nuropb

Release 0.1.7

Robert Betts

Oct 14, 2023

CONTENTS

CHAPTER

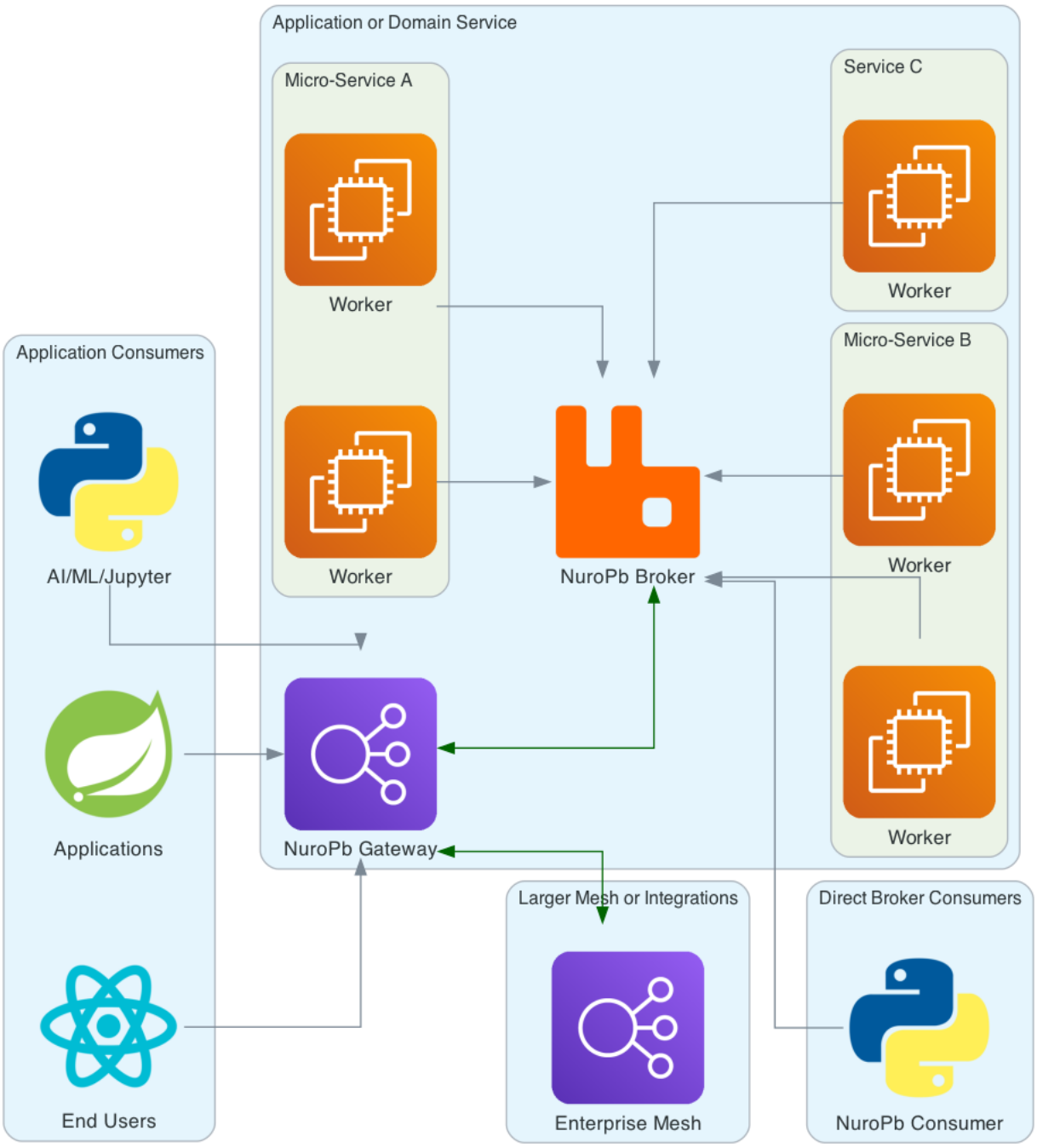
ONE

PLUMBING, ROUTING AND COMMUNICATIONS FOR DISTRIBUTED, ASYNCHRONOUS, EVENT DRIVEN, SERVICES

If you have code that you want to easily scale, communicate with other services or provide access to consumers, or:

- You'd like to scale a service horizontally, many times over at an unknown scale.
- Your service needs to communicate with other services
- Implement event driven processes and flows
- A need for websocket endpoints that integrate seamlessly to backend services and events
- A proxy for REST Consumers to interact with asynchronous services
- A growing team of Ml-Ops and Datascience engineers who'd like to deploy their models as services
- Require service gateway that bridges cloud VPNs and on-premise networks
- Wrap an existing or legacy service to benefit from any of the above

If any of these are of interest to you, NuroPb is worth considering.



1.1 Quickstart

The best way to get started is to look through the *examples*. These quick fire steps are the minimum to get going.:

- Install Python >= 3.10
- An accessible RabbitMQ >= 3.8.0 + Management Plugin
- Install the nuropb package, `pip install nuropb`

Run this code block to see a client and service running in the same Python module.

```
import logging
from typing import Any, Dict
from uuid import uuid4
import asyncio

from nuropb.contexts.context_manager import NuropbContextManager
from nuropb.contexts.context_manager_decorator import nuropb_context
from nuropb.contexts.describe import publish_to_mesh
from nuropb.rmq_api import RMQAPI

logger = logging.getLogger("nuropb-all-in-one")


def get_claims_from_token(bearer_token: str) -> Dict[str, Any] | None:
  """ This is a stub for the required implementation of validating and decoding the␣
↪bearer token
  """
  _ = bearer_token
  return {
    "sub": "test_user",
    "user_id": "test_user",
    "scope": "openid, profile",
    "roles": "user, admin",
  }


class QuickExampleService:
  _service_name = "quick-example"
  _instance_id = uuid4().hex

  @nuropb_context
  @publish_to_mesh(authorize_func=get_claims_from_token)
  def test_requires_user_claims(self, ctx, **kwargs: Any) -> str:
    logger.info("test_requires_user_claims called")
    assert isinstance(ctx, NuropbContextManager)
    return f"hello {ctx.user_claims['user_id']}"

  def test_method(self, param1, param2: Dict[str, Any]) -> Dict[str, Any]:
    logger.info("test_method called")
    _ = self
    return {
      "param1": param1,
      "param2": param2,
```

(continues on next page)

(continued from previous page)

```
      "reply": "response from test_method",
    }


async def main():
  logging.info("All in one example done")
  amqp_url = "amqp://guest:guest@localhost:5672/nuropb-example"
  service_instance = QuickExampleService()
  transport_settings = {
    "rpc_bindings": [service_instance._service_name],
  }
  service_api = RMQAPI(
    service_instance=service_instance,
    service_name=service_instance._service_name,
    instance_id=service_instance._instance_id,
    amqp_url=amqp_url,
    transport_settings=transport_settings,
  )
  await service_api.connect()
  logger.info("Service Ready")

  client_api = RMQAPI(
    amqp_url=amqp_url,
  )
  await client_api.connect()
  logger.info("Client connected")

  context = {
    "Authorization": "Bearer 1234567890",
  }
  response = await client_api.request(
    service="quick-example",
    method="test_requires_user_claims",
    params={},
    context=context,
  )
  logger.info(f"Response: {response}")

  response = await client_api.request(
    service="quick-example",
    method="test_method",
    params={
      "param1": "value1",
      "param2": {
        "param2a": "value2a",
      }
    },
    context={},
  )
  logger.info(f"Response: {response}")

  await client_api.disconnect()
```

(continues on next page)

(continued from previous page)

```
  await service_api.disconnect()

  logging.info("All in one example done")


if __name__ == "__main__":
  log_format = (
    "%(levelname).1s %(asctime)s %(name) -25s %(funcName) "
    "-35s %(lineno) -5d: %(message)s"
  )
  logging.basicConfig(level=logging.INFO, format=log_format)
  logging.getLogger("pika").setLevel(logging.WARNING)
  logging.getLogger("etcd3").setLevel(logging.WARNING)
  logging.getLogger("urllib3").setLevel(logging.WARNING)
  asyncio.run(main())
```

1.2 Examples

The example code and setup instructions are intended to help get going quickly and to demonstrate concepts applied by NuroPb. These examples are also used by the project's developers with integration testing and to validate designs and general improvements.

1.2.1 Demonstrated Concepts

- Connecting a service and connect to services for making rpc / request-response calls
- A very basic service with nuropb
- more complexity such as:
  - Contexts and context propagation
  - Authorisation
  - Point-to-point encryption

1.2.2 Prerequisites

Note:

- Has been tested and developed on macOS, Windows 10 and various Linux distros
- Standalone, VSI, Docker or Kubernetes friendly
- RabbitMQ only, no other database or infrustructure required

---

- Python >= 3.10
  - Development and testing on 3.11
- RabbitMQ >= 3.8.0 + Management Plugin

  - Likely work on earlier versions, not tested
- Python packages:
  - Tornado >= 6.3.0 (likely to work of earlier versions of 6.x, not tested)
  - Pika >= 1.2.0

1.2.3 Environment Setup

Install RabbitMQ in any fashion you like, it's quick and easy using Docker.

Assuming you are able to run a docker container, here is an example of running RabbitMQ.

```
# Update as needed, the docker ip address which in many cases is `localhost`. This will
# also be the RabbitMQ host address used by the examples.
export DOCKER_HOST_NAME=localhost

# run the RabbitMQ docker image with management plugin, exposing the amqp and management
# ports
docker run -d --name nuropb-rabbitmq -p 5672:5672 -p 15672:15672 rabbitmq:3-management
```

You can either install nuropb from PYPI, copy the examples from github and run them locally or clone the repo and setup the repo:

```
pip install nuropb
# alternatively, if using poetry
poetry add nuropb
```

For cloning the repo and running the examples locally, follow these steps:

Note: This is where in addition to having Python >= 3.10, you will also require the `poetry` package manager insatlled.
`pip install poetry`

```
git clone https://github.com/robertbetts/nuropb.git
cd nuropb
poetry install
```

1.2.4 Running an Example

The next step is to initialize the nuropb service mesh configuration in RabbitMQ. This can be performed by running the *scripted_mesh_setup.py* and is required to be run before trying *server_basic.py*. With the example *server.py*, the setup of the service mesh configuration is done automatically.

```
# Check that the RabbitMQ container is running and in the code , that the variables␣
↪`amqp_url` and `rmq_api_url`
# are correct. The default values are `amqp://guest:guest@localhost:5672/nuropb-example`␣
↪and
# `http://guest:guest@localhost:15672/api/` respectively.

poetry run python examples/scripted_mesh_setup.py
```

The *server_basic.py* example is for reference mainly, there are no requests from *client.py*. Add them at your pleasure.

Finally, run the client example.

```
poetry run python examples/client.py
```

all_in_one.py is a single file example of a client and server running in the same python file. It also demonstrates the use of the nuropb_contextandpublish_to_mesh` context manager decorators

And there you are. Let us know what you think! All feedback is welcome.

1.3 Discovery and Context Management

1.3.1 NuroPb Context Manager and Decorator

When a service instance method is decorated with @nuropb_context, a NuropbContextManager instance will be injected into the method, and usually into an argument named ctx. This context manager is callable via the `with` statement. Once the context manager has exited, it is considered done and immutable.

@nuropb_context

```
def nuropb_context(
    original_method=None,
    *,
    context_parameter: Optional[str] = "ctx",
    suppress_exceptions: Optional[bool] = False,
    authorise_key: Optional[str] = None,
    authorise_func: Optional[callable] = None,
) -> Any:
    """
    :param context_parameter: str, (ctx) alternative context argument name
    :param suppress_exceptions: bool, (True), if False then exceptions will be raised␣
↪during `with ctx:`
    """
```

This decorator function injects a NuropbContext instance into a method that has ctx:NuropbContext as an argument. The ctx parameter of the decorated method is hidden from the method's signature visible on the service mesh.

NOTE: This decorator is intended only for class methods, using it with functions will have unexpected results and is likely to result in either and compile time or runtime exception.

This decorator if used in conjunction with the @publish_to_mesh decorator, must be applied after.

```
    @nuropb_context
    @publish_to_mesh(authorise_func=get_claims_from_token, requires_encryption=True)
    def test_requires_encryption(self, ctx: NuropbContextManager, **kwargs: Any) -> Any:
        ...
```

The name of the ctx parameter can be changed by entering and alternative name using the context_parameter argument.

```
    @nuropb_context
    @publish_to_mesh(authorise_func=get_claims_from_token, requires_encryption=True,␣
↪context_parameter="myctx")
    def test_requires_encryption(self, myctx: NuropbContextManager, **kwargs: Any) ->␣
↪Any:
        ...
```

Any caller of class_instance.method(ctx=ctx) can either pass a NuropbContext instance or a dict. If a dict is passed, a NuropbContext instance will be created from the dict and injected into the method.

1.3.2 Service Mesh Configuration Decorator

When wrapping a class with the nuropb service mesh api, all public methods are made available to all participants connected to the service mesh. Any participant can request a service specification from any service, and will receive a json object describing the service and its methods.

