> dump_object(MyClassA())
"myclass"
>>> dump_object(MyClassB())
"myclass"
```

4.2 Changelog

Here you can see the full list of changes between each Serializer release.

4.2.1 0.2.1 (2013-02-16)

- Added API documentation

4.2.2 0.2 (2013-01-26)

- Added XML serialization

4.2.3 0.1 (2012-12-04)

- Initial public release

4.3 License

PYTHON MODULE INDEX

S