Nameko Salesforce Documentation

Release

Student.com

Nov 29, 2017

Contents:

A Nameko extension with entrypoints for handling Salesforce Streaming API events and a dependency provider for easy communication with Salesforce REST API.

The Streaming API extension is based on Nameko Cometd Bayeux Client and the REST API dependency id based on Simple Salesforce.

CHAPTER 1

Quick Start

Create a service which handles Salesforce Contact objects changes and also has an RPC endpoint for creating new Contact objects in Salesforce.

```
# service.py

from nameko.rpc import rpc
from nameko_salesforce.streaming import handle_sobject_notification
from nameko_salesforce.api import SalesforceAPI

class Service:

    name = 'some-service'

    salesforce = SalesforceAPI()

    @handle_sobject_notification('Contact', exclude_current_user=False)
    def handle_contact_updates(
        self, sobject_type, record_type, notification
    ):
    """ Handle Salesforce contacts updates
    """
    print(notification)

    @rpc
    def create_contact(self, last_name, email_address):
    """ Create a contact in Salesforce
    """
    self.salesforce.Contact.create(
        {'LastName': last_name,'Email': email_address})
```

Create a config file with essential settings:

```
# config.yaml

AMQP_URI: 'pyamqp://guest:guest@localhost'
```

```
SALESFORCE:
    USERNAME: ${SALESFORCE_USERNAME}
    PASSWORD: ${SALESFORCE_PASSWORD}
    SECURITY_TOKEN: ${SALESFORCE_SECURITY_TOKEN}
    SANDBOX: False
```

Run the service providing your Salesforce credentials:

```
$ SALESFORCE_USERNAME=rocky \
  SALESFORCE_PASSWORD=*** \
  SALESFORCE_SECURITY_TOKEN=*** \
  nameko run --config config.yaml service
```

Finally, open another shell and call the RPC endpoint to create a new user:

```
$ nameko shell --config config.yaml
In [1]: n.rpc['some-service'].create_contact('Yo', 'yo@yo.yo')
```

You should see a new contact created in Salesforce and your service should get a notification. In the first shell you'll find the notification printed:

```
{'event': {'replayId': 1, 'type': 'created' ...
```

CHAPTER 2

Salesforce Streaming API Client

2.1 Nameko Entripoints

The Streaming API extension comes with the following set of entrypoints:

subscribe Implementing *"subscribe, listen and handle"* mechanism.

handle_notification **and** ***handle_sobject_notification*** Implementing *"declare, subscribe, listen and handle"* mechanism.

2.1.1 Simple Subscription

If you have Push Topics defined in Salesforce, you can use the basic `subscribe` entrypoint decorator for a simple *subscribe, listen and handle* kind of work:

```
# service.py

from nameko_salesforce.streaming import subscribe

class Service:

    name = 'some-service'

    @subscribe('/topic/InvoiceStatementUpdates')
    def handle_event(self, topic, event):
        """ Handle Salesforce invoice statement updates
        """
```

2.1.2 Generic Notification Handling

The `handle_notification` entrypoint has the ability to declare Push Topics for you. Pass the Push Topic Query string in `query` argument and the service will create Push Topics automatically on start up so then it can follow with

subscription:

```
# service.py

from nameko_salesforce.streaming import handle_notification

class Service:

    name = 'some-service'

    @handle_notification(
        name='ContactUpdates',
        query='SELECT ...'
    )
    def handle_contact_updates(self, name, notification):
        """ Handle Salesforce contacts updates
        """
```

If a Push Topic with the same name already exist, it will be updated.

There are more options available for defining Push Topics:

```
# service.py

from nameko_salesforce.streaming import handle_notification

class Service:

    name = 'some-service'

    @handle_notification(
        name='ContactNameUpdated',
        query='SELECT firstName, lastName ...',
        notify_for_fields=NotifyForFields.select,
        notify_for_operation_update=True,
        notify_for_operation_create=False,
        notify_for_operation_undelete=False,
        notify_for_operation_delete=False)
    def handle_contact_updates(self, name, notification):
        """ Handle Salesforce contacts name changes

        Handles only first and last name changes of existing contacts.
        Ignores any other modification.

        """
```

Find details about `notify_for_fields` and `notify_for_operation_...` argument options in Event Notification Rules section of Salesforce documentation website.

2.1.3 Salesforce Objects Notification Handling

There is also `handle_sobject_notification` entrypoint extending the *generic* `handle_notification` by a functionality which constructs the Push Topic query automatically in the form ready for handling notification of Salesforce object changes. Instead of `query` argument it requires Salesforce object name as `sobject_type` argument to be set defining the object the query should be created for.