@nuropb_context

```
def publish_to_mesh(
        original_method: Optional[Callable[..., Any]] = None,
        *,
        hide_method: Optional[bool] = False,
        authorize_func: Optional[AuthorizeFunc] = None,
        context_token_key: Optional[str] = "Authorization",
        requires_encryption: Optional[bool] = False,
        description: Optional[str] = None,
) -> Any:
    """
    :param hide_method: bool, (False) if True then the method will not be published to␣
↪the service mesh
    :param authorize_func: callable(token: str) -> dict
    :param context_token_key: str, ("Authorization"), incoming request context key␣
↪containing the bearer token
    :param requires_encryption: bool, (False) if True then the service mesh expect and␣
↪encrypted payload
    :param description: str, if present then override the methods doc string
    """
```

Example Mesh Service Describe Information:

```
{
    "service_name": "order_management_service",
    "service_version": "0.1.0",
    "description": "A service the manages orders",
    "warnings": [],
    "encrypted_methods": [],
    "public_key": None,
    "methods": [
        {
            "name": "create_order",
            "description": (
```

(continues on next page)

(continued from previous page)

```
                "Create an order for a given security, quantity, side and order_type.␣
↪Order price input is dependant "
                "on order_type and if a date is not given the current date is used,␣
↪dates prior to today will be "
                "rejected by the API."
            ),
            "parameters": {
                "type": "object",
                "properties": {
                    "account": {
                        "type": "string",
                        "description": "The account into which the executed trade will␣
↪be booked",
                    },
                    "security": {
                        "type": "string",
                        "description": "the security to be traded",
                    },
                    "status": {
                        "type": "string",
                        "enum": ["Open", "Filled", "Cancelled"],
                        "description": "The status of the order.",
                    },
                    "quantity": {
                        "type": "integer",
                        "description": "the quantity to be traded",
                    },
                    "side": {
                        "type": "string",
                        "enum": ["Buy", "Sell"],
                        "description": "The side of the order",
                    },
                    "order_type": {
                        "type": "string",
                        "enum": ["Market", "Limit", "Stop"],
                        "description": "The order type",
                    },
                    "price": {
                        "type": "number",
                        "description": "The price of the order",
                    },
                    "time_in_force": {
                        "type": "string",
                        "enum": ["Day", "GTC", "FOK", "IOC"],
                        "description": "The time in force of the order",
                    },
                    "stop_price": {
                        "type": "number",
                        "description": "The stop price of the order",
                    },
                    "order_date": {
                        "type": "string",
```

(continues on next page)

(continued from previous page)

```
                        "format": "date",
                        "description": "The date of the order",
                    },
                },
                "required": ["account", "security", "quantity", "side"]
            }
        }
    ]
}
```

1.4 Serialization and Encryption

Out of the box NuroPb transmits JSON encoded message payloads over the AMQP protocol. There are plans to support other encodings such as Protocol Buffers and m.a.y.b.e Avro.

1.4.1 JSON

JSON encoding is the default encoding for NuroPb. It's a handy human-readable encoding and is easy to debug, however it's also relatively verbose compares to other binary approaches such as ProtoBuf, pickle etc. Take a look *nuropb/interface.py* for type specifications on message structure as this is describes what is being encoded.

1.4.2 Encrypted Payloads

This is the first extension to the NuroPb payload serialisation, and ahead of Protocol Buffers other serialisation formats. After a serialised payload produces a byte stream, it can then be encrypted.

It is good practice to transmit any network traffic over TLS, however this only ensures a level of protection over the network between point a and point b. As the payload is passed through Gateways, Message Brokers and other network and application infrastructure, it is possible for the payload to be inspected or modified, but most commonly it can be logged. This is entirely out of the control of the NuroPb protocol and the application owner or developer.

In exceeding the requirements of Data Privacy and Data Protection, it is important to ensure that the visibility and integrity of data be accounted for at all times. This is the primary purpose of the encrypted payload capability.

NuroPb uses hybrid asymmetric and symmetric key approach to encrypt request and response payloads.

High Level Overview

Where Nuropb enabled service, requires encryption for one or any of its methods is started, it instantiates a cryptographic private key. The public key is shared with all other services and service mesh consumers. A message sender will encrypy the message payload with a Fernet symmetric key. The symmetric key is then encrypted with the public key of the receiver and is sent as part of the encrypted payload. The message receiver will then decrypt the symmetric key with its private key in order to decrypt the payload.

A response message is encrypted with the symmetric key from the request and the encrypted payload sent back to the sender. On receipt of the response, the original sender will decrypt the response message with the symmetric key used to encrypt the original message payload and continue processing the response.

Encryption Algorithms

- The encryption algorithm uses RSA private and public keys. The RSA algorithm is used to encrypt the symmetric key after it was used to encrypt the message payload.
- The Fernet algorithm is used for the symmetric key generation and payload encryption.

Note: The recipe for this approach is prompted by an article from Igor Filatov on Medium.com. The article can be found here: https://medium.com/@igorfilatov/hybrid-encryption-in-python-3e408c73970c

---

1.5 Background

1.5.1 Where does the name originate from?

NuroPb is a contraction of the term nervous [system] and the scientific symbol for Lead and its association with plumbing. So then NuroPb, the plumbing, routing and communications for connected services.

1.5.2 History, Pattern and Approach

NuroPb is a pattern and approach that supports event driven and service mesh engineering requirements. The early roots evolved in the 2000's, and during the 2010's the pattern was used to drive hedge funds, startups, banks and crypto and blockchain ventures. It's core development is in Python, and the pattern has been used alongside Java, JavasScript and GoLang, Unfortunately those efforts are copyrighted. Any platform with support for RabbitMQ should be able to implement the pattern and exist on the same mesh as Python Services.

RabbitMQ is the underlying message broker for the NuroPb service mesh library. Various message brokers and brokerless tools and approaches have been tried with the nuropb pattern. Some of these are Kafka, MQSeries and ZeroMQ. RabbitMQ's AMPQ routing capabilities, low maintenance and robustness have proved the test of time. With RabbitMQ's release of the streams feature, and offering message/logs streaming capabilities, there are new and interesting opportunities all on a single platform.

Why not Kafka? Kafka is a great tool, but it's not a message broker. It's a distributed log stream and probably one of the best available. There are many use cases where NuroPb + RabbitMQ would integrate very nicely alongside Kafka. Especially inter-process rpc and orderless event driven flows that are orchestrated by NuroPb and ordered log/event streaming over Kafka. Kafka is also been used as to ingest all nuropb traffic for auditing and tracing.

1.6 API Reference

This page contains auto-generated API reference documentation[1].

[1] Created with sphinx-autodoc2

1.6.1 examples

Submodules

examples.client

Module Contents

Functions

perform_request
perform_command
publish_event
main

Data

logger

API

examples.client.**logger**
None

async examples.client.**perform_request**(*api:* nuropb.rmq_api.RMQAPI, *encryption_test: bool = False*)

async examples.client.**perform_command**(*api:* nuropb.rmq_api.RMQAPI)

async examples.client.**publish_event**(*api:* nuropb.rmq_api.RMQAPI)

async examples.client.**main**()

examples.server_basic

Module Contents

Classes

BasicExampleService

Functions

main

Data

logger

API

examples.server_basic.**logger**

None

class examples.server_basic.**BasicExampleService**

_service_name

'basic-example'

_instance_id

None

async test_async_method(***params: Any*) → str

test_method(*param1*, *param2: Dict[str, Any]*) → Dict[str, Any]

async examples.server_basic.**main**()

examples.service_example

Module Contents

Classes

ServiceExample

Data

logger

API

examples.service_example.**logger**

None

class examples.service_example.**ServiceExample**(*service_name: str*, *instance_id: str*, *private_key: cryptography.hazmat.primitives.asymmetric.rsa.RSAPrivateKey*)

Initialization

_service_name: str

None

_instance_id: str

None

_private_key: cryptography.hazmat.primitives.asymmetric.rsa.RSAPrivateKey

None

_method_call_count: int

None

classmethod _handle_event_(*topic: str*, *event: dict*, *target: list[str] | None = None*, *context: dict | None = None*, *trace_id: str | None = None*) → None

test_method(***kwargs*) → str

test_encrypt_method(***kwargs*) → str

test_exception_method(***kwargs*) → str

async test_async_method(***kwargs*) → str

async async_method(***kwargs*) → int

sync_method(***kwargs*) → int

method_with_exception(***kwargs*) → None

async_method_with_exception(***kwargs*) → None

method_with_nuropb_exception(***kwargs*) → None

examples.scripted_mesh_setup

Module Contents

Classes

MeshServiceInstance	Helper service instance class used purely for the virtual host and exchanges configured in RabbitMQ. The

Functions

main

Data

logger	This script is used to setup the RabbitMQ exchanges, queues, and bindings for the NuroPb examples. A running Rab
amqp_url	
rmq_api_url	

API

examples.scripted_mesh_setup.**logger**

None

This script is used to setup the RabbitMQ exchanges, queues, and bindings for the NuroPb examples. A running RabbitMQ instance with the management plugin is required.

The default login credentials are assumed. If you have changed the default login credentials for RabbitMQ, you will need to update the amqp_url and rmq_api_url variables below.

examples.scripted_mesh_setup.**amqp_url**

'amqp://guest:guest@localhost:5672/nuropb-example'

examples.scripted_mesh_setup.**rmq_api_url**

'http://guest:guest@localhost:15672/api'

class examples.scripted_mesh_setup.**MeshServiceInstance**

Helper service instance class used purely for the virtual host and exchanges configured in RabbitMQ. The service queue and dead letter created for this instance during the setup can be deleted.

_service_name

'mesh-setup'

_instance_id

None

async examples.scripted_mesh_setup.**main**()

examples.all_in_one

Module Contents

Classes

QuickExampleService

Functions

get_claims_from_token	This stub is for the implementation to complete for the validation and decoding of the bearer token
main	

Data

logger

API

examples.all_in_one.**logger**

None

examples.all_in_one.**get_claims_from_token**(*bearer_token: str*) → Dict[str, Any] | None

This stub is for the implementation to complete for the validation and decoding of the bearer token

class examples.all_in_one.**QuickExampleService**

_service_name

'quick-example'

_instance_id

None

test_requires_user_claims(*ctx*, ***kwargs: Any*) → str

test_method(*param1*, *param2: Dict[str, Any]*) → Dict[str, Any]

async examples.all_in_one.**main**()

examples.server

Module Contents

Functions

main	load private_key and create one if it done not exist

Data

logger

API

examples.server.**logger**

None

async examples.server.**main**()

load private_key and create one if it done not exist

1.6.2 nuropb

Subpackages

nuropb.contexts

Submodules

nuropb.contexts.describe

Module Contents

Functions

method_visible_on_mesh	This function checks if a method has been decorated with @publish_to_mesh
publish_to_mesh	Decorator to expose class methods to the service mesh
describe_service	Returns a description of the class methods that will be exposed to the service mesh

Data

logger
AuthorizeFunc

API

nuropb.contexts.describe.**logger**

None

nuropb.contexts.describe.**AuthorizeFunc**

None

nuropb.contexts.describe.**method_visible_on_mesh**(*method: Callable[..., Any]*) → bool

This function checks if a method has been decorated with @publish_to_mesh

Parameters
method – callable

Returns
bool

nuropb.contexts.describe.**publish_to_mesh**(*original_method: Optional[Callable[..., Any]] = None*, ***, *hide_method: Optional[bool] = False*, *authorize_func: Optional[nuropb.contexts.describe.AuthorizeFunc] = None*, *context_token_key: Optional[str] = 'Authorization'*, *requires_encryption: Optional[bool] = False*, *description: Optional[str] = None*) → Any

Decorator to expose class methods to the service mesh

When a service instance is connected to a service mesh via the service mesh client, all the standard public methods of the service instance is available to the service mesh. Methods that start with underscore will always remain hidden. methods that are explicitly marked as hidden by publish_to_mesh will also not be published.

When and authorise_func is specified, this function will be called with the contents of context[context_token_key]. if the token validation is unsuccessful, then a failed authorization exception is raised. If successful then ctx.user_claims is populated with claims attached to the token.

When requires_encryption is True, the service mesh will encrypt the payload of the service message request and response. It is the responsibility of the process making the request to ensure that it has the target service's public key.

As illustrated by the example below, @nuropb_context must always be on top of @publish_to_mesh when both decorators are used.

```
@nuropb_context
@publish_to_mesh(context_token_key="Authorization", authorise_func=authorise_token)
def hello_requires_auth(...)
```

Parameters

- **original_method** –
- **hide_method** – bool, (False) if True then the method will not be published to the service mesh
- **authorize_func** – callable(token: str) -> dict
- **context_token_key** – str, ("Authorization"), incoming request context key containing the bearer token

- **requires_encryption** – bool, (False) if True then the service mesh expect and encrypted payload
- **description** – str, if present then override the methods doc string

Returns

nuropb.contexts.describe.**describe_service**(*class_instance: object*) → Dict[str, Any] | None

Returns a description of the class methods that will be exposed to the service mesh

nuropb.contexts.context_manager_decorator

Module Contents

Functions

method_requires_nuropb_context	This function checks if a method has been decorated with @nuropb_context
nuropb_context	This decorator function injects a NuropbContext instance into a method that has ctx:Nuropb

API

nuropb.contexts.context_manager_decorator.**method_requires_nuropb_context**(*method: Callable[..., Any]*) → bool

This function checks if a method has been decorated with @nuropb_context

Parameters
method – Callable

Returns
bool

nuropb.contexts.context_manager_decorator.**nuropb_context**(*original_method: Optional[Callable[..., Any]] = None*, *, *context_parameter: str = 'ctx'*, *suppress_exceptions: bool = False*) → Any

This decorator function injects a NuropbContext instance into a method that has ctx:NuropbContext as an argument. The ctx parameter of the decorated method is hidden from the method's signature visible on the service mesh.

The name of the ctx parameter can be changed by passing a string to the context_parameter argument.

The caller of class_instance.method(ctx=ctx) can either pass a NuropbContext instance or a dict. If a dict is passed, a NuropbContext instance will be created from the dict.

