Rückenwind Documentation

Release 0.0.1-

Florian Ludwig

March 31, 2015

Contents

Contents:

CHAPTER 1

Getting Started

1.1 What and Why

Scope is a dependency injection mechanism for python. It solves two things:

- **Storing thread-local data in tornado**: Somewhere deep down your call stack (within several coroutines) you end up needing the current user name or some request handler. You may pass these information down the call stack, expanding all function attributes along the way. Or you inject them.
- **Keeping track of replacable components**: Maybe you implement two different login systems and depending on some configuration want to switch that backend. Injecting the login system instead of importing it where you need it is the scope way to do it.

Note: `Scope` was written for tornado. It can be used outside of tornado but it depends on tornado. The examples are not tornado specific in any way.

1.2 In Action

A scope is basically a dictionary:

```
import scope

my_scope = scope.Scope()
my_scope['foo'] = 'bar'
assert my_scope.get('foo') == 'bar'
```

So `'bar'` is what you want to store and `'foo'` is the key where it is stored. `'bar'` might be as well that login system instance, your request handler or current user name.

The *scoping* of *Scope* comes when it is used as context manager:

```
my_scope = scope.Scope()
my_scope['foo'] = 'bar'

with my_scope():
    # the global module scope's .get
    # knows we are inside "my_scope"
    assert scope.get('foo') == 'bar'
```

The usage becomes clearer when introducing some function calls:

```
def some_other_function():
    assert scope.get('foo') == 'bar'

    # we can access the "current scope"
    current_scope = scope.get_current_scope()

    # and use it like a dict
    assert current_scope['foo'] == 'bar'


def main():
    # the my_scope variable is now defined
    # inside a function scope, so it is not
    # accessable outside of it, like in
    # some other function
    my_scope = scope.Scope()
    my_scope['foo'] = 'bar'

    with my_scope():
        some_other_function()


main()
```

You might wonder what happens when you enter a scope inside a scope: What you expect. Being nested is what they where build for.

```
app_scope = scope.Scope()
app_scope['db_password'] = 'foobar'
app_scope['user'] = 'anon'

nested_user_scope = scope.Scope()

def handle_user():
    current_scope = scope.get_current_scope()

    # now inside a user session we might want
    # to store some user data
    current_scope['user'] = 'admin'
    # and we still want to access the
    # app data
    assert current_scope.get('db_password') == 'foobar'

    # we can change the db_password
    current_scope['db_password'] = 'secret_admin_password'

    # and all our data is as expected:
    assert current_scope.get('db_password') == 'secret_admin_password'
    assert current_scope.get('user') == 'admin'


def main():
    with app_scope():
        # we are inside "some app" that
        # stores it "db_password" inside
        # a scope
        assert scope.get('db_password') == 'foobar'
```

```
        with nested_user_scope():
            handle_user()

        # the app scope is untouched
        assert scope.get('db_password') == 'foobar'
        assert scope.get('user') == 'anon'


main()
```

And one final difference you might want to be aware of compared to a standard dict:

```
my_scope = scope.Scope()

# using .get on a non existing key
# does NOT return None as a python
# dict but raises an IndexError.
with pytest.raises(IndexError):
    my_scope.get('foo')

# if you want to None for non existing
# values, do so explicitly:
assert my_scope.get('foo', None) is None
```

CHAPTER 2

Indices and tables

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