Declaring notification of Salesforce object changes:

```
# service.py

from nameko_salesforce.streaming import handle_sobject_notification

class Service:

    name = 'some-service'

    @handle_sobject_notification('Contact')
    def handle_contact_updates(
        self, sobject_type, record_type, notification
    ):
        """ Handle Salesforce contacts updates
        """
```

The entrypoint decorator also takes optional `record_type` argument narrowing down the query filters by selecting objects of a specific Salesforce RecordType.

Tip: In addition to type filters there is also `exclude_current_user` argument which filters out notifications about changes done by the same user as the one the entrypoint uses to connect to Salesforce server. You may find this filter useful when listening to changes which may be also done by the Salesforce API dependency of the same service and you want to avoid circular handling (see the *Quick Start* example).

The following example shows available notification options:

```
# service.py

from nameko_salesforce.streaming import handle_sobject_notification

class Service:

    name = 'some-service'

    @handle_sobject_notification(
        sobject_type='Contact',
        record_type='Student',
        exclude_current_user=True,
        notify_for_fields=NotifyForFields.select,
        notify_for_operation_update=True,
        notify_for_operation_create=False,
        notify_for_operation_undelete=False,
        notify_for_operation_delete=False)
    def handle_contact_updates(
        self, sobject_type, record_type, notification
    ):
        """ Handle Salesforce student contacts name changes

        Handles only name changes of existing contacts of type of student.
        Ignores any other modification.

        Also ignores changes done by this service (more precisely changes
        done by the same API user as this extension use for connection
        to Salesforce streaming API).

        """
```

Note that the entrypoint decorator creates a Push Topic in Salesforce which will exclude changes not satisfying the defined conditions already in Salesforce. Therefore the server will send to clients notifications only for relevant changes.

2.2 Message Durability

The streaming API extension allows you to track last received replay IDs for each topic and use it on subscription to ask Salesforce to replay all missed events from that point.

Salesforce calls this mechanism "Replaying PushTopic Streaming Events". For more information about durable events, see Salesforce documentation on Message Durability.

The streaming API extension has the ability to persist replay IDs in Redis and load them when subscribing to channels. To enable the replay mechanism add the following keys to your Nameko configuration:

```
# config.yaml

SALESFORCE:
    ...
    PUSHTOPIC_REPLAY_ENABLED: True
    PUSHTOPIC_REPLAY_REDIS_URI: redis://some.redis.host:6379/11
    PUSHTOPIC_REPLAY_TTL: 3600
```

Salesforce promises to keep events for 24 hours, however we noticed that the real maximum retention window is somehow smaller and that Salesforce sometimes complains about invalid replay IDs even after only 18 hours.

2.3 Subscription Stacking

Note that the decorated entrypoint method gets the `topic`, notification `name` or defined `sobject_type` and `record_types` as first arguments. This is useful when making a single entrypoint method handling notifications of multiple channels by stacking the decorators. See the example in the following section.

2.3.1 Salesforce to Nameko Event Proxy

The following snippet shows a simple mechanism proxying Salesforce notifications to Nameko events.

```
# service.py

from nameko.events import EventDispatcher
from nameko_salesforce.streaming import handle_sobject_notification

class Service:

    name = 'some-service'

    dispatch = EventDispatcher()

    @handle_sobject_notification('Lead')
    @handle_sobject_notification('Opportunity')
    def handle_salesforce_updates(
        self, sobject_type, record_type, notification
    ):
        """ Proxy Salesforce object changes notifications to Nameko events
```

```
        """
        event = 'salesforce_{}_{}'.format(
            sobject_type.lower(), notification['event']['type'])
        payload = notification['sobject']
        self.dispatch(event, payload)
```

The proxy will dispatch events with descriptive names such as `salesforce_lead_updated` or `salesforce_opportunity_created` and with details of affected Salesforce object as payload.

CHAPTER 3

Salesforce Rest API Client

3.1 Nameko Dependency Provider

The `SalesforceAPI` dependency provider wraps Simple Salesforce client plus brings additional benefits of **client pooling** and connection setup using standard Nameko config.

Usage:

```
# service.py

from nameko_salesforce.api import SalesforceAPI

class Service:

    name = 'some-service'

    salesforce = SalesforceAPI()

    @rpc
    def create_contact(self, last_name, email_address):
        self.salesforce.Contact.create(
            {'LastName': last_name,'Email': email_address})
```