NOTE This decorator is only for with class methods, using with functions will have unexpected results and is likely to raise a TypeException

As illustrated by the example below, @nuropb_context must always be on top of @publish_to_mesh when both decorators are used.

```
@nuropb_context
@publish_to_mesh(context_token_key="Authorization", authorize_func=authorise_token)
def hello_requires_auth(...)
```

Parameters

- **original_method** – the method to be decorated
- **context_parameter** – str, (ctx) alternative context argument name
- **suppress_exceptions** – bool, (True), if False then exceptions will be raised during `with ctx:`

Returns

a decorated method

nuropb.contexts.service_handlers

This module provides features that handle the execution of nuropb service messages: Requests, Commands and Events.

Attempt is made to make the code agnostic to the underlying transport.

ROADMAP Features and Considerations (in no particular order):

- method parameter validation
- post ack commit or rollback configured by the implementation

Module Contents

Functions

verbose	
error_dict_from_exception	Creates an error dict from an exception
create_transport_response_from_rmq_decode_exception	Creates a NuroPb response from an unsupported message receiv
create_transport_responses_from_exceptions	Creates a NuroPb response from an exceptions and also accomm
handle_execution_result	This function is called from the execute_request() to handle both
execute_request	Executes a transport request and call the message_complete_cal

Data

logger
_verbose

API

nuropb.contexts.service_handlers.**logger**

None

nuropb.contexts.service_handlers.**_verbose**

False

nuropb.contexts.service_handlers.**verbose**() → bool

nuropb.contexts.service_handlers.**error_dict_from_exception**(*exception: Exception | BaseException*) → Dict[str, str]

Creates an error dict from an exception

Parameters
exception –

Returns

nuropb.contexts.service_handlers.**create_transport_response_from_rmq_decode_exception**(*exception: Exception | BaseException*, *basic_deliver: pika.spec.Basic.Deliver*, *properties: pika.spec.BasicProperti* → Tuple[nuropb.interface.Ac list[*nuropb.interface.Tra*

Creates a NuroPb response from an unsupported message received over RabbitMQ

nuropb.contexts.service_handlers.**create_transport_responses_from_exceptions**(*service_message:* nuropb.interface.TransportServiceP *exception: Exception | BaseException*) → Tuple[nuropb.interface.AcknowledgeA list[*nuropb.interface.TransportResp*

Creates a NuroPb response from an exceptions and also accommodates special cases like NuropbCallAgain and NuropbSuccess

Parameters

- **service_message** –
- **exception** –

Returns

nuropb.contexts.service_handlers.**handle_execution_result**(*service_message:* nuropb.interface.TransportServicePayload, *result: Any*, *message_complete_callback: nuropb.interface.MessageCompleteFunction*) → None

This function is called from the execute_request() to handle both synchronous and asynchronous results

With standard implementation message_complete_callback is defined in the transport layer.

- For service messages transport.on_service_message_complete() is used
- For response messages transport.on_response_message_complete() is used.

#FUTURE: There is consideration of how to pass a context through this flow to until the transport acknowledgement has completed. This is to allow for the possibility of a post ack commit or rollback. It also allows to more flexible handling for events raised during requests, commands and incoming events

Parameters

- **service_message** –
- **result** –
- **message_complete_callback** –

Returns

nuropb.contexts.service_handlers.**execute_request**(*service_instance: object*, *service_message:* nuropb.interface.TransportServicePayload, *message_complete_callback:* *nuropb.interface.MessageCompleteFunction*) → None

Executes a transport request and call the message_complete_callback with the result

PLEASE NOTE: At first glance awaitable nature of the result_future is not obvious from the code. read the comments in transformed_async_future() to understand how the result_future is handled before making any changes.

Parameters

- **service_instance** – object
- **service_message** – TransportServicePayload
- **message_complete_callback** – MessageCompleteFunction

Returns
None

nuropb.contexts.context_manager

Module Contents

Classes

NuropbContextManager	This class is a context manager that's used to manage a transaction's context relating to an incoming nuro

Data

logger
_test_token_cache
_test_user_id_cache

API

nuropb.contexts.context_manager.**logger**

None

nuropb.contexts.context_manager.**_test_token_cache: Dict[str, Any]**

None

nuropb.contexts.context_manager.**_test_user_id_cache: Dict[str, Any]**

None

class nuropb.contexts.context_manager.**NuropbContextManager**(*context: Dict[str, Any]*, *suppress_exceptions: Optional[bool] = True*)

This class is a context manager that's used to manage a transaction's context relating to an incoming nuropb service message. When a class instance's method is decorated with the nuropb_context decorator, the context manager is instantiated and injected into the method as a ctx parameter.

The nuropb context manager is both a sync and async context manager. Events can be added to the context manager and will be sent to the service mesh when the context manager successfully exits. If an exception is raised while the context manager is in scope, the exception is recorded and the transaction in context is considered to have failed. Any events added to the context manager are discarded.

Initialization

_suppress_exceptions: bool

None

_nuropb_payload: Dict[str, Any] | None

None

_context: Dict[str, Any]

None

_user_claims: Dict[str, Any] | None

None

_events: List[Dict[str, Any]]

None

_exc_type: Type[BaseException] | None

None

_exec_value: BaseException | None

None

_exc_tb: types.TracebackType | None

None

_started: bool

None

_done: bool

None

property context: Dict[str, Any]

property user_claims: Dict[str, Any] | None

property events: List[Dict[str, Any]]

property error: Dict[str, Any] | None

add_event(*event: Dict[str, Any]*) → None

Add an event to the context manager. The event will be sent to the service mesh when the context manager exits successfully.

Event format

- "topic": "test_topic",
- "event": "test_event_payload",
- "context": {}

Parameters
event –

Returns

_handle_context_exit(*exc_type: Type[BaseException] | None*, *exc_value: BaseException | None*, *exc_tb: types.TracebackType | None*) → bool

This method is for customising the behaviour when a context manager exits. It has the same signature as **exit** or **aexit**.

If an exception was raised while the context manager was running, the exception information is recorded and content transaction is considered to have failed. If any events were added to the context manager, they are discarded.

__enter__() → Any

This method is called when entering a context manager with a with statement

async __aenter__() → Any

__exit__(*exc_type: Type[BaseException] | None*, *exc_value: BaseException | None*, *exc_tb: types.TracebackType | None*) → bool | None

async __aexit__(*exc_type: Type[BaseException] | None*, *exc_value: BaseException | None*, *exc_tb: types.TracebackType | None*) → bool | None

nuropb.encodings

Submodules

nuropb.encodings.json_serialisation

This module provides entire nuropb package with json serialisation logic and features

Module Contents

Classes

NuropbEncoder	*NOTE 1* This class must be kept in step with the function to_json_compatible, above. Next, read NOTE 2.
JsonSerializor	Serializes and deserializes nuropb payloads to and from JSON format.

Functions

to_json_compatible	Returns a json compatible value for obj, if obj is not a native json type, then return a string representation.
to_json	Returns a json string representation of the input object, if not a native json type

API

nuropb.encodings.json_serialisation.**to_json_compatible**(*obj: Any*, *recursive: bool = True*, *max_depth: int = 4*) → Any

Returns a json compatible value for obj, if obj is not a native json type, then return a string representation.

NOTE 1 This function must be kept in step with the custom json encoder, NuropbEncoder, below. Next, read NOTE 2.

NOTE 2 This function does not exactly follow the structure of the custom json encoder, NuropbEncoder, below. Difference is that the json library implements its own object traversal logic. In this function it's required to be done explicitly.

datetime.datetime: isoformat() + "Z". if there's timezone info, the datetime is converted to utc. if there is no timezone info, the datetime is assumed to be utc.

Parameters

- **obj** – Any
- **recursive** – bool, whether to recursively convert obj to json compatible types
- **max_depth** – int, the maximum depth to recurse

Returns

str, or other json compatible type

class nuropb.encodings.json_serialisation.**NuropbEncoder**(*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *default=None*)

Bases: `json.JSONEncoder`

NOTE 1 This class must be kept in step with the function to_json_compatible, above. Next, read NOTE 2.

NOTE 2 This class does not exactly follow the structure of the function, to_json_compatible, above. Difference is that the json library implements its own object traversal logic. In the function it's required to be done explicitly.

Initialization

Constructor for JSONEncoder, with sensible defaults.

If skipkeys is false, then it is a TypeError to attempt encoding of keys that are not str, int, float or None. If skipkeys is True, such items are simply skipped.

If ensure_ascii is true, the output is guaranteed to be str objects with all incoming non-ASCII characters escaped. If ensure_ascii is false, the output can contain non-ASCII characters.

If check_circular is true, then lists, dicts, and custom encoded objects will be checked for circular references during encoding to prevent an infinite recursion (which would cause an RecursionError). Otherwise, no such check takes place.

If allow_nan is true, then NaN, Infinity, and -Infinity will be encoded as such. This behavior is not JSON specification compliant, but is consistent with most JavaScript based encoders and decoders. Otherwise, it will be a ValueError to encode such floats.

If sort_keys is true, then the output of dictionaries will be sorted by key; this is useful for regression tests to ensure that JSON serializations can be compared on a day-to-day basis.

If indent is a non-negative integer, then JSON array elements and object members will be pretty-printed with that indent level. An indent level of 0 will only insert newlines. None is the most compact representation.

If specified, separators should be an (item_separator, key_separator) tuple. The default is (', ', ': ') if *indent* is `None` and (',', ': ') otherwise. To get the most compact JSON representation, you should specify (',', ':') to eliminate whitespace.

If specified, default is a function that gets called for objects that can't otherwise be serialized. It should return a JSON encodable version of the object or raise a `TypeError`.

default(*obj: Any*) → Any

`nuropb.encodings.json_serialisation.`**to_json**(*obj: Any*) → str

Returns a json string representation of the input object, if not a native json type

class `nuropb.encodings.json_serialisation.`**JsonSerializor**

Bases: `object`

Serializes and deserializes nuropb payloads to and from JSON format.

Initialization

Initializes a new JsonSerializor instance.

_encryption_keys: Dict[str, Any]

None

encryption keys related to a given correlation_id

encode(*payload: Any*) → str

Encodes a nuropb encoded payload to JSON.

Parameters
payload – Any, The encoded payload to encode.

Returns
str, The JSON-encoded encoded payload.

decode(*json_payload: str*) → Any

Decodes a JSON-encoded nuropb encoded payload.

Parameters

json_payload – str, The JSON-encoded encoded payload to decode.

Returns

Any, The decoded encoded payload.

TODO: Re-check the serialised datetime, date, time and timedelta formats. Look for standards.

nuropb.encodings.encryption

Module Contents

Classes

Encryptor	

Functions

encrypt_payload	Encrypt encoded payload with a key
decrypt_payload	Decrypt encoded payload with a key
encrypt_key	Encrypt a symmetric key with an RSA public key
decrypt_key	Decrypt a symmetric key encrypted with an RSA private key

API

nuropb.encodings.encryption.**encrypt_payload**(*payload: str | bytes*, *key: str | bytes*) → bytes

Encrypt encoded payload with a key

Parameters

- **payload** – str | bytes
- **key** – str

Returns

bytes

nuropb.encodings.encryption.**decrypt_payload**(*encrypted_payload: str | bytes*, *key: str | bytes*) → bytes

Decrypt encoded payload with a key

Parameters

- **encrypted_payload** – str | bytes
- **key** – str

Returns

bytes

nuropb.encodings.encryption.**encrypt_key**(*symmetric_key: bytes*, *public_key: cryptography.hazmat.primitives.asymmetric.rsa.RSAPublicKey*) → bytes

Encrypt a symmetric key with an RSA public key

Parameters

- **symmetric_key** – bytes
- **public_key** – rsa.RSAPublicKey

Returns
bytes, encrypted symmetric key

nuropb.encodings.encryption.**decrypt_key**(*encrypted_symmetric_key: bytes*, *private_key: cryptography.hazmat.primitives.asymmetric.rsa.RSAPrivateKey*) → bytes

Decrypt a symmetric key encrypted with an RSA private key

Parameters

- **encrypted_symmetric_key** – bytes
- **private_key** – rsa.RSAPrivateKey

Returns
bytes, decrypted symmetric key

class nuropb.encodings.encryption.**Encryptor**(*service_name: Optional[str] = None*, *private_key: Optional[cryptography.hazmat.primitives.asymmetric.rsa.RSAPrivateKey] = None*)

Initialization

_service_name: str | None

None

_private_key: cryptography.hazmat.primitives.asymmetric.rsa.RSAPrivateKey | None

None

_service_public_keys: Dict[str, cryptography.hazmat.primitives.asymmetric.rsa.RSAPublicKey]

None

_correlation_id_symmetric_keys: Dict[str, bytes]

None

classmethod new_symmetric_key() → bytes

Generate a new symmetric key

Returns
bytes

add_public_key(*service_name: str*, *public_key: cryptography.hazmat.primitives.asymmetric.rsa.RSAPublicKey*) → None

Add a public key for a service

Parameters

- **service_name** – str

- **public_key** – rsa.RSAPublicKey

get_public_key(*service_name: str*) → cryptography.hazmat.primitives.asymmetric.rsa.RSAPublicKey | None

Get a public key for a service

Parameters
service_name – str

Returns
rsa.RSAPublicKey

has_public_key(*service_name: str*) → bool

Check if a service has a public key

Parameters
service_name – str

Returns
bool

encrypt_payload(*payload: bytes*, *correlation_id: str*, *service_name: Optional[str | None] = None*) → bytes

Encrypt a payload with a symmetric key

Modes:

1. Sending a payload to a service (Encrypt)
2. Sending a response payload from a service (Encrypt)

Parameters

- **payload** –
- **correlation_id** –
- **service_name** –

Returns
bytes

decrypt_payload(*payload: bytes*, *correlation_id: str*) → bytes

Encrypt a payload with a symmetric key

Modes: 2. Receiving a response payload from a service (Decrypt) 3. Receiving a rpc payload (Decrypt)

