UDC 004.432

NAMEKO LIBRARY

ALIAKSANDR VOITAU, ARKADZI OSKIN
Polotsk State University, Belarus

In this article I want to describe some of the features of Nameko, the problems I encountered, and their solutions, as well as some useful extensions.

Nameko is a python library for building microservices. The Nameko service is just a Python class, some of whose methods are labeled with @rpc decorator. RPC is a remote procedure call which means you can call these methods from other Nameko services. Also you can call service methods from non-Nameko services. For example, if you build a web application and you need to perform some background logic you can call this RPC method.

Nameko RPC implementation works on top of the AMQP protocol. That means you need a AMQP broker. One of them is a RabbitMQ. Briefly, when you start the Nameko service, RabbitMQ creates queues, and when you call the RPC method, Nameko sends it to the queue. And when the service is not busy, it executes this method. In the example below you can see how it works [1].

```
#service.py
from nameko.rpc import rpc, RpcProxy
class ServiceY:
    name = "service_y"
    @rpc
    def append_identifier(self, value):
        return u"{}-y".format(value)

class ServiceX:
    name = "service_x"
    y = RpcProxy("service_y")
    @rpc
    def remote_method(self, value):
        res = u"{}-x".format(value)
        return self.y.append_identifier(res)


#app.py
from nameko.standalone.rpc import ClusterRpcProxy
config = {
    'AMQP_URI': AMQP_URI  # e.g. "pyamqp://guest:guest@localhost"
}
with ClusterRpcProxy(config) as cluster_rpc:
    cluster_rpc.service_x.remote_method("hello") # "hello-x-y"
```

Normal RPC calls block until the remote method completes, but proxies also have an asynchronous calling mode to background or parallelize RPC calls:

```
with ClusterRpcProxy(config) as cluster_rpc:
    hello_res = cluster_rpc.service_x.remote_method.call_async("hello")
    world_res = cluster_rpc.service_x.remote_method.call_async("world")
    # do work while waiting
    hello_res.result()  # "hello-x-y"
    world_res.result()  # "world-x-y"
```

In some cases this code may have problems. For example, if the service doesn't start, the ClusterRpcProxy will wait forever. And it is very difficult to understand when your backend is not responding. The solution is to use the timeout parameter when creating a ClusterRpcProxy. And when the time is out it raises an exception which you can catch and say your frontend about the problem and also log it. Here is the example:

```
try:
    with ClusterRpcProxy(config, timeout=3) as cluster_rpc:
        cluster_rpc.service_x.remote_method("hello")
except RpcTimeout:
    logger.error("service is not responding")
```

There is no possibility to set a timeout inside the service in Nameko by using RpcProxy. But Nameko uses eventlet library to do multithreading. And you can use eventlet's timeout to limit waiting [2].

```
with eventlet.timeout.Timeout(3):
    try:
        self.y.append_identifier(res)
    except eventlet.timeout.Timeout:
        logger.error("service is not responding")
```

One cool thing is that Nameko has extensions support which you may find useful when developing your own Nameko services. One of them is nameko-sqlalchemy. SQLAlchemy is the Python SQL toolkit and Object Relational Mapper that gives application developers the full power and flexibility of SQL. To use it with Nameko you need to create YAML config file and describe databases and services. Then you need to declare the database using the "Declarative_base" function and set the name constructor parameter as in the config file. After that you can create a service field using "DatabaseSession" with declared database as a constructor parameter [3].

```
#config.yaml
DB_URIS:
    "service_x:first_base": "mysql://user:pass@host:port/firstbase?charset=utf8"
    "service_x:second_base": "mysql://user:pass@host:port/secondbase?charset=utf8"
    "service_y:other_base": "postgresql://user:pass@host:port/otherbase"
```

```
#service.py
from nameko_sqlalchemy import DatabaseSession
from sqlalchemy.ext.declarative import declarative_base

FirstBase = declarative_base(name="first_base")
SecondBase = declarative_base(name="second_base")
OtherBase = declarative_base(name="other_base")

class ServiceX:
    name = "service_y"
    first_base = DatabaseSession(FirstBase)
    second_base = DatabaseSession(SecondBase)
class ServiceY:
    name = "service_x"
    other_base = DatabaseSession(OtherBase)
```

Ok, now you can declare a database session and what can you do with it? How to request some data and transfer it between other services? The main problem is that the sqlalchemy object is not serializable. A serializable object means that it can be represented, for example, as json. It is necessary to transfer objects through a message broker. It is used when calling the RPC method or when you want to return the result. A simple way to create a serializable object is to create a python dict or list. It is easy to do because they have a similar structure with JSON.

The marshmallow-sqlalchemy library can serialize the SQLAlchemy object very easily. To do it the first thing you need is to create a table class and declare the table fields, then you need to create a shema object, where you can declare the Meta SQLAlchemy object. And that's all, now you can do the serialization [4].

```
Base = declarative_base()
class Author(Base):
    __tablename__ = 'authors'
    id = sa.Column(sa.Integer, primary_key=True)
    name = sa.Column(sa.String)
def __repr__(self):
        return '<Author(name={self.name!r})>'.format(self=self)
class AuthorSchema(ModelSchema):
    class Meta:
        model = Author
author_schema = AuthorSchema()
author = Author(name='Chuck Paluhniuk')
session.add(author)
session.commit()
dump_data = author_schema.dump(author).data
# {'id': 321, 'name': 'Chuck Paluhniuk'}
author_schema.load(dump_data, session=session).data
# <Author(name='Chuck Paluhniuk')>
```

REFERENCES

1. Nameko documentation [Electronic resource]. – Mode of access:https://nameko.readthedocs.io/en/stable/index.html – Date of access: 09.02.2019.
2. Eventlet documentation: timeout [Electronic resource]. – Mode of access: http://eventlet.net/doc/modules/timeout.html – Date of access: 09.02.2019.
3. Source code of nameko_sqlalchemy: class Database [Electronic resource]. – Mode of access: https://github.com/nameko/nameko-sqlalchemy/blob/master/nameko_sqlalchemy/database.py – Date of access: 09.02.2019.
4. Marshmallow-SQLAlchemy documentation [Electronic resource]. – Mode of access: https://marshmallow-sqlalchemy.readthedocs.io/en/latest/ – Date of access: 16.02.2019.