soap Documentation

Release 0.1a

Joe Dallago

November 08, 2012

CONTENTS

Soap is a validation library heavily adapted from Colander, another library with similar intentions. As you're creating web applications, you undoubtedly reach a point where you're accepting data from HTML POST forms or from AJAX requests. The data users enter needs to be validated before it is put into a database, for both security and data integrity reasons. Additionally, you might want to alter a user's data to match a specific format, such as trimming leading and trailing whitespace, in a consistent manner. This is where a library such as Colander/Soap comes in. Just so you're forwarned, Colander is a two year project in the making, so it is definitely far more mature than Soap. I created Soap because I am currently doing a project that I am heavily organizing around the data model, and I wanted to be able to configure my validation library to accommodate the large range of different relationships that occur between my data structures. This is something that Colander is only semi-capable of achieving. Let's look at how you use Soap:

CHAPTER

ONE

SCHEMA CONFIGURATION(IMPERATIVE)

Soap allows you to define a number of schemas, and then use those schemas to validate a Python dict. Like Colander, Soap is capable of being configuring schemas both imperatively and declaratively. Let's take a look at the imperative configuration first, it will look something like this:

```
from soap import (
    SchemaMode,
    Mapping,
    SchemaNode,
    Int,
    String
)


TestSchema = SchemaModel('TestSchema',
                         Mapping(),
                         SchemaNode(Int(), name='id'),
                         SchemaNode(String(), name='name', missing=''))
```

As you can see, you need to create an instance of `soap.SchemaModel` to create a schema. `soap.SchemaModel` takes the name of the schema as its first argument, the type of schema as its second argument, and its child `soap.SchemaNode`s as its successive arguments. Each :class:`soap.SchemaNode` takes a type for its first argument, in this case an instance of `soap.Int` and `soap.String`, and additionally the following kwargs:

name The name of the current node. Represented by a key in the value being deserialized.

missing The value that is given to the specified node if it is missing.

validator A single validator or a list of validators that will be used to make sure that the data is both safe and of the correct format.

preparer A single preparer or a list of preparer that will be used to alter the data in a specific way, to the application developers liking. For example, removing any unnecessary newlines would be done with a preparer.

Once a schema is defined, then we want to deserialize it. This function call will insert any missing values, run all of the specified preparers, run all of the specified validators, and make sure all required data is given. At any point during this process, if something fails, a `soap.Invalid` exception is thrown.

```
json = {
    'id': '0',
    'name': 'jayd3e'
}
```

```
payload = TestSchema.deserialize(json)
# payload = {
#     'id': 0,
#     'name': 'jayd3e'
# }
```

Notice how the '0' gets changes to an integer? Thus becoming 0. This example isn't very exciting, but it lays out the general process for using Soap, define a schema, pass your user created data into the `deserialize` function, receive some validated output. There is the additional step of handling errors, but we can get to that in a second.

CHAPTER

TWO

SCHEMA CONFIGURATION(DECLARATIVE)

If you have ever used SQLAlchemy, you are probably very familiar with the declarative style of model configuration. With Soap, you can configure your schemas in a similar style. They end up looking something like this:

```
from soap import (
    SchemaMode,
    Mapping,
    SchemaNode,
    Int,
    String,
    Boolean,
    DateTime
)

class TestSchema(SchemaModel):
    id = SchemaNode(Int())
    name = SchemaNode(String())
    booly = SchemaNode(Boolean())
    datey = SchemaNode(DateTime())
```

Here we introduced the declarative style of configuring Soap, as well as a couple new datatypes, `soap.Boolean` and `soap.DateTime`. These two configuration mechanisms effectively create the same thing, except just using different styles. To deserialize this schema, we would do the following:

```
json = {
    'id': 0,
    'name': 'blah',
    'booly': 'true',
    'datey': '2007-01-25T12:00:00Z'
}

schema = TestSchema()
payload = schema.deserialize(json)
# payload = {
#     'id': 0,
#     'name': 'blah',
#     'booly': True,
#     'datey': date
# }
```

It should be noted that you can use all of Soap's functionality regardless of which configuration style your're using, but for the rest of the docs, I plan on using the declarative style.

CHAPTER

THREE

RELATIONSHIPS

Relationships are where Soap really shines in comparison to Colander. Soap allows you to define SQLAlchemy-like relationships between your Schemas, so you can reuse ALL of your defined schemas. This is ideal for advanced relationships between data structures. We can achieve this by creating `soap.SchemaNode`s with the `:class:`soap.Relationship` type. To configure some relationships, do something like this:

```
from soap import (
    SchemaMode,
    Mapping,
    SchemaNode,
    Int,
    String,
    Boolean,
    DateTime
)

class ChildSchema(SchemaModel):
        id = SchemaNode(Int())
        name = SchemaNode(String())
        parent_node = SchemaNode(Relationship('TestSchema', uselist=False), missing={})

class TestSchema(SchemaModel):
    id = SchemaNode(Int())
    name = SchemaNode(String())
    booly = SchemaNode(Boolean())
    datey = SchemaNode(DateTime())
    sub_node = SchemaNode(Relationship('ChildSchema', uselist=False), missing={})
    sub_seq_nodes = SchemaNode(Relationship('ChildSchema'), missing=[])
```

So here we took our TestSchema from before, and added a ChildSchema. We then define three total relationships between them. We want to have a field named `sub_node` in TestSchema that contains a single instance of ChildSchema, and a field named `sub_seq_nodes` that contains a list of ChildSchemas. Notice that in order to specify that we only want a single ChildSchema stored under `sub_node` we set the `uselist` kwarg of `soap.Relationship` to False. Additionally, we also mention that each ChildSchema should have a `parent_node` field that contains a single TestSchema. This will result in the following deserialization:

```
json = {
    'id': 0,
    'name': 'blah',
    'booly': 'true',
    'datey': date_str,
    'sub_node': {
        'id': 0,
        'name': 'sub_blah',
```

```
        'del_key': 'this key should get removed'
    },
    'sub_seq_nodes': [{
        'id': '0',
        'name': 'sub_seq_blah_0',
        'parent_node': {
            'id': 0,
            'name': 'blah',
            'booly': 'false',
            'datey': date_str
        },
        'del_key': 'this key should be removed'
    },
    {
        'id': 1,
        'name': 'sub_seq_blah_1'
    }]
}

schema = TestSchema()
payload = schema.deserialize(json)
# payload = {
#     'id': 0,
#     'name': 'blah',
#     'booly': True,
#     'datey': date,
#     'sub_node': {
#         'id': 0,
#         'name': 'sub_blah',
#         'parent_node': {}
#     },
#     'sub_seq_nodes': [{
#         'id': 0,
#         'name': 'sub_seq_blah_0',
#         'parent_node': {
#             'id': 0,
#             'name': 'blah',
#             'booly': False,
#             'datey': date,
#             'sub_seq_nodes': [],
#             'sub_node': {}
#         }
#     }, {
#         'parent_node': {},
#         'id': 1,
#         'name': 'sub_seq_blah_1'
#     }]
# }
```

CHAPTER

FOUR

VALIDATORS