Parameters

- **payload** –
- **correlation_id** –

Returns

nuropb.encodings.serializer

Module Contents

Functions

get_serializor	Returns a serializor object for the specified encoded payload type
encode_payload	**param public_key** **param payload** **param payload_type** "json" **param public_key** rsa.PublicKey **return** a json bytes string imputed encoded payload
decode_payload	**param encoded_payload** **param payload_type** "json" **return** PayloadDict

Data

SerializorTypes

API

nuropb.encodings.serializor.**SerializorTypes**

None

nuropb.encodings.serializor.**get_serializor**(*payload_type: str = 'json'*) → nuropb.encodings.serializor.SerializorTypes

Returns a serializor object for the specified encoded payload type

Parameters
payload_type – "json"

Returns
a serializor object

nuropb.encodings.serializor.**encode_payload**(*payload: nuropb.interface.PayloadDict*, *payload_type: str = 'json'*, *public_key: cryptography.hazmat.primitives.asymmetric.rsa.RSAPublicKey = None*) → bytes

Parameters

- **public_key** –
- **payload** –
- **payload_type** – "json"
- **public_key** – rsa.PublicKey

Returns
a json bytes string imputed encoded payload

nuropb.encodings.serializor.**decode_payload**(*encoded_payload: bytes*, *payload_type: str = 'json'*) → nuropb.interface.PayloadDict

Parameters

- **encoded_payload** –
- **payload_type** – "json"

Returns
PayloadDict

nuropb.testing

Submodules

nuropb.testing.stubs

Module Contents

Classes

ServiceStub	
ServiceExample	

Functions

get_claims_from_token	This is a stub for the authorize_func that is used in the tests

Data

logger
IN_GITHUB_ACTIONS

API

nuropb.testing.stubs.**logger**

None

nuropb.testing.stubs.**IN_GITHUB_ACTIONS**

None

nuropb.testing.stubs.**get_claims_from_token**(*bearer_token: str*) → Dict[str, Any] | None

This is a stub for the authorize_func that is used in the tests

class nuropb.testing.stubs.**ServiceStub**(*service_name: str*, *instance_id: Optional[str] = None*, *private_key: Optional[cryptography.hazmat.primitives.asymmetric.rsa.RSAPrivateKey] = None*)

Initialization

_service_name: str

None

_instance_id: str

None

_private_key: cryptography.hazmat.primitives.asymmetric.rsa.RSAPrivateKey

None

property service_name: str

property instance_id: str

property private_key: cryptography.hazmat.primitives.asymmetric.rsa.RSAPrivateKey

class nuropb.testing.stubs.**ServiceExample**(**args: Any*, ***kwargs: Any*)

Bases: *nuropb.testing.stubs.ServiceStub*

Initialization

_method_call_count: int

None

_raise_call_again_error: bool

None

test_method(***kwargs: Any*) → str

async do_async_task(***kwargs: Any*) → str

async test_async_method(***kwargs: Any*) → str

test_success_error(***kwargs: Any*) → None

test_requires_user_claims(*ctx:* nuropb.contexts.context_manager.NuropbContextManager, ***kwargs: Any*) → Any

test_requires_encryption(*ctx:* nuropb.contexts.context_manager.NuropbContextManager, ***kwargs: Any*) → Any

test_call_again_error(***kwargs: Any*) → Dict[str, Any]

test_call_again_loop(***kwargs: Any*) → None

Submodules

nuropb.rmq_api

Module Contents

Classes

RMQAPI	The primary nuropb API.

Data

logger
verbose

API

nuropb.rmq_api.**logger**

None

nuropb.rmq_api.**verbose**

False

class nuropb.rmq_api.**RMQAPI**(*amqp_url: str | Dict[str, Any]*, *service_name: str | None = None*, *instance_id: str | None = None*, *service_instance: object | None = None*, *rpc_exchange: Optional[str] = None*, *events_exchange: Optional[str] = None*, *transport_settings: Optional[Dict[str, Any]] = None*)

Bases: *nuropb.interface.NuropbInterface*

The primary nuropb API.

When an existing transport initialised and connected, and a subsequent transport instance is connected with the same service_name and instance_id as the first, the broker will shut down the channel of subsequent instances when they attempt to configure their response queue. This is because the response queue is opened in exclusive mode. The exclusive mode is used to ensure that only one consumer (nuropb api connection) is consuming from the response queue.

Deliberately specifying a fixed instance_id, is a valid mechanism to ensure that a service can only run in single instance mode. This is useful for services that are not designed to be run in a distributed manner or where there is specific service configuration required.

Initialization

RMQAPI: A NuropbInterface implementation that uses RabbitMQ as the underlying transport.

Where exchange inputs are none, but they user present in transport_settings, then use the values from transport_settings

_mesh_name: str

None

_connection_name: str

None

_response_futures: Dict[str, nuropb.interface.ResultFutureResponsePayload]

None

_transport: *nuropb.rmq_transport.RMQTransport*

None

_rpc_exchange: str

None

_events_exchange: str

None

_service_instance: object | None

None

_default_ttl: int

None

_client_only: bool

None

_encryptor: *nuropb.encodings.encryption.Encryptor*

None

_service_discovery: Dict[str, Any]

None

_service_public_keys: Dict[str, Any]

None

classmethod _get_vhost(*amqp_url: str | Dict[str, Any]*) → str

_service_name

None

Is also a label for the api whether in client or service mode.

property service_name: str

service_name: returns the service_name of the underlying transport

property is_leader: bool

property client_only: bool

client_only: returns the client_only status of the underlying transport

property connected: bool

connected: returns the connection status of the underlying transport

Returns
bool

property transport: *nuropb.rmq_transport.RMQTransport*

transport: returns the underlying transport

Returns
RMQTransport

async connect() → None

connect: connects to the underlying transport

Returns
None

async disconnect() → None

disconnect: disconnects from the underlying transport

Returns
None

receive_transport_message(*service_message:* nuropb.interface.TransportServicePayload, *message_complete_callback: nuropb.interface.MessageCompleteFunction*, *metadata: Dict[str, Any]*) → None

receive_transport_message: handles a messages received from the transport layer. Both incoming service messages and response messages pass through this method.

Returns
None

classmethod _handle_immediate_request_error(*rpc_response: bool*, *payload:* nuropb.interface.RequestPayloadDict | nuropb.interface.ResponsePayloadDict, *error: Dict[str, Any] | BaseException*) → *nuropb.interface.ResponsePayloadDict*

async request(*service: str*, *method: str*, *params: Dict[str, Any]*, *context: Dict[str, Any]*, *ttl: Optional[int] = None*, *trace_id: Optional[str] = None*, *rpc_response: bool = True*, *encrypted: bool = False*) → Union[*nuropb.interface.ResponsePayloadDict*, Any]

Makes a rpc request for a method on a service mesh service and waits until the response is received.

Parameters

- **service** – str, The routing key on the rpc exchange to direct the request to the desired service request queue.
- **method** – str, the name of the api call / method on the service
- **params** – dict, The method input parameters
- **context** – dict The context of the request. This is used to pass information to the service manager and is not used by the transport. Example content includes: - user_id: str # a unique user identifier or token of the user that made the request - correlation_id: str # a unique identifier of the request used to correlate the response to the # request or trace the request over the network (e.g. uuid4 hex string) - service: str - method: str
- **ttl** – int optional expiry is the time in milliseconds that the message will be kept on the queue before being moved to the dead letter queue. If None, then the message expiry configured on the transport is used.
- **trace_id** – str optional an identifier to trace the request over the network (e.g. uuid4 hex string)
- **rpc_response** – bool optional if True (default), the actual response of the RPC call is returned and where there was an error, that is raised as an exception. Where rpc_response is a ResponsePayloadDict, is returned.
- **encrypted** – bool if True then the message will be encrypted in transit

Return ResponsePayloadDict | Any
representing the response from the requested service with any exceptions raised

command(*service: str*, *method: str*, *params: Dict[str, Any]*, *context: Dict[str, Any]*, *ttl: Optional[int] = None*, *trace_id: Optional[str] = None*, *encrypted: bool = False*) → None

command: sends a command to the target service. I.e. a targeted event. response is not expected and ignored.

Parameters

- **service** – the service name
- **method** – the method name
- **params** – the method arguments, these must be easily serializable to JSON
- **context** – additional information that represent the context in which the request is executed. The must be easily serializable to JSON.
- **ttl** – the time to live of the request in milliseconds. After this time and dependent on the underlying transport, it will not be consumed by the target or assumed by the requester to have failed with an undetermined state.

- **trace_id** – an identifier to trace the request over the network (e.g. uuid4 hex string)
- **encrypted** – bool, if True then the message will be encrypted in transit

Returns
None

publish_event(*topic: str*, *event: Dict[str, Any]*, *context: Dict[str, Any]*, *trace_id: Optional[str] = None*, *encrypted: bool = False*) → None

Broadcasts an event with the given topic.

Parameters

- **topic** – str, The routing key on the events exchange
- **event** – json-encodable Python Dict.
- **context** – dict, The context around gent generation, example content includes:
  - str user_id: # a unique user identifier or token of the user that made the request
  - str correlation_id: # a unique identifier of the request used to correlate the response # to the request # or trace the request over the network (e.g. an uuid4 hex string)
  - str service:
  - str method:
- **trace_id** – str optional an identifier to trace the request over the network (e.g. an uuid4 hex string)
- **encrypted** – bool, if True then the message will be encrypted in transit

async describe_service(*service_name: str*, *refresh: bool = False*) → Dict[str, Any] | None

describe_service: returns the service information for the given service_name, if it is not already cached or refresh is try then the service discovery is queried directly.

Parameters

- **service_name** – str
- **refresh** – bool

Returns
dict

async requires_encryption(*service_name: str*, *method_name: str*) → bool

requires_encryption: Queries the service discovery information for the service_name and returns True if encryption is required else False. none of encryption is not required.

Parameters

- **service_name** –
- **method_name** –

Returns
bool

async has_public_key(*service_name: str*) → bool

service_has_public_key: returns True if the service has a public key registered, else False

nuropb.service_runner

This module provides the runtime configuration for the RabbitMQ transport.

- NuroPb Service leader election
- RMQ Exchange and Queue configuration

A service Leader is elected using etcd. The leader is responsible for creating the RMQ Exchange and Queues, and binding the Queues to the Exchange. Due to the additional administrative responsibilities, the leader's message prefetch size should be smaller relative to the other service instances.

TODO: The service leader is also responsible for monitoring and managing overall service health.

- if the service queue is not draining fast enough, the leader will signal for additional instances to be started.
- if the service queue is draining too fast, the leader will signal for instances to be stopped.
- Connections to RMQ that are initiating service messages with high error rates will be terminated.
- The leader will also monitor the dead letter queue and take action if the dead letter queue is growing too fast.

NuroPb services instances wait for the elected leader to signal that the RMQ Exchange and Queues are ready before connecting and starting to consume messages from the RMQ Queues.

Module Contents

Classes

ServiceRunner	ServiceRunner represents the state of the service configuration.
ServiceContainer	

Data

logger
LEADER_KEY
LEASE_TTL
ContainerRunningState

API

nuropb.service_runner.**logger**

None

nuropb.service_runner.**LEADER_KEY**

'/leader'

nuropb.service_runner.**LEASE_TTL**

15

class nuropb.service_runner.**ServiceRunner**

ServiceRunner represents the state of the service configuration.

service_name: str

None

service_name: the name of the service.

leader_id: str

None

service_leader_id: the identifier of the service leader.

configured: bool

None

rmq_configured: indicates if the RMQ Exchange and Queues have been configured.

ready: bool

None

rmq_ready: indicates if the RMQ Exchange and Queues are ready to receive messages.

consume: bool

None

rmq_consume: indicates if all service instance should consume messages or not from the RMQ Queues.

hw_mark: int

None

trigger for when to start or stop service instances.

_etc_client: etcd3.Client

None

_etc_client: the etcd3 client used to communicate with the etcd3 server.

nuropb.service_runner.**ContainerRunningState**

None

class nuropb.service_runner.**ServiceContainer**(*rmq_api_url: str*, *instance:* nuropb.rmq_api.RMQAPI, *etcd_config: Optional[Dict[str, Any]] = None*)

Bases: *nuropb.service_runner.ServiceRunner*

_instance: *nuropb.rmq_api.RMQAPI*

None

_rmq_api_url: str

None

_service_name: str

None

_transport: *nuropb.rmq_transport.RMQTransport*

None

_running_state: nuropb.service_runner.ContainerRunningState

None

_rmq_config_ok: bool

None

_shutdown: bool

None

_is_leader: bool

None

_leader_reference: str

None

_etcd_config: Dict[str, Any]

None

_etcd_client: etcd3.Client | None

None

_etcd_lease: etcd3.stateful.lease.Lease | None

None

_etcd_watcher: etcd3.stateful.watch.Watcher | None

None

_etcd_prefix: str

None

_container_running_future: Awaitable[bool] | None

None

_container_shutdown_future: Awaitable[bool] | None

None

property running_state: nuropb.service_runner.ContainerRunningState

running_state: the current running state of the service container.

async init_etcd(*on_startup: bool = True*) → bool

init_etcd: initialises the etcd client and connection. Not a fan of the threading started here, when defining a watcher and calling runDaemon(). It would be way nicer to have a fully asynchronous etcd3 client. There is also threading used for the lease keepalive.

NOTES:

- this function will run until successful etcd connection is established, or until self._shutdown is set to True.

Parameters
on_startup – True, if the etcd3 client is being initialised at startup.

Returns
None

nominate_as_leader() → None

nominate_as_leader: nominates the service instance as the leader. if first to secure a lease on the leader key, then the instance is the leader. Otherwise, the instance is a follower.

Handling the following scenarios: On Startup: [No transition to leader or from leader required] * Can secure the leader key lease - startup: the first instance to start, no existing entry becomes the leader. * Cannot secure the leader key lease - startup: there is an existing leader entry, all services must wait until key's lease expires and can then compete to become the leader.

```
Post Startup: [Possible transition to leader or from leader is required]
* No change to self._is_leader and continue as before. - No transition required
* Can secure the leader key lease
    - running: the leader entry is reset to None, followers then compete to␣
↪become leader.
```

ETCD NOTES: it appears that a lease can only be associated with a key it that key was created with the lease. If the key is created without a lease, then the lease cannot be associated with the key.

Returns
None

update_etcd_service_property(*key: str*, *value: Any*) → bool

update_etcd_service_property: updates the etcd3 service property.

The use of this method is restricted to the service leader

Parameters

- **key** –
- **value** –

Returns

check_and_configure_rmq() → None

check_and_configure_rmq: checks that the RMQ Exchange and Queues are correctly configured. If not, then the NuroPb RMQ configuration for self.service_name is applied.

Returns
None

etcd_event_handler(*event: etcd3.stateful.watch.Event*) → None

etc_event_handler: handles the etcd3 events.

async startup_steps() → None

startup_steps: the startup steps for the service container.

- wait for the leader to be elected.
- check and configure the RMQ Exchange and Queues.
- connect the service instance to RMQ.
- start the etcd lease refresh loop.

Returns
None

async start() → bool

- Starts the etcd client if configured
- Startup Steps:
  - Leader election
  - Configures the brokers nuropb service mesh configuration if not done
  - Starts the container service instance.

Returns
None

`async stop() → None`

stop: stops the container service instance.

- primary entry point to stop the service container.

Returns
None

nuropb.interface

Module Contents

Classes

ErrorDescriptionType	
EventType	For compatibility with better futureproof serialisation support, Any encoded payload type is supporte
RequestPayloadDict	Type[RequestPayloadDict]: represents a request that is sent to a service: A request has a response, it
CommandPayloadDict	Type[CommandPayloadDict]: represents a command that is sent to a service: A command has no res
EventPayloadDict	Type[EventPayloadDict]: represents an event that is published to a topic:
ResponsePayloadDict	Type[ResponsePayloadDict]: represents a response to a request:
TransportServicePayload	Type[TransportServicePayload]: represents valid service instruction encoded payload. Depending on
TransportRespondPayload	Type[TransportRespondPayload]: represents valid service response message, valid nuropb encoded p
NuropbInterface	NuropbInterface: represents the interface that must be implemented by a nuropb API implementation

Data

logger	
NUROPB_VERSION	
NUROPB_PROTOCOL_VERSION	
NUROPB_PROTOCOL_VERSIONS_SUPPORTED	
NUROPB_MESSAGE_TYPES	
NuropbSerializeType	
NuropbMessageType	
NuropbLifecycleState	
PayloadDict	
ServicePayloadTypes	
ResponsePayloadTypes	
ResultFutureResponsePayload	
ResultFutureAny	
AcknowledgeAction	
AcknowledgeCallbackFunction	AcknowledgeCallbackFunction: represents a callable with the inputs:
MessageCompleteFunction	MessageCompleteFunction: represents a callable with the inputs:
MessageCallbackFunction	MessageCallbackFunction: represents a callable with the inputs:
ConnectionCallbackFunction	ConnectionCallbackFunction: represents a callable with the inputs:

API

nuropb.interface.**logger**

None

nuropb.interface.**NUROPB_VERSION**

'0.1.8'

nuropb.interface.**NUROPB_PROTOCOL_VERSION**

'0.1.1'

nuropb.interface.**NUROPB_PROTOCOL_VERSIONS_SUPPORTED**

('0.1.1',)

nuropb.interface.**NUROPB_MESSAGE_TYPES**

('request', 'response', 'event', 'command')

nuropb.interface.**NuropbSerializeType**

None

nuropb.interface.**NuropbMessageType**

None

nuropb.interface.**NuropbLifecycleState**

None

class nuropb.interface.**ErrorDescriptionType**

Bases: typing.TypedDict

error: str

None

description: Optional[str]

None

context: Optional[Dict[str, Any]]

None

class nuropb.interface.**EventType**

Bases: typing.TypedDict

For compatibility with better futureproof serialisation support, Any encoded payload type is supported.It is encouraged to use a json compatible key/value Type e.g. Dict[str, Any]

Target
is currently provided here as an aid for the implementation, there are use cases where events are targeted to a specified audience or list or targets. In the NuroPb paradigm, targets could be individual users or other services. A service would represent the service as a whole, NOT any individual instance of that service.

It is also advised not to use NuroPb for communication between instances of a service.

Reference the notes on EventPayloadDict

Initialization

Initialize self. See help(type(self)) for accurate signature.

topic: str

None

payload: Any

None

target: Optional[List[Any]]

None

class nuropb.interface.**RequestPayloadDict**

Bases: typing.TypedDict

Type[RequestPayloadDict]: represents a request that is sent to a service: A request has a response, it is acknowledged by the transport layer after the destination service has handled the request.

REMINDER FOR FUTURE: It is very tempting to support the Tuple[...,Any] for the param key. There is much broader downstream compatibility in keeping this as a dictionary / key-value mapping.

Initialization

Initialize self. See help(type(self)) for accurate signature.

tag: Literal[request]

None

correlation_id: str

None

context: Dict[str, Any]

None

trace_id: Optional[str]

None

service: str

None

method: str

None

params: Dict[str, Any]

None

class nuropb.interface.**CommandPayloadDict**

Bases: typing.TypedDict

Type[CommandPayloadDict]: represents a command that is sent to a service: A command has no response, it is acked immediately by the transport layer. The originator of the command has now knowledge of the execution of the command. If a response is required, a request should be used.

A command is useful when used as a type of directed event.

REMINDER FOR FUTURE: It is very tempting to support the Tuple[...,Any] for the param key. There is much broader downstream compatibility in keeping this as a dictionary / key-value mapping.

Initialization

Initialize self. See help(type(self)) for accurate signature.

tag: Literal[command]

None

correlation_id: str

None

context: Dict[str, Any]

None

trace_id: Optional[str]

None

service: str

None

method: str

None

params: Dict[str, Any]

None

class nuropb.interface.**EventPayloadDict**

Bases: typing.TypedDict

Type[EventPayloadDict]: represents an event that is published to a topic:

Target
: is currently provided here as an aid for the implementation, there are use cases where events are targeted to a specified audience or list or targets. In the NuroPb paradigm, targets could be individual users or other services. A service would represent the service as a whole, NOT any individual instance of that service.

Reference the notes on EventType

Initialization

Initialize self. See help(type(self)) for accurate signature.

tag: Literal[nuropb.interface.EventPayloadDict.event]

None

correlation_id: str

None

context: Dict[str, Any]

None

trace_id: Optional[str]

None

topic: str

None

event: Any

None

target: Optional[List[Any]]

None

class nuropb.interface.**ResponsePayloadDict**

Bases: typing.TypedDict

Type[ResponsePayloadDict]: represents a response to a request:

Initialization

Initialize self. See help(type(self)) for accurate signature.

tag: Literal[response]

None

correlation_id: str

None

context: Dict[str, Any]

None

trace_id: Optional[str]

None

result: Any

None

error: Optional[Dict[str, Any]]

None

warning: Optional[str]

None

reply_to: str

None

nuropb.interface.**PayloadDict**

None

nuropb.interface.**ServicePayloadTypes**

None

nuropb.interface.**ResponsePayloadTypes**

None

class nuropb.interface.**TransportServicePayload**

Bases: typing.TypedDict

Type[TransportServicePayload]: represents valid service instruction encoded payload. Depending on the transport implementation, there wire encoding and serialization may be different, and some of the fields may be in the body or header of the message.

Initialization

Initialize self. See help(type(self)) for accurate signature.

nuropb_protocol: str

None

correlation_id: str

None

trace_id: Optional[str]

None

ttl: Optional[int]

None

nuropb_type: nuropb.interface.NuropbMessageType

None

nuropb_payload: Dict[str, Any]

None

class nuropb.interface.**TransportRespondPayload**

Bases: typing.TypedDict

Type[TransportRespondPayload]: represents valid service response message, valid nuropb encoded payload types are ResponsePayloadDict, and EventPayloadDict

Initialization

Initialize self. See help(type(self)) for accurate signature.

nuropb_protocol: str

None

correlation_id: str

None

trace_id: Optional[str]

None

ttl: Optional[int]

None

nuropb_type: nuropb.interface.NuropbMessageType

None

nuropb_payload: nuropb.interface.ResponsePayloadTypes

None

nuropb.interface.**ResultFutureResponsePayload**

None

nuropb.interface.**ResultFutureAny**

None

nuropb.interface.**AcknowledgeAction**

None

nuropb.interface.**AcknowledgeCallbackFunction**

None

AcknowledgeCallbackFunction: represents a callable with the inputs:

- action: AcknowledgeAction # one of "ack", "nack", "reject"

nuropb.interface.**MessageCompleteFunction**

None

MessageCompleteFunction: represents a callable with the inputs:

- response_messages: List[TransportRespondPayload] # the responses to be sent
- acknowledge_action: AcknowledgeAction # one of "ack", "nack", "reject"

nuropb.interface.**MessageCallbackFunction**

None

MessageCallbackFunction: represents a callable with the inputs:

- message: TransportServicePayload # the decoded message
- message_complete: Optional[AcknowledgeCallbackFunction] # a function that is called to acknowledge the message
- metadata: Dict[str: Any] # the context of the message

nuropb.interface.**ConnectionCallbackFunction**

None

ConnectionCallbackFunction: represents a callable with the inputs:

- instance: type of NuropbInterface
- status: str # the status of the connection (connected, disconnected)
- reason: str # the reason for the connection status change

exception nuropb.interface.**NuropbException**(*description: Optional[str] = None*, *payload: Optional[nuropb.interface.PayloadDict] = None*, *exception: Optional[BaseException] = None*)

Bases: Exception

NuropbException: represents a base exception for all exceptions raised by the nuropb API although the input parameters are optional, it is recommended that the message is set to a meaningful value and the nuropb_message is set to the values that were present when the exception was raised.

Initialization

Initialize self. See help(type(self)) for accurate signature.

description: str

None

payload: nuropb.interface.PayloadDict | *nuropb.interface.TransportServicePayload* | *nuropb.interface.TransportRespondPayload* | Dict[str, Any] | None

None

exception: BaseException | None

None

to_dict() → Dict[str, Any]

exception nuropb.interface.**NuropbTimeoutError**(*description: Optional[str] = None*, *payload: Optional[nuropb.interface.PayloadDict] = None*, *exception: Optional[BaseException] = None*)

Bases: *nuropb.interface.NuropbException*

NuropbTimeoutError: represents an error that occurred when a timeout was reached.

Initialization

Initialize self. See help(type(self)) for accurate signature.

exception nuropb.interface.**NuropbTransportError**(*description: Optional[str] = None*, *payload: Optional[nuropb.interface.PayloadDict] = None*, *exception: Optional[BaseException] = None*, *close_connection: bool = False*)

Bases: *nuropb.interface.NuropbException*

NuropbTransportError: represents an error that inside the plumbing.

Initialization

Initialize self. See help(type(self)) for accurate signature.

_close_connection: bool

None

property close_connection: bool

close_connection: returns True if the connection should be closed

exception nuropb.interface.**NuropbMessageError**(*description: Optional[str] = None*, *payload: Optional[nuropb.interface.PayloadDict] = None*, *exception: Optional[BaseException] = None*)

Bases: *nuropb.interface.NuropbException*

NuropbMessageError: represents an error that occurred during the encoding or decoding of a message.

Initialization

Initialize self. See help(type(self)) for accurate signature.

exception nuropb.interface.**NuropbHandlingError**(*description: Optional[str] = None*, *payload: Optional[nuropb.interface.PayloadDict] = None*, *exception: Optional[BaseException] = None*)

Bases: *nuropb.interface.NuropbException*

NuropbHandlingError: represents an error that occurred during the execution or fulfilment of a request or command. An error response is returned to the requester.

Initialization

Initialize self. See help(type(self)) for accurate signature.

exception nuropb.interface.**NuropbDeprecatedError**(*description: Optional[str] = None*, *payload: Optional[nuropb.interface.PayloadDict] = None*, *exception: Optional[BaseException] = None*)

Bases: *nuropb.interface.NuropbHandlingError*

NuropbDeprecatedError: represents an error that occurred during the execution or fulfilment of a request, command or event topic that has been marked deprecated.

Initialization

Initialize self. See help(type(self)) for accurate signature.

exception nuropb.interface.**NuropbValidationError**(*description: Optional[str] = None*, *payload: Optional[nuropb.interface.PayloadDict] = None*, *exception: Optional[BaseException] = None*)

Bases: *nuropb.interface.NuropbException*

NuropbValidationError: represents an error that occurred during the validation of a request or command. An error response is returned to the requester.

An error response is returned to the requester ONLY for requests and commands. Events will be rejected with a NACK with requeue=False.

Initialization

Initialize self. See help(type(self)) for accurate signature.

exception nuropb.interface.**NuropbAuthenticationError**(*description: Optional[str] = None*, *payload: Optional[nuropb.interface.PayloadDict] = None*, *exception: Optional[BaseException] = None*)

Bases: *nuropb.interface.NuropbException*

NuropbAuthenticationError: when this exception is raised, the transport layer will ACK the message and return an authentication error response to the requester.

This exception occurs whe the identity of the requester can not be validated. for example an unknown, invalid or expired user identifier or auth token.

In most cases, the requester will not be able to recover from this error and will need provide valid credentials and retry the request. The approach to this retry outside the scope of the nuropb API.

Initialization

Initialize self. See help(type(self)) for accurate signature.

exception nuropb.interface.**NuropbAuthorizationError**(*description: Optional[str] = None*, *payload: Optional[nuropb.interface.PayloadDict] = None*, *exception: Optional[BaseException] = None*)

Bases: *nuropb.interface.NuropbException*

NuropbAuthorizationError: when this exception is raised, the transport layer will ACK the message and return an authorization error response to the requester.

This exception occurs whe the requester does not have the required privileges to perform the requested action of either a request or command.

In most cases, the requester will not be able to recover from this error and will need provide valid credentials and retry the request. The approach to this retry outside the scope of the nuropb API.

Initialization

Initialize self. See help(type(self)) for accurate signature.

exception nuropb.interface.**NuropbNotDeliveredError**(*description: Optional[str] = None*, *payload: Optional[nuropb.interface.PayloadDict] = None*, *exception: Optional[BaseException] = None*)

Bases: *nuropb.interface.NuropbException*

NuropbNotDeliveredError: when this exception is raised, the transport layer will ACK the message and return an error response to the requester.

Initialization

Initialize self. See help(type(self)) for accurate signature.

exception nuropb.interface.**NuropbCallAgainReject**(*description: Optional[str] = None*, *payload: Optional[nuropb.interface.PayloadDict] = None*, *exception: Optional[BaseException] = None*)

Bases: *nuropb.interface.NuropbException*

NuropbCallAgainReject: when this exception is raised, the transport layer will REJECT the message

To prevent accidental use of the redelivered parameter and to ensure system predictability on the Call Again feature, messages are only allowed to be redelivered once and only once. To this end all messages that have redelivered == True will be rejected.

Initialization

Initialize self. See help(type(self)) for accurate signature.

exception nuropb.interface.**NuropbCallAgain**(*description: Optional[str] = None*, *payload: Optional[nuropb.interface.PayloadDict] = None*, *exception: Optional[BaseException] = None*)

Bases: *nuropb.interface.NuropbException*

NuropbCallAgain: when this exception is raised, the transport layer will NACK the message and schedule it to be redelivered. The delay is determined by the transport layer or message broker. A call again will result in a forced repeated call of the original message, with the same correlation_id and trace_id.

The call again "feature" is ignored for event service messages and response messages, as these are acked in all cases. The call again feature by implication is only supported for request and command messages.

WARNING: with request messages, if a response has been returned, then this pattern SHOULD NOT be used. The requester will receive the same response again, which will be ignored as an unpaired response. if the underlying service method has no idempotence guarantees, the service could end up in an inconsistent state.

Initialization

Initialize self. See help(type(self)) for accurate signature.

exception nuropb.interface.**NuropbSuccess**(*result: Any*, *description: Optional[str] = None*, *payload: Optional[*nuropb.interface.ResponsePayloadDict*] = None*, *events: Optional[List[*nuropb.interface.EventType*]] = None*)

Bases: *nuropb.interface.NuropbException*

NuropbSuccessError: when this exception is raised, the transport layer will ACK the message and return a success response if service encoded payload is a 'request'. This is useful when the request is a command or event and is executed asynchronously.

There are some use cases where the service may want to return a success response irrespective to the handling of the request.

A useful example is to short circuit processing when an outcome can be predetermined from the inputs alone. For end to end request-response consistency, this class must be instantiated with ResponsePayloadDict that contains a result consistent with the method and inputs provided.

Another use case is for the transmission of events raised during the execution of an event, command or request. Events will only be sent to the transports layer after the successful processing of a service message.

Initialization

Initialize self. See help(type(self)) for accurate signature.

result: **Any**

None

payload: *nuropb.interface.ResponsePayloadDict* **| None**

None

events: **List[***nuropb.interface.EventType***]**

[]

class nuropb.interface.**NuropbInterface**

Bases: abc.ABC

NuropbInterface: represents the interface that must be implemented by a nuropb API implementation

_service_name: **str**

None

_instance_id: **str**

None

_service_instance: object

None

property service_name: str

service_name: returns the service name

property instance_id: str

instance_id: returns the instance id

abstract async connect() → None

connect: waits for the underlying transport to connect, an exception is raised if the connection fails to be established

Returns
None

abstract async disconnect() → None

disconnect: disconnects from the underlying transport

Returns
None

abstract property connected: bool

connected: returns the connection status of the underlying transport

Returns
bool

abstract property is_leader: bool

is_leader: returns the leader status of the service instance

Returns
bool

abstract receive_transport_message(*service_message:* nuropb.interface.TransportServicePayload, *message_complete_callback: nuropb.interface.MessageCompleteFunction*, *metadata: Dict[str, Any]*) → None

handle_message: does the processing of a NuroPb message received from the transport layer.

All response, request, command and event messages received from the transport layer are handled here.

For failures service messages are handled, other than for events, a response including details of the error is returned to the flow originator.

Parameters

- **service_message** – TransportServicePayload
- **message_complete_callback** – MessageCompleteFunction
- **metadata** – Dict[str, Any] - metric gathering information

Returns
None

abstract async request(*service: str*, *method: str*, *params: Dict[str, Any]*, *context: Dict[str, Any]*, *ttl: Optional[int] = None*, *trace_id: Optional[str] = None*, *rpc_response: bool = True*) → Union[*nuropb.interface.ResponsePayloadDict*, Any]

request: sends a request to the target service and waits for a response. It is up to the implementation to manage message idempotency and message delivery guarantees.

Parameters

- **service** – the service name
- **method** – the method name
- **params** – the method arguments, these must be easily serializable to JSON
- **context** – additional information that represent the context in which the request is executed. The must be easily serializable to JSON.
- **ttl** – the time to live of the request in milliseconds. After this time and dependent on the state and underlying transport, it will not be consumed by the target service and should be assumed by the requester to have failed with an undetermined state.
- **trace_id** – an identifier to trace the request over the network (e.g. uuid4 hex string)
- **rpc_response** – if True (default), the actual response of the RPC call is returned and where there was an error, that is raised as an exception. Where rpc_response is a ResponsePayloadDict, it is returned.

Returns
ResponsePayloadDict

abstract command(*service: str*, *method: str*, *params: Dict[str, Any]*, *context: Dict[str, Any]*, *ttl: Optional[int] = None*, *trace_id: Optional[str] = None*) → None

command: sends a command to the target service. It is up to the implementation to manage message idempotency and message delivery guarantees.

any response from the target service is ignored.

Parameters

- **service** – the service name
- **method** – the method name
- **params** – the method arguments, these must be easily serializable to JSON
- **context** – additional information that represent the context in which the request is executed. The must be easily serializable to JSON.
- **ttl** – the time to live of the request in milliseconds. After this time and dependent on the underlying transport, it will not be consumed by the target or assumed by the requester to have failed with an undetermined state.
- **trace_id** – an identifier to trace the request over the network (e.g. uuid4 hex string)

Returns
None

abstract publish_event(*topic: str*, *event: Any*, *context: Dict[str, Any]*, *trace_id: Optional[str] = None*) → None

publish_event: publishes an event to the transport layer. the event sender should not have any open transaction that is waiting for a response from this message. It is up to the implementation to manage message idempotency and message delivery guarantees.

Parameters

- **topic** – the topic to publish to
- **event** – the message to publish, must be easily serializable to JSON
- **context** – additional information that represent the context in which the event is executed. The must be easily serializable to JSON.

- **trace_id** – an identifier to trace the request over the network (e.g. uuid4 hex string)

Returns
None

nuropb.utils

Module Contents

Functions

obfuscate_credentials	obfuscate_secret: obfuscate the password in the AMQP url

API

nuropb.utils.**obfuscate_credentials**(*url_with_credentials: str | Dict[str, Any]*) → str

obfuscate_secret: obfuscate the password in the AMQP url

Parameters
url_with_credentials –

Returns
str

nuropb.rmq_transport

Module Contents

Classes

RabbitMQConfiguration	
RMQTransport	RMQTransport is the base class for the RabbitMQ transport. It wraps the NuroPb service mesh patterns

Functions

verbose	
decode_rmq_body	Map the incoming RabbitMQ message to python compatible dictionary as defined by ServicePayloadDict

Data

logger	
CONSUMER_CLOSED_WAIT_TIMEOUT	The wait when shutting down consumers before closing the connection
_verbose	

API

class nuropb.rmq_transport.**RabbitMQConfiguration**

Bases: typing.TypedDict

rpc_exchange: str

None

events_exchange: str

None

dl_exchange: str

None

dl_queue: str

None

service_queue: str

None

response_queue: str

None

rpc_bindings: List[str]

None

event_bindings: List[str]

None

default_ttl: int

None

client_only: bool

None

nuropb.rmq_transport.**logger**

None

nuropb.rmq_transport.**CONSUMER_CLOSED_WAIT_TIMEOUT**

10

The wait when shutting down consumers before closing the connection

nuropb.rmq_transport.**_verbose**

False

nuropb.rmq_transport.**verbose**() → bool

nuropb.rmq_transport.**decode_rmq_body**(*method: pika.spec.Basic.Deliver*, *properties: pika.spec.BasicProperties*, *body: bytes*) → *nuropb.interface.TransportServicePayload*

Map the incoming RabbitMQ message to python compatible dictionary as defined by ServicePayloadDict

exception nuropb.rmq_transport.**ServiceNotConfigured**

Bases: Exception

Raised when a service is not properly configured on the RabbitMQ broker. the leader will be expected to configure the Exchange and service queues

Initialization

Initialize self. See help(type(self)) for accurate signature.

class nuropb.rmq_transport.**RMQTransport**(*service_name: str*, *instance_id: str*, *amqp_url: str | Dict[str, Any]*, *message_callback: nuropb.interface.MessageCallbackFunction*, *default_ttl: Optional[int] = None*, *client_only: Optional[bool] = None*, *encryptor: Optional[*nuropb.encodings.encryption.Encryptor*] = None*, ***kwargs: Any*)

RMQTransport is the base class for the RabbitMQ transport. It wraps the NuroPb service mesh patterns and rules over the AMQP protocol.

When RabbitMQ closes the connection, this class will stop and alert that reconnection is necessary, in some cases re-connection takes place automatically. Disconnections should be continuously monitored, there are various reasons why a connection or channel may be closed after being successfully opened, and usually related to authentication, permissions, protocol violation or networking.

TODO: Configure the Pika client connection attributes in the pika client properties.

Initialization

Create a new instance of the consumer class, passing in the AMQP URL used to connect to RabbitMQ.

Parameters

- **service_name** (*str*) – The name of the service
- **instance_id** (*str*) – The instance id of the service
- **amqp_url** (*str*) – The AMQP url to connect string or TLS connection details
  - cafile: str | None
  - certfile: str | None
  - keyfile: str | None
  - verify: bool (default True)
  - hostname: str
  - port: int
  - username: str
  - password: str
- **message_callback** (*MessageCallbackFunction*) – The callback to call when a message is received

- **default_ttl** (*int*) – The default time to live for messages in milliseconds, defaults to 12 hours.
- **client_only** (*bool*) –
- **encryptor** (Encryptor) – The encryptor to use for encrypting and decrypting messages
- **kwargs** – Mostly transport configuration options
  - str rpc_exchange: The name of the RPC exchange
  - str events_exchange: The name of the events exchange
  - str dl_exchange: The name of the dead letter exchange
  - str dl_queue: The name of the dead letter queue
  - str service_queue: The name of the requests queue
  - str response_queue: The name of the responses queue
  - List[str] rpc_bindings: The list of RPC bindings
  - List[str] event_bindings: The list of events bindings
  - int prefetch_count: The number of messages to prefetch defaults to 1, unlimited is 0. Experiment with larger values for higher throughput in your user case.

  When an existing transport is initialised and connected, and a subsequent transport instance is connected with the same service_name and instance_id as the first, the broker will shut down the channel of subsequent instances when they attempt to configure their response queue. This is because the response queue is opened in exclusive mode. The exclusive mode is used to ensure that only one consumer (nuropb api connection) is consuming from the response queue.

  Deliberately specifying a fixed instance_id, is a valid mechanism to ensure that a service can only run in single instance mode. This is useful for services that are not designed to be run in a distributed manner or where there is specific service configuration required.

_service_name: str

None

_instance_id: str

None

_amqp_url: str | Dict[str, Any]

None

_rpc_exchange: str

None

_events_exchange: str

None

_dl_exchange: str

None

_dl_queue: str

None

_service_queue: str

None

_response_queue: str
None

_rpc_bindings: Set[str]
None

_event_bindings: Set[str]
None

_prefetch_count: int
None

_default_ttl: int
None

_client_only: bool
None

_message_callback: nuropb.interface.MessageCallbackFunction
None

_encryptor: *nuropb.encodings.encryption.Encryptor* | None
None

_connected_future: Any
None

_disconnected_future: Any
None

_is_leader: bool
None

_is_rabbitmq_configured: bool
None

_connection: pika.adapters.asyncio_connection.AsyncioConnection | None
None

_channel: pika.channel.Channel | None
None

_consumer_tags: Set[Any]
None

_consuming: bool
None

_connecting: bool
None

_closing: bool
None

_connected: bool
None

_was_consuming: bool
None

property service_name: str

property instance_id: str

property amqp_url: str | Dict[str, Any]

property is_leader: bool

property connected: bool

connected: returns the connection status of the underlying transport

Returns
bool

property rpc_exchange: str

rpc_exchange: returns the name of the RPC exchange

Returns
str

property events_exchange: str

events_exchange: returns the name of the events exchange

Returns
str

property response_queue: str

response_queue: returns the name of the response queue

Returns
str

property rmq_configuration: *nuropb.rmq_transport.RabbitMQConfiguration*

rmq_configuration: returns the RabbitMQ configuration

Returns
Dict[str, Any]

configure_rabbitmq(*rmq_configuration: Optional[*nuropb.rmq_transport.RabbitMQConfiguration*] = None*, *amqp_url: Optional[str | Dict[str, Any]] = None*, *rmq_api_url: Optional[str] = None*) → None

configure_rabbitmq: configure the RabbitMQ transport with the provided configuration

if rmq_configuration is None, then the transport will be configured with the configuration provided during the transport's **init**().

if the virtual host in the build_amqp_url is not configured, then it will be created.

Parameters

- **rmq_configuration** – RabbitMQConfiguration
- **amqp_url** – Optional[str] if not provided then self._amqp_url is used
- **rmq_api_url** – Optional[str] if not provided then it is created amqp_url

Returns
None

async start() → None

Start the transport by connecting to RabbitMQ

async stop() → None

Cleanly shutdown the connection to RabbitMQ by stopping the consumer with RabbitMQ. When RabbitMQ confirms the cancellation, on_cancelok will be invoked by pika, which will then closing the channel and connection. The IOLoop is started again because this method is invoked when CTRL-C is pressed raising a KeyboardInterrupt exception. This exception stops the IOLoop which needs to be running for pika to communicate with RabbitMQ. All commands issued prior to starting the IOLoop will be buffered but not processed.

connect() → asyncio.Future[bool]

This method initiates a connection to RabbitMQ, returning the connection handle. When the connection is established, the on_connection_open method will be invoked by pika.

When the connection and channel is successfully opened, the incoming messages will automatically be handled by _handle_message()

Return type
asyncio.Future

disconnect() → Awaitable[bool]

This method closes the connection to RabbitMQ. the pika library events will drive the closing and reconnection process.

Returns
asyncio.Future

on_connection_open(*_connection: pika.adapters.asyncio_connection.AsyncioConnection*) → None

This method is called by pika once the connection to RabbitMQ has been established. It passes the handle to the connection object in case we need it, but in this case, we'll just mark it unused.

Parameters
_connection (*pika.adapters.asyncio_connection.AsyncioConnection*) – The connection

on_connection_open_error(*conn: pika.adapters.asyncio_connection.AsyncioConnection*, *reason: Exception*) → None

This method is called by pika if the connection to RabbitMQ can't be established.

Parameters

- **conn** (*pika.adapters.asyncio_connection.AsyncioConnection*) –
- **reason** (*Exception*) – The error

on_connection_closed(*_connection: pika.adapters.asyncio_connection.AsyncioConnection*, *reason: Exception*) → None

This method is invoked by pika when the connection to RabbitMQ is closed unexpectedly. Since it is unexpected, we will reconnect to RabbitMQ if it disconnects.

Parameters

- **_connection** (*pika.connection.Connection*) – The closed connection obj
- **reason** (*Exception*) – exception representing reason for loss of connection.

open_channel() → None

Open a new channel with RabbitMQ by issuing the Channel.Open RPC command. When RabbitMQ responds that the channel is open, the on_channel_open callback will be invoked by pika.

on_channel_open(*channel: pika.channel.Channel*) → None

This method is invoked by pika when the channel has been opened. The channel object is passed in so that we can make use of it.

Parameters
channel (*pika.channel.Channel*) – The channel object

on_channel_closed(*channel: pika.channel.Channel*, *reason: Exception*) → None

Invoked by pika when the channel is closed. Channels are at times close by the broker for various reasons, the most common being protocol violations e.g. acknowledging messages using an invalid message_tag. In most cases when the channel is closed by the broker, nuropb will automatically open a new channel and re-declare the service queue.

In the following cases the channel is not automatically re-opened: * When the connection is closed by this transport API * When the connection is close by the broker 403 (ACCESS_REFUSED): Typically these examples are seen: - "Provided JWT token has expired at timestamp". In this case the transport will require a fresh JWT token before attempting to reconnect. - queue 'XXXXXXX' in vhost 'YYYYY' in exclusive use. In this case there is another response queue setup with the same name. Having a fixed response queue name is a valid mechanism to enforce a single instance of a service. * When the connection is close by the broker 403 (NOT_FOUND): Typically where there is an exchange configuration issue.

Always investigate the reasons why a channel is closed and introduce logic to handle that scenario accordingly. It is important to continuously monitor for this condition.

TODO: When there is message processing in flight, and particularly with a prefetch count > 1, then those messages are now not able to be acknowledged. By doing so will result in a forced channel closure by the broker and potentially a poison pill type scenario.

Parameters

- channel (*pika.channel.Channel*) – The closed channel
- reason (*Exception*) – why the channel was closed

declare_service_queue(*frame: pika.frame.Method*) → None

Refresh the request queue on RabbitMQ by invoking the Queue.Declare RPC command. When it is complete, the on_service_queue_declareok method will be invoked by pika.

This call is idempotent and will not fail if the queue already exists.

on_service_queue_declareok(*frame: pika.frame.Method*, *_userdata: str*) → None

declare_response_queue() → None

Set up the response queue on RabbitMQ by invoking the Queue.Declare RPC command. When it is complete, the on_response_queue_declareok method will be invoked by pika.

on_response_queue_declareok(*frame: pika.frame.Method*, *_userdata: str*) → None

Method invoked by pika when the Queue.Declare RPC call made in setup_response_queue has completed. In this method we will bind request queue and the response queues. When this command is complete, the on_bindok method will be invoked by pika.

No explicit binds required for the response queue as it relies on the default exchange and routing key to the name of the queue.

Parameters

- frame (*pika.frame.Method*) – The Queue.DeclareOk frame
- _userdata (*str|unicode*) – Extra user data (queue name)

on_bindok(*_frame: pika.frame.Method*, *userdata: str*) → None

Invoked by pika when the Queue.Bind method has completed. At this point we will set the prefetch count for the channel.

Parameters

- **_frame** (*pika.frame.Method*) – The Queue.BindOk response frame
- **userdata** (*str|unicode*) – Extra user data (queue name)

on_basic_qos_ok(*_frame: pika.frame.Method*) → None

Invoked by pika when the Basic.QoS method has completed. At this point we will start consuming messages.

A callback is added that will be invoked if RabbitMQ cancels the consumer for some reason. If RabbitMQ does cancel the consumer, on_consumer_cancelled will be invoked by pika.

Parameters

_frame (*pika.frame.Method*) – The Basic.QosOk response frame

on_consumer_cancelled(*method_frame: pika.frame.Method*) → None

Invoked by pika when RabbitMQ sends a Basic.Cancel for a consumer receiving messages.

Parameters

method_frame (*pika.frame.Method*) – The Basic.Cancel frame

on_message_returned(*channel: pika.channel.Channel*, *method: pika.spec.Basic.Return*, *properties: pika.spec.BasicProperties*, *body: bytes*) → None

Called when message has been rejected by the server.

callable callback: The function to call, having the signature callback(channel, method, properties, body)

Parameters

- **channel** – pika.Channel
- **method** – pika.spec.Basic.Deliver
- **properties** – pika.spec.BasicProperties
- **body** – bytes

send_message(*payload: Dict[str, Any]*, *expiry: Optional[int] = None*, *priority: Optional[int] = None*, *encoding: str = 'json'*, *encrypted: bool = False*) → None

Send a message to over the RabbitMQ Transport

```
TODO: Consider alternative handling when the channel is closed.
    also refer to the notes in the on_channel_closed method.
    - Wait and retry on a new channel?
    - setup a retry queue?
    - should there be a high water mark for the number of retries?
    - should new messages not be consumed until the channel is re-established␣
↪and retry queue drained?
```

Parameters

- **payload** (*Dict[str, Any]*) – The message contents
- **expiry** – The message expiry in milliseconds
- **priority** – The message priority

- **encoding** – The encoding of the message
- **encrypted** – True if the message is to be encrypted

classmethod acknowledge_service_message(*channel: pika.channel.Channel*, *delivery_tag: int*, *action: Literal[ack, nack, reject]*, *redelivered: bool*) → None

Acknowledgement of a service message

In NuroPb, acknowledgements of service message requests have three possible outcomes:

- ack: Successfully processed, acknowledged and removed from the queue
- nack: A recoverable error occurs, the message is not acknowledged and requeued
- reject: An unrecoverable error occurs, the message is not acknowledged and dropped

To prevent accidental use of the redelivered parameter and to ensure system predictability on the Call Again feature, messages are only allowed to be redelivered once and only once. To this end all messages that have redelivered == True will be rejected. if redelivered is overridden with None, it is assumed True.

Parameters

- **channel** (*pika.channel.Channel*) – The channel object
- **delivery_tag** (*int*) – The delivery tag
- **action** (*str*) – The action to take, one of ack, nack or reject
- **redelivered** (*bool*) – True if the message is being requeued / replayed.

classmethod metadata_metrics(*metadata: Dict[str, Any]*) → None

Invoked by the transport after a service message has been processed.

NOTE - METADATA: keep this metadata in sync with across all these methods: - on_service_message, on_service_message_complete - on_response_message, on_response_message_complete - metadata_metrics

Parameters
metadata – information to drive message processing metrics

Returns
None

on_service_message_complete(*channel: pika.channel.Channel*, *basic_deliver: pika.spec.Basic.Deliver*, *properties: pika.spec.BasicProperties*, *private_metadata: Dict[str, Any]*, *response_messages: List[*nuropb.interface.TransportRespondPayload*]*, *acknowledgement: nuropb.interface.AcknowledgeAction*) → None

Invoked by the implementation after a service message has been processed.

This is provided to the implementation as a helper function to complete the message flow. The message flow state parameters: channel, delivery_tag, reply_to and private_metadata are hidden from the implementation through the use of functools.partial. The interface of this function as it appears to the implementation is: response_messages: List[TransportRespondPayload] acknowledgement: Literal[“ack”, “nack”, “reject”]

NOTE: The acknowledgement references the incoming service message that resulted in these responses

Parameters

- **channel** –
- **basic_deliver** (*pika.spec.Basic.Deliver*) – basic_deliver method
- **properties** (*pika.spec.BasicProperties*) – properties
- **private_metadata** – information to drive message processing metrics

- **response_messages** – List[TransportRespondPayload]
- **acknowledgement** –

Returns

on_service_message(*queue_name: str*, *channel: pika.channel.Channel*, *basic_deliver: pika.spec.Basic.Deliver*, *properties: pika.spec.BasicProperties*, *body: bytes*) → None

Invoked when a message is delivered to the service_queue.

DESIGN PARAMETERS: - Incoming service messages handling (this) require a deliberate acknowledgement. * Acknowledgements must be sent on the same channel and with the same delivery tag - Incoming response messages (on_response_message) are automatically acknowledged - Not all service messages require a response, e.g. events and commands - Transport must remain agnostic to the handling of the service message

OTHER DESIGN CONSIDERATIONS: - Connections to the RabbitMQ broker should be authenticated, and encrypted using TLS - Payload data not used for NuroPb message handling should be separately encrypted, and specially when containing sensitive data e.g. PI or PII .

FLOW DESIGN:

1. Python json->dict compatible message is received from the service request (rpc) queue
2. message is decoded into a python typed dictionary: TransportServicePayload
3. this message is passed to the message_callback which has no return value, but may raise any type of exception.
   - message_callback is provided by the implementation and supplied on transport instantiation
   - message_callback is responsible for handling all exceptions, acknowledgements and appropriate responses
   - a helper function is provided that will handle message acknowledgement and any responses. the function has all the required state to do so. See note 4 below.
4. State required to be preserved until the message is acknowledged:
   - channel + delivery_tag
   - reply_to
   - correlation_id
   - trace_id
   - incoming decoded message
5. This state can only exist in-memory as it's not all "picklable" and bound to the current open channel
6. If an exception is raised, then the message is immediately rejected and dropped.

(transport) -> on_service_message -> (transport) message_callback(TransportServicePayload, MessageCompleteFunction, Dict[str, Any]) -> (api) message_complete_callback(List[TransportRespondPayload], AcknowledgeAction) -> (transport) acknowledge_service_message(channel, delivery_tag, action) -> (transport) send_response_messages(reply_to, response_messages) -> (transport) metadata_metrics(metadata)

TODO: Needing to think about excessive errors and decide on a strategy for for shutting down

the service instance. Should this take place in the transport layer or the API ?

- what happens to the result if the channel is closed?

- What happens if the request is resent, can we leverage the existing result

- what happens if the request is resent to another service worker?

- what happens to the request sender waiting for a response?

NOTES:

- Call Again: When a message is nack'd and requeued, there is no current way to track how many times this may have occurred for the same message. To ensure stability, behaviour predictability and to limit the abuse of patterns, the use of NuropbCallAgain is limited to once and only once. This is enforced at the transport layer. For RabbitMQ if the incoming message is a requeued message, the basic_deliver.redelivered is True. Call again for all redelivered messages will be rejected.

Parameters

- **queue_name** (*str*) – The name of the queue that the message was received on
- **channel** (*pika.channel.Channel*) – The channel object
- **basic_deliver** (*pika.spec.Basic.Deliver*) – basic_deliver method
- **properties** (*pika.spec.BasicProperties*) – properties
- **body** (*bytes*) – The message body

on_response_message_complete(*channel: pika.channel.Channel*, *basic_deliver: pika.spec.Basic.Deliver*, *properties: pika.spec.BasicProperties*, *private_metadata: Dict[str, Any]*, *response_messages: List[*nuropb.interface.TransportRespondPayload*]*, *acknowledgement: nuropb.interface.AcknowledgeAction*) → None

Invoked by the implementation after a service message has been processed.

This is provided to the implementation as a helper function to complete the message flow. The message flow state parameters: channel, delivery_tag, reply_to and private_metadata are hidden from the implementation through the use of functools.partial. The interface of this function as it appears to the implementation is: response_messages: List[TransportRespondPayload] acknowledgement: Literal["ack", "nack", "reject"]

NOTE: The acknowledgement references the incoming service message that resulted in these responses

Parameters

- **channel** –
- **basic_deliver** (*pika.spec.Basic.Deliver*) – basic_deliver method
- **properties** (*pika.spec.BasicProperties*) – properties
- **private_metadata** – information to drive message processing metrics
- **response_messages** – List[TransportRespondPayload]
- **acknowledgement** –

Returns

on_response_message(*_queue_name: str*, *channel: pika.channel.Channel*, *basic_deliver: pika.spec.Basic.Deliver*, *properties: pika.spec.BasicProperties*, *body: bytes*) → None

Invoked when a message is delivered to the response_queue. The channel is passed for your convenience. The basic_deliver object that is passed in carries the exchange, routing key, delivery tag and a redelivered flag for the message. The properties passed in is an instance of BasicProperties with the message properties and the body is the message that was sent.

Parameters

- **_queue_name** (*str*) – The name of the queue that the message was received on
- **channel** (*pika.channel.Channel*) – The channel object
- **basic_deliver** (*pika.spec.Basic.Deliver*) – basic_deliver
- **properties** (*pika.spec.BasicProperties*) – properties
- **body** (*bytes*) – The message body

async stop_consuming() → None

Tell RabbitMQ that you would like to stop consuming by sending the Basic.Cancel RPC command.

close_channel() → None

Call to close the channel with RabbitMQ cleanly by issuing the Channel.Close RPC command.

nuropb.nuropb_api

Factory functions for instantiating nuropb api's.

Module Contents

Classes

MeshService	A generic service class that can be used to create a connection only service instance for the nuropb service mesh. Th

Functions

default_connection_properties	
create_client	Create a client api instance for the nuropb service mesh. This caller of this function will have
connect	
configure_mesh	
create_service	Create a client api instance for the nuropb service mesh. This caller of this function will have

Data

logger

API

nuropb.nuropb_api.**logger**

None

nuropb.nuropb_api.**default_connection_properties**(*connection_properties: Dict[str, Any]*) → Dict[str, Any]

nuropb.nuropb_api.**create_client**(*name: Optional[str] = None*, *instance_id: Optional[str] = None*, *connection_properties: Optional[Dict[str, Any]] = None*, *transport_settings: Optional[str | Dict[str, Any]] = None*, *transport: Optional[*nuropb.rmq_api.RMQAPI*] = RMQAPI*) → *nuropb.rmq_api.RMQAPI*

Create a client api instance for the nuropb service mesh. This caller of this function will have to implement the asyncio call to connect to the service mesh: await client_api.connect()

Parameters

- **name** – used to identify the api connection to the service mesh
- **instance_id** – used to create the service mesh response queue for this api connection
- **connection_properties** – str or dict with values as required for the chosen transport api client
- **transport_settings** – dict with values as required for the underlying transport api
- **transport** – the class of the transport api client to use

Returns

async nuropb.nuropb_api.**connect**(*instance_id: Optional[str] = None*)

nuropb.nuropb_api.**configure_mesh**(*mesh_name: Optional[str] = None*, *connection_properties: Optional[Dict[str, Any]] = None*, *transport_settings: Optional[str | Dict[str, Any]] = None*)

class nuropb.nuropb_api.**MeshService**(*service_name: str*, *instance_id: Optional[str] = None*, *event_bindings: Optional[list[str]] = None*, *event_callback: Optional[Callable] = None*)

A generic service class that can be used to create a connection only service instance for the nuropb service mesh. This class could also be used as a template or to define a subclass for creating a service instance.

Initialization

_service_name: str

None

_instance_id: str

None

_event_bindings: list[str]

None

_event_callback: Optional[Callable]

None

async _handle_event_(*topic: str*, *event: dict*, *target: list[str] | None = None*, *context: dict | None = None*, *trace_id: str | None = None*)

nuropb.nuropb_api.**create_service**(*name: str*, *instance_id: Optional[str] = None*, *service_instance: Optional[object] = None*, *connection_properties: Optional[Dict[str, Any]] = None*, *transport_settings: Optional[str | Dict[str, Any]] = None*, *transport: Optional[*nuropb.rmq_api.RMQAPI*] = RMQAPI*, *event_bindings: Optional[list[str]] = None*, *event_callback: Optional[Callable] = None*) → *nuropb.rmq_api.RMQAPI*

Create a client api instance for the nuropb service mesh. This caller of this function will have to implement the asyncio call to connect to the service mesh: await client_api.connect()

Parameters

- **name** – used to identify this service to the service mesh
- **instance_id** – used to create the service mesh response queue for this individual api connection
- **service_instance** – the instance of the service class that is intended to be exposed to the service mesh
- **connection_properties** – str or dict with values as required for the chosen transport api client
- **transport_settings** – dict with values as required for the underlying transport api
- **transport** – the class of the transport api client to use
- **event_bindings** – when service_instance is None, a list of event topics that this service will subscribe to. when service_instance is not None, the list will override the event_bindings of the transport_settings if any are defined.
- **event_callback** – when service_instance is None, a callback function that will be called when an event is received

Returns

nuropb.rmq_lib

RabbitMQ Utility library for NuroPb

Module Contents

Functions

build_amqp_url	Creates an AMQP URL for connecting to RabbitMQ
build_rmq_api_url	Creates an HTTP URL for connecting to RabbitMQ management API
rmq_api_url_from_amqp_url	Creates an HTTP URL for connecting to RabbitMQ management API from an AMQP U
get_client_connection_properties	Returns the client connection properties for the transport
get_connection_parameters	Return the connection parameters for the transport
management_api_session_info	Creates a requests session for connecting to RabbitMQ management API
blocking_rabbitmq_channel	Useful for initialisation of queues / exchanges.
configure_nuropb_rmq	Configure the RabbitMQ broker for this transport.
nack_message	nack_message: nack the message and requeue it, there was likely a recoverable problem v
reject_message	reject_message: If the message is not a request, then reject the message and move on
ack_message	ack_message: ack the message
get_virtual_host_queues	Creates a virtual host on the RabbitMQ server using the REST API
get_virtual_hosts	Creates a virtual host on the RabbitMQ server using the REST API
create_virtual_host	Creates a virtual host on the RabbitMQ server using the REST API
delete_virtual_host	Deletes a virtual host on the RabbitMQ server using the REST API

Data

logger

API

nuropb.rmq_lib.**logger**

None

nuropb.rmq_lib.**build_amqp_url**(*host: str*, *port: str | int*, *username: str*, *password: str*, *vhost: str*, *scheme: str = 'amqp'*) → str

Creates an AMQP URL for connecting to RabbitMQ

nuropb.rmq_lib.**build_rmq_api_url**(*scheme: str*, *host: str*, *port: str | int*, *username: str | None*, *password: str | None*) → str

Creates an HTTP URL for connecting to RabbitMQ management API

nuropb.rmq_lib.**rmq_api_url_from_amqp_url**(*amqp_url: str*, *scheme: Optional[str] = None*, *port: Optional[int | str] = None*) → str

Creates an HTTP URL for connecting to RabbitMQ management API from an AMQP URL

Parameters

- **amqp_url** – the AMQP URL to use

- **scheme** – the scheme to use, defaults to http
- **port** – the port to use, defaults to 15672

Returns
the RabbitMQ management API URL

nuropb.rmq_lib.**get_client_connection_properties**(*name: Optional[str] = None*, *instance_id: Optional[str] = None*, *client_only: Optional[bool] = None*) → Dict[str, str]

Returns the client connection properties for the transport

nuropb.rmq_lib.**get_connection_parameters**(*amqp_url: str | Dict[str, Any]*, *name: Optional[str] = None*, *instance_id: Optional[str] = None*, *client_only: Optional[bool] = None*, ***overrides: Any*) → pika.ConnectionParameters | pika.URLParameters

Return the connection parameters for the transport

Parameters

- **amqp_url** – the AMQP URL or connection parameters to use
- **name** – the name of the service or client
- **instance_id** – the instance id of the service or client
- **client_only** –
- **overrides** – additional keyword arguments to override the connection parameters

nuropb.rmq_lib.**management_api_session_info**(*scheme: str*, *host: str*, *port: str | int*, *username: Optional[str] = None*, *password: Optional[str] = None*, *bearer_token: Optional[str] = None*, *verify: bool = False*, ***headers: Any*) → Dict[str, Any]

Creates a requests session for connecting to RabbitMQ management API

Parameters

- **scheme** – http or https
- **host** – the host name or ip address of the RabbitMQ server
- **port** – the port number of the RabbitMQ server
- **username** – the username to use for authentication
- **password** – the password to use for authentication
- **bearer_token** – the bearer token to use for authentication
- **verify** – whether to verify the SSL certificate

Returns
a requests session

nuropb.rmq_lib.**blocking_rabbitmq_channel**(*rmq_url: str | Dict[str, Any]*) → pika.channel.Channel

Useful for initialisation of queues / exchanges.

nuropb.rmq_lib.**configure_nuropb_rmq**(*rmq_url: str | Dict[str, Any]*, *events_exchange: str*, *rpc_exchange: str*, *dl_exchange: str*, *dl_queue: str*, ***kwargs: Any*) → bool

Configure the RabbitMQ broker for this transport.

Calls to this function are IDEMPOTENT. However, previously named exchanges, queues, and declared bindings are not be removed. These will have to be done manually as part of broker housekeeping. This is to prevent

accidental removal of queues and exchanges. It is safe to call this function multiple times and while other services are running, as it will not re-declare exchanges, queues, or bindings that already exist.

PRODUCTION AUTHORISATION NOTE: The RabbitMQ user used to connect to the broker must have the following permissions:

- configure: .*
- write: .*
- read: .*
- access to the vhost Client only applications and services should not have configuration permissions. For completeness, there may be specific implementation need to for a client only services to register service queue bindings, for example a client only service that is also a gateway or proxy service. In this case, the treating it as a service is the correct approach.

Settings for Exchange and default dead letter configuration apply to all services that use the same RabbitMQ broker. The rpc and event bindings are exclusive to the service, and are not shared with other services.

The response queues are not durable, and are auto-deleted when the connections close. This approach is taken as response queues are only used for RPC responses, and there is no need to keep and have to handle stale responses.

There is experimental work underway using etcd to manage the runtime configuration of a service leader and service followers. This will allow for persistent response queues and other configuration settings to be shared across multiple instances of the same service. This is not yet ready for production use. Experimentation scope:

- service leader election
- named instances of a service each with their own persistent response queue
- Notification and handling of dead letter messages relating to a service

Parameters

- **rmq_url** (*str*) – The URL of the RabbitMQ broker
- **events_exchange** (*str*) – The name of the events exchange
- **rpc_exchange** (*str*) – The name of the RPC exchange
- **dl_exchange** (*str*) – The name of the dead letter exchange
- **dl_queue** (*str*) – The name of the dead letter queue
- **kwargs** – Additional keyword argument overflow from the transport settings.
  - client_only: bool - True if this is a client only service, False otherwise

Returns

True if the RabbitMQ broker was configured successfully

nuropb.rmq_lib.**nack_message**(*channel: pika.channel.Channel*, *delivery_tag: int*, *properties: pika.spec.BasicProperties*, *mesg: nuropb.interface.PayloadDict | None*, *error: Exception | None = None*) → None

nack_message: nack the message and requeue it, there was likely a recoverable problem with this instance while processing the message

nuropb.rmq_lib.**reject_message**(*channel: pika.channel.Channel*, *delivery_tag: int*, *properties: pika.spec.BasicProperties*, *mesg: nuropb.interface.PayloadDict | None*, *error: Exception | None = None*) → None

reject_message: If the message is not a request, then reject the message and move on

nuropb.rmq_lib.**ack_message**(*channel: pika.channel.Channel*, *delivery_tag: int*, *properties: pika.spec.BasicProperties*, *mesg: nuropb.interface.PayloadDict | None*, *error: Exception | None = None*) → None

ack_message: ack the message

nuropb.rmq_lib.**get_virtual_host_queues**(*api_url: str*, *vhost_url: str*) → Any | None

Creates a virtual host on the RabbitMQ server using the REST API

Parameters

- **api_url** – the url to the RabbitMQ API
- **vhost_url** – the virtual host to create

Returns
None

nuropb.rmq_lib.**get_virtual_hosts**(*api_url: str*, *vhost_url: str | Dict[str, Any]*) → Any | None

Creates a virtual host on the RabbitMQ server using the REST API

Parameters

- **api_url** – the url to the RabbitMQ API
- **vhost_url** – the virtual host to create

Returns
None

nuropb.rmq_lib.**create_virtual_host**(*api_url: str*, *vhost_url: str | Dict[str, Any]*) → None

Creates a virtual host on the RabbitMQ server using the REST API

Parameters

- **api_url** – the url to the RabbitMQ API
- **vhost_url** – the virtual host to create

Returns
None

nuropb.rmq_lib.**delete_virtual_host**(*api_url: str*, *vhost_url: str | Dict[str, Any]*) → None

Deletes a virtual host on the RabbitMQ server using the REST API

Parameters

- **api_url** – the url to the RabbitMQ API
- **vhost_url** – the virtual host to delete

Returns
None

PYTHON MODULE INDEX

e

n

INDEX

Symbols

A

B

C

D

E

G

H

I

J

L

M

N

O

P

U

